SoC-FPGA MobileNets for Embedded Vision Applications

Tiago De Smet

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor(s): Prof. Horácio Cláudio De Campos Neto

Prof. Mário Pereira Véstias

Examination committee

Chairperson: Prof. Teresa Maria Sá Ferreira Vazão Vasques

Supervisor: Prof. Horácio Cláudio De Campos Neto

Member of the Committee: Prof. Paulo Ferreira Godinho Flores

September 2021

Acknowledgments

I would like to thank my supervisor, Professor Horácio Neto for all the guidance, availability and patience, but also for all the knowledge and encouragement he gave me. Without your help Professor, this work would not be possible.

Also, I would like to thank INESC-ID for providing me with the tools to develop the experimental work for this thesis.

Lastly, a big and heartfelt thank you to my girlfriend Catarina Ramos, who was the person who gave me the most strength and who always believed in me. This master's thesis is dedicated to you, Cati.

Abstract

Nowadays, the use of artificial intelligence in many software applications is increasingly common. However, decades of development were necessary, especially at hardware level for the use of artificial intelligence to become viable. MobileNets model developed by Google researchers is used for image classification and primarily suited for embedded systems because of the lighter computation compared to its competitors. This model uses the depthwise separable convolution concept to perform the convolutions and despite being a very optimized model, the processing time of this network may still be high when used on low-cost devices.

This work aimed to develop a hardware/software multiprocessing architecture in an SoC FPGA platform for image classification using MobileNets model. The main contributions of this project are the development of 3 IPs to process the depthwise separable convolution layers, using an effective parallelization and allocation of resources to achieve an efficient multi-processing, and a quantized data model to reduce the memory requirements and improve the communications.

The system was implemented on a Zynq 7010 device using a quantized MobileNets model with 26% of the parameters represented in a 16-bit fixed-point format and 74% of the parameters using a 12-bit custom floating-point representation. This quantization process produced a model 60% smaller than the standard MobileNets with only 0.78% of accuracy loss. The final solution allows the classification of 1 image in 469 ms which corresponds to a speed up of 11 times compared to a software-only solution executing on the embedded ARM processor.

Keywords: Artificial intelligence, Google, Depthwise Separable Convolution, MobileNets, FPGA.

Resumo

Hoje em dia é cada vez mais comum o uso de inteligência artificial em diversas aplicações. No entanto foram precisas décadas de desenvolvimento, sobretudo ao nível de *hardware* para que a inteligência artificial pudesse ser viável. O modelo *MobileNets*, desenvolvido por investigadores da *Google*, é usado para a classificação de imagens e devido ao processamento mais leve que as concorrentes, é sobretudo utilizado nos sistemas embebidos. Este modelo usa o conceito de convolução separável em profundidade e apesar de bastante optimizado, o tempo de processamento pode ainda ser elevado quando usado em dispositivos de baixo custo.

Neste trabalho pretendeu-se desenvolver uma arquitectura de multiprocessamento *hardware/software* numa plataforma SoC FPGA para a classificação de imagens usando a *MobileNets.* Os contributos principais deste projecto são o desenvolvimento de 3 *IPs* para processar as camadas das convoluções separáveis em profundidade, usando para isso, uma paralelização e alocação eficiente dos recursos de forma a obter um multiprocessamento eficaz, e a quantização dos dados para reduzir as necessidades de memória e tempo de comunicação.

O sistema foi implementado num dispositivo Zynq 7010, utilizando um modelo *MobileNets* quantizado com 26% dos parâmetros representados em ponto fixo com 16 *bits* e 74% dos parâmetros usando uma representação de ponto flutuante personalizada de 12 *bits*. Isto permitiu obter um modelo 60% menor com uma perda na precisão de apenas 0.78%. A solução final permite classificar 1 imagem em 469 ms, significando uma aceleração de 11 vezes em relação à solução de software executada no ARM do sistema embebido.

Palavras-chave: Inteligência artificial, *Google*, Convolução Separável em Profundidade, *MobileNets*, *FPGA*.

Table of Contents

List of figures

List of Tables

List of Acronyms

ACP	Accelerator Coherency Port
AI	Artificial Intelligence
AMBA	Advanced Microcontroller Bus Architecture
ANN	Artificial Neural Network
APU	Application Processing Unit
ARM	Advanced RISC Machine
AXI	Advanced eXtensible Interface
Batch Norm	Batch Normalization
BD	Buffer Descriptor
BRAM	Block Random Access Memory
CLB	Configurable Logic Blocks
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DMA	Direct Memory Access
FC	Fully Connected
FF	Flip-Flop
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
FPU	Floating-Point Unit
GP	General-Purpose
HP	High-Performance
IFM	Input Feature Map
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
IOB	Input/Output Blocks

IP	Intellectual Property
KB	Kilobyte
L1	Level 1
L2	Level 2
LUT	Lookup Table
MAC	Multiplications And Accumulation
MB	Megabyte
MME	Matrix Multiplication Engine
OFM	Output Feature Map
PE	Processing Element
PL	Programmable Logic
PS	Processing System
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RGB	Red, Green, Blue
ROM	Read-Only Memory
SoC	System-on-Chip
SRL	Shift Register LUT
VGG	Visual Geometry Group

Chapter 1

Introduction

The objective of this work was to develop a custom hardware/software architecture to execute the MobileNets image classification process on an SoC FPGA. This chapter describes the motivation to accelerate the process for this particular neural network, details the main objectives of the work and presents the thesis outline.

1.1. Motivation

Image recognition is an easy task for humans but it has proved to be a complex problem for machines to perform due to the computational effort involved. The evolution of high-capacity computers and new artificial intelligence (AI) techniques has generated an interest in object classification algorithms. Computer image classification uses a form of AI denominated machine learning, which uses a variety of algorithms that iteratively learn with the available data. This iterative learning is also called training, and is done to create a model to make predictions or decisions without being programmed for that. Specific models can be trained to be used on a wide range of applications such as object detection, medical diagnostic, voice processing, biometry, and many others.

The trained model receives an input (an image for instance) and gives an output (what the image represents), which sounds simple, but in reality, is a very complex process. To perform this classification, these models use the concept of network [1]. A complex system can be broken into simpler elements to be easier to solve, or vice versa, simple elements can be combined to build a complex system, and networks can be used to accomplish this. These networks are composed of a set of nodes and connections between them, used to transfer information. This way, nodes receive inputs and process them to obtain an output, which can be transferred to other nodes.

Nodes can be seen as artificial neurons by the networks, and in this case, the network is defined as an artificial neural network (ANN) [1]. An artificial neuron is inspired by the natural neuron functionality, which receives signals through synapses. When one of these received signals is strong enough, the neuron is activated and also emits a signal, that might be propagated to other neurons and even activate some of them.

The class of ANN covers several architectures, and one very popular kind of ANN used in image recognition is the convolutional neural network (CNN) [2]. CNNs are designed having in mind that the inputs will be images, which allows making the network more adapted to image tasks while reducing the parameters required. Despite this, CNN models have intensive computing and not all are efficient to be used in mobile or embedded devices. Some CNN models have been created to deal with this constraint, which are smaller and lighter than common CNNs, and have a decent accuracy in image classification. One of these models is MobileNets [3] which proposes to reduce the computational effort and resource requirements of standard CNNs, by reducing the number of parameters, allowing the model to be used in mobile and embedded vision applications.

Even with a smaller and lighter model, the implementation on mobile or embedded devices may not be easy. Moreover, some implementations need to process an image in hundreds or tens of milliseconds, which may require very expensive devices. Thus, an efficient and optimized architecture can help to accelerate the model with fewer resources, which represents less power consumption or even cheaper devices.

1.2. Specifications and goals

The main objective of this project was to implement a hardware/software architecture to execute the MobileNets image classification process on a System-on-Chip (SoC) Field-Programmable Gate Array (FPGA). However, three requirements were defined to consider this work successfully developed:

- the architecture should be implemented on a low-cost device to make this work innovative compared to most of the projects available;
- if the implementation required some type of optimization of the MobileNets model, the final result should not incur a loss in accuracy larger than 1% (over the original model);
- the final hardware/software architecture should be able to classify the image in less than 1 second;

To meet the requirements defined, the project had four main goals:

1. Develop a C language program to execute the MobileNets image classification procedure;
2. Develop an algorithm to analyze and quantize the MobileNets model using only fixed-point representation or combine it with a custom floating-point representation. Adapt the previous C program to execute the quantized model;
3. Identify the functions which consume more execution time of the MobileNets image classification procedure and design the hardware capable to process these functions.
4. Implement the architecture in a Xilinx Zynq 7010 device and demonstrate that the MobileNets image classification is processed accurately.

1.3. Thesis Outline

This thesis is organized as follows:

- **Chapter 2** provides a context overview and background knowledge. The concept of artificial neuron and ANN are first introduced. Then, CNN models are described in detail and the most popular architectures are analysed. Finally, the MobileNets model is introduced and explained in detail as well as some possible optimizations.

- **Chapter 3** reviews existing works of CNN model optimizations and CNN implementations on FPGAs. The chapter ends with the selection of the techniques that are most suited for this project resulting from the analyse of other works.

- **Chapter 4** presents the algorithm of the MobileNets image classification process and analyses the memory requirements, number of operations and software execution time. This chapter also evaluates the quantization of the model using different types of representations and selects the most promising for this project.

- **Chapter 5** describes the hardware architecture developed. Firstly, the Zynq-7000 SoC is described, and its communications interface and memory management are analysed. Then the custom hardware modules designed for this project and the way they interact are explained in detail.

- **Chapter 6** describes the MobileNets accelerator and the complete hardware/software architecture, and analyses the results achieved.

- **Chapter 7** presents the conclusion of the project along with future work to be done.

Chapter 2

Background

This chapter presents the basic structure of ANNs and some popular CNNs. First, an explanation of the functionality of the artificial neuron. Then, the difference between simple and deep ANNs is described. The chapter ends with some examples of CNNs and the functionality of MobileNets.

2.1. Artificial neuron

An artificial neuron [4] is a node of the ANN which processes inputs, that can be assigned directly or come from other nodes, and generates an output according to a transfer function. This transfer function, the activation function denoted by $\phi(\cdot)$, maps the resulting value into the desired range. Figure 2.1 presents a mathematical model of the artificial neuron.

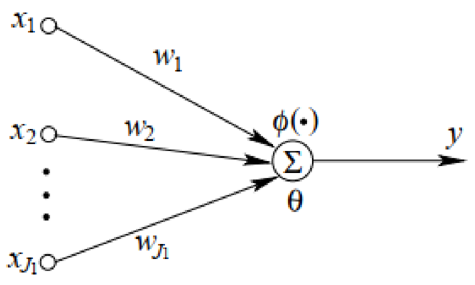


Figure 2.1: The mathematical model of the artificial neuron (from [4]).

The output of the neuron is given by (2.1) and (2.2):

$$net = \sum_{i=1}^{n} w_i x_i - \theta \tag{2.1}$$

$$y = \phi(net) \tag{2.2}$$

where x_i is the ith input, w_i is the weight from the ith input, θ is a threshold or bias, and n is the number of inputs. The earliest and simplest ANN model is called perceptron, used by Rosenblatt [5] to classify linearly separable patterns. This perceptron is a one-neuron layer topology shown in figure 2.1 that uses the hard-limiter activation function defined by equation (2.3):

$$\phi(u) = \begin{cases} 1 \, if \; u > 0 \\ -1 \; (or \; 0) \; otherwise \end{cases} \tag{2.3}$$

Therefore, the output of the perceptron is determined by the parameter θ, which in this case is seen as a threshold used to shift the decision boundary away from the origin. This means that the output of the perceptron can take only two values, and a small variation on the weights or bias can make the output flip from -1 (or 0) to 1, and vice versa, and change the behaviour of the network drastically. To solve this problem, other activations functions are used to allow a more significant range of output values. An example of two activation functions are the logistic function presented in equation (2.4):

$$\phi(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

and the hyperbolic tangent function in equation (2.5):

$$\phi(x) = \tanh(u) = \frac{\sinh(x)}{\cosh(x)} \tag{2.5}$$

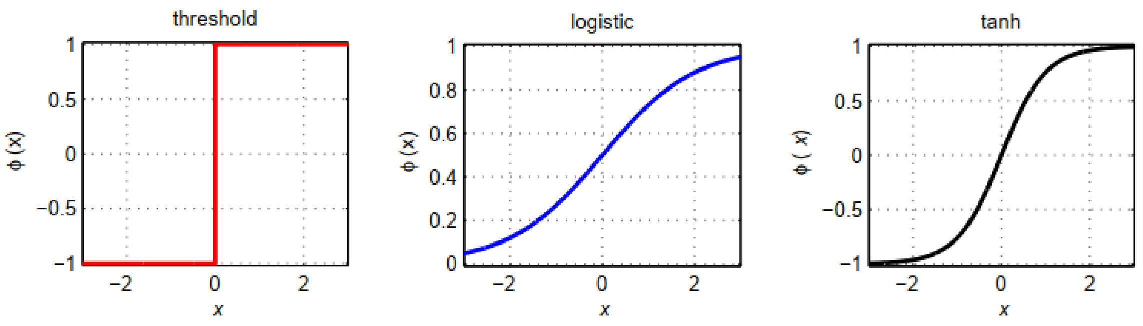


Figure 2.2: Sigmoidal activation functions (from [4]).

The smoothness of the activation function means that small changes in the weights and bias will produce a slight change in the output of the neuron, and not change so drastically the behaviour of the network.

2.2. Neural networks and deep neural networks

A neural network [6] is commonly organized as connections of three or more neuron layers. Each layer is composed of a set of artificial neurons that connect to the neurons of the next layer. A simple neural network consists of an input layer, a hidden layer and an output layer. However, in the present days, the most common architectures have multiple hidden layers. Figure 2.3 shows an example of the two neural networks described.

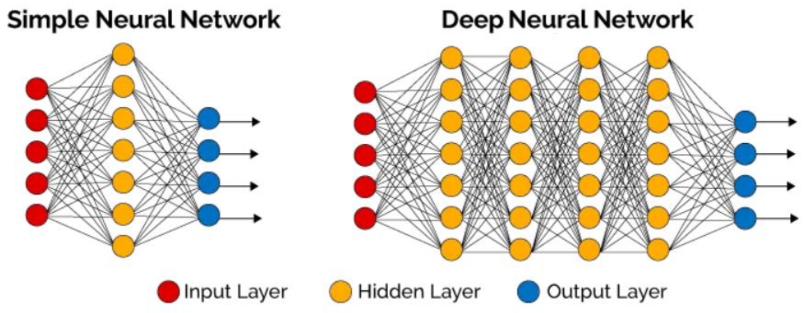


Figure 2.3: Simple neural network and deep neural network (from [7]).

There are several types of ANNs, and their use can be applied to different purposes. To use an ANN on a determined application is necessary to train the network. What training does is to adjust the weights and bias of the neurons to obtain the desired output of the network. Since all the neurons of a fully connected (FC) ANN layer connect to all the next layer neurons (except for the last one), these large networks require a significant number of parameters and therefore become difficult to train. However, for a specific task like image classification, deep ANNs without FC layers can be efficiently trained and are commonly used.

2.3. Convolutional neural networks

CNNs are primarily suited for image-focused tasks like recognition or classification of images. These networks connect only a small region (receptive field) of the input to the neuron. Consider an input image of $100 \times 100 \times 3$ pixels (a 100×100 Red, Green and Blue (RGB) image) and setting the receptive field size to 10×10. Each CNN neuron will have a total of 300 weights ($10 \times 10 \times 3 = 300$), while a traditional ANN neuron would have 100 times more ($100 \times 100 \times 3 = 30{,}000$).

The CNN layers are organized into three dimensions: the height and width that compose the spatial dimensionality of the input, and the depth, which represents the third dimension of an output map. CNN architectures have three types of layers: convolutional layers, pooling layers and FC layers as described below. Figure 2.4 illustrates an example of a CNN architecture applied to image recognition of handwritten digits.

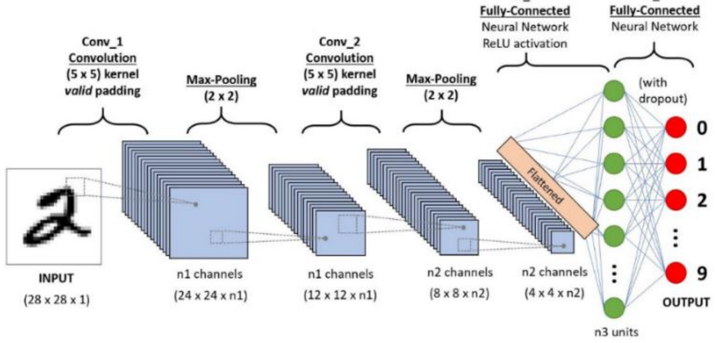


Figure 2.4: Example of a CNN to classify handwritten digits (from [8]).

2.3.1. Convolutional layer

The convolutional layer receives an image as an input and produces a set of output images, by gliding kernels (which is a set of weights) through the input image and calculating the scalar product for each value in that kernel. As a result, the convolutional layer output will be the set of output images produced by all kernels along the depth dimension. These output images are also designated output feature maps (OFMs) because they indicate the locations and intensity of a detected feature on the input image. Figure 2.5 illustrates the convolution of an input image with two sets of three kernels producing two OFMs. From here, a group of kernels will be denominated as a filter.

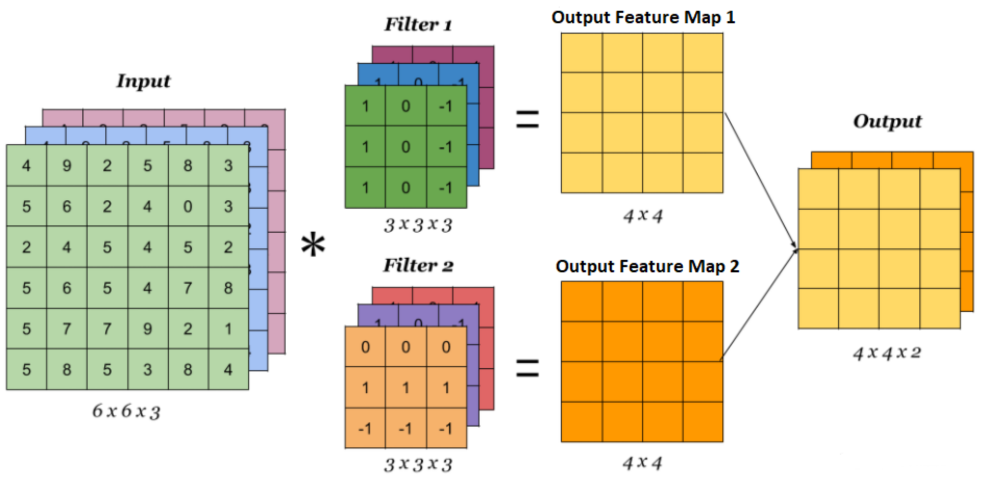


Figure 2.5: Convolution of an input image with two filters producing two OFMs (adapted from [9]).

These OFMs become input feature maps (IFMs) of the following convolutional layer, that will be convolved with new filters to produce new OFMs.

Optimizing the output of the convolutional layers can reduce the complexity of the model. One possible step is reducing the depth of the output volume, which can minimize the number of neurons of the network. However, this procedure may also result in a loss of accuracy. Another optimization is to change stride, which is the step that controls how the kernel convolves around the input volume. For instance, the kernel will convolve around the input image by shifting one unit at a time, if the stride is 1, or two units if the stride is set to 2 (Figure 2.6).

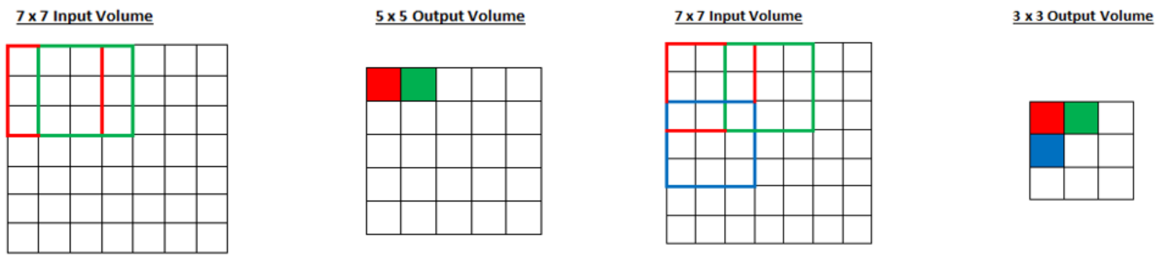


Figure 2.6: Kernel with stride = 1 (left) and stride = 2 (right) applied to an Input image with 7 x 7 dimension (from [10]).

The last process here described is padding with zeros the border of the input, zero-padding (Figure 2.7), which is used to control the size of the output volumes, thus, the output of the convolutional layer. The dimension of the convolutional layer output is given by equation (2.6):

$$O = \frac{(I - K) + 2P}{S} + 1 \tag{2.6}$$

where O is the output height/length, I the input height/length, K is the filter height/length, P the padding and S the stride.

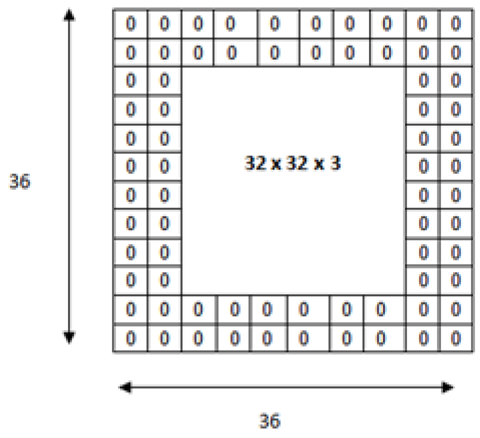


Figure 2.7: Example of a zero-padding of size = 2 (from [10]).

2.3.2. Pooling layer

The Pooling layer reduces the number of parameters and the computational complexity of the model by scaling the dimensionality of each OFM, using a specific function as the max, global or average pooling [11]. In most CNNs the function used is max-pooling, where a 2×2 kernel is applied to the output of the layer to calculate the maximum value, in each 2×2 block of each feature map.

2.3.3. Fully-connected layer

The FC layer is similar to a traditional ANN layer, where all the neurons connect directly to the neurons of the two adjacent layers. When the FC is the last convolutional layer, the number of neurons is equal to the number of classes to predict.

2.3.4. Layer Functions

Some CNNs models use Batch Normalization (Batch Norm) [12] and Rectified Linear Unit (ReLU) [13] non-linearity after performing the convolution. Batch Norm is used to improving the performance and stability of neural networks. All the next layer inputs are normalized so that they have a mean output activation of zero and a standard deviation of one. The ReLU function is an activation function where the output is 0 when $x < 0$, and linear with a slope of 1 when $x > 0$ (figure 2.8).

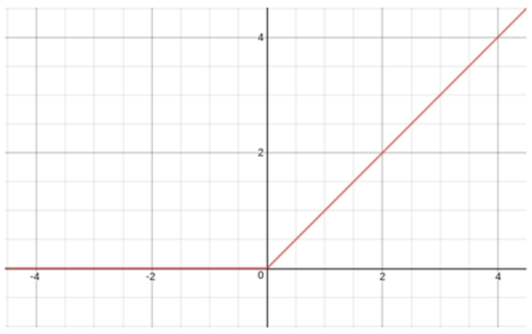


Figure 2.8: The ReLU activation function (from [13]).

2.4. Popular CNN models

AlexNet [14] [15] is one example of a very popular CNN model. Presented in 2012, won the most difficult ImageNet competition for visual object recognition designated the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [15] in the same year. AlexNet is composed of 5 convolution layers and 2 FC layers and about 60 million parameters. Figure 2.9 presents the architecture described.

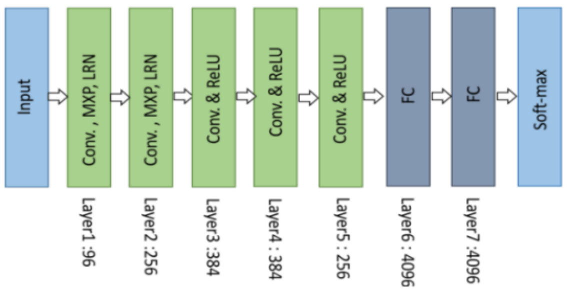


Figure 2.9: Architecture of AlexNet (from [14]).

Two years later, in 2014, the ILSVRC competition was won by GoogleNet [15]. This architecture proposed incorporating what they call “Inception Layers” that had variable receptive fields created by different kernel sizes. Figure 2.10 illustrate the basic version of the inception layer. The GoogleNet model has 22 layers and about 7 million parameters.

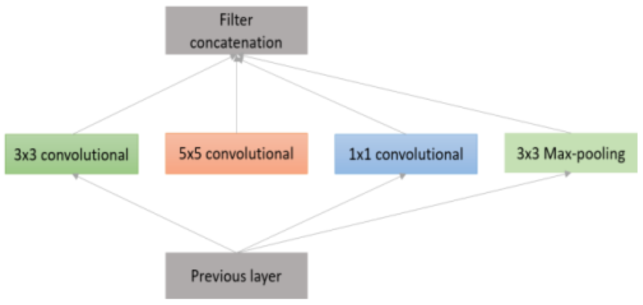


Figure 2.10: Basic version of the inception layer (from [14]).

In the same year (2014), the second place was conquered by the Visual Geometry Group (VGG) [15]. The principal idea to develop this architecture is that the depth of a network is a critical component to achieve better recognition or classification accuracy in CNNs. The main structure of the VGG model is composed of two convolutional layers, each one followed by one ReLU activation function, and a max-pooling layer after these. This structure repeats several times and ends with 3 FC layers (figure 2.11). There are three VGG models: VGG-11, VGG-16 and VGG-19, which represent the number of layers on the model. VGG-16 has 16 layers and 138 million parameters.

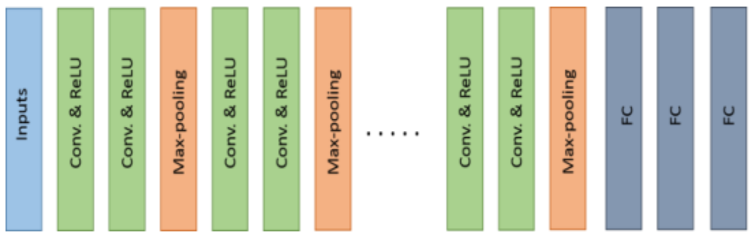


Figure 2.11: Basic architecture of VGG (from [14]).

The last model here presented is SqueezeNet [16], which has the same accuracy as AlexNet but uses 50 times fewer parameters. This architecture has 10 layers, 2 convolution layers and 8 squeeze convolution layers with 1×1 kernels, which they define as “fire” layers. This architecture is shown in figure 2.12.

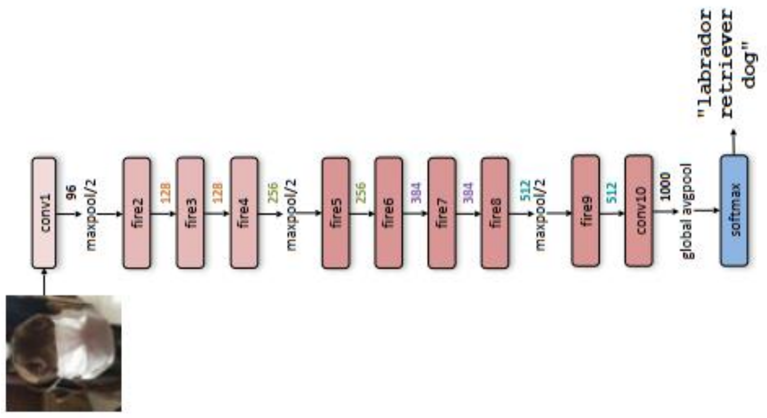


Figure 2.12: Macro architectural view of SqueezeNet architecture (from [16]).

Figure 2.13 presents the Top-1 accuracy of some of the most popular CNNs, which consists to determine if the class with the highest probability in the model matches the label of the image.

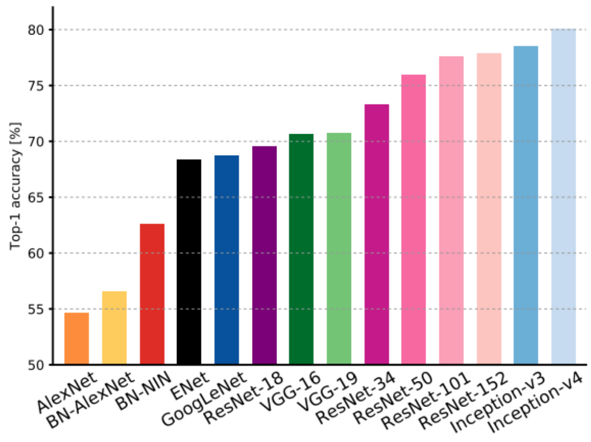


Figure 2.13: Top-1 accuracy of some of the most popular CNN models (from [15]).

2.5. MobileNets

MobileNets [3] is based on depthwise separable convolutions which factorize a standard convolution (figure 2.14) into a depthwise convolution and a pointwise convolution. The depthwise convolution applies a single $k \times k$ kernel to each input channel (figure 2.15 a)) and the following pointwise convolution applies a $1 \times 1 \times I$ filter to linearly combine the output of the depthwise convolution (figure 2.15 b)).

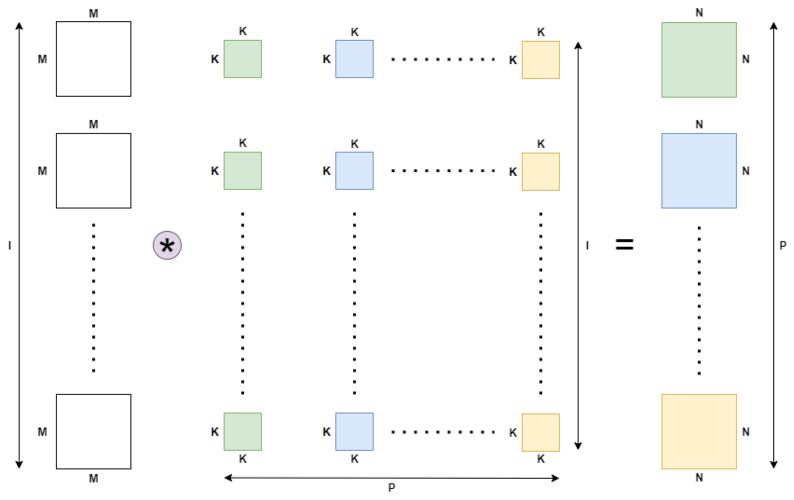


Figure 2.14: Standard convolution.

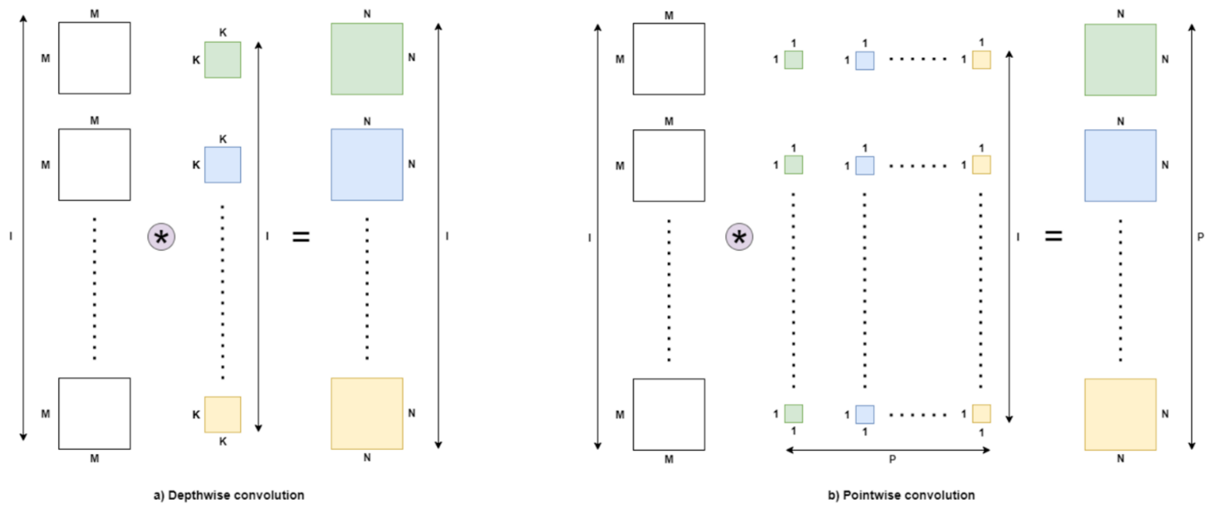


Figure 2.15: Depthwise separable convolution.

The complete model of MobileNets is composed of 30 layers. To avoid losing too much information at the beginning, the first layer is a standard convolution, which receives the input image. Then, the next 26 layers are depthwise and pointwise convolutions, arranged in an interleaved way, which perform the feature maps. The last 3 layers are a pooling layer, to select the average value of the previous feature map; an FC layer that classifies the input image according to the available labels; and a SoftMax function to convert the values received from the FC layer to probabilities. All the layers are followed by Batch Norm and ReLU, except for the last 3 layers. Thus, from here, the set of one layer followed by Batch Norm and ReLU will be referred to a stage. The next sub-sections of this chapter analyses in detail the complete MobileNets architecture.

2.5.1. Standard convolutional layer (3-Dimension convolution)

In the MobileNets model, this layer receives an input image size of 224×224 pixels in RGB format. RGB colour format uses 8 bits (for each colour) to represent the image pixels, therefore the values of these pixels are integers in the interval $P \in [0, 255]$, which are then normalized between -1 and 1 before computing the standard convolution. To perform the convolution, first, a half zero padding is applied, i.e., adding an extra row and column to each sub-image that composes the RGB image and fill them with zeros. This produces an RGB input image of 225×225 which after convolving with a $3 \times 3 \times 3$ filter and stride 2, creates an output image of 112×112 pixels. This result can be obtained, setting the values on equation (2.6):

$$O = \frac{(225 - 3) + 2 \times 0}{2} + 1 = 112 \tag{2.7}$$

Thus, the convolution of the RGB image with the 32 filters of size $3 \times 3 \times 3$ and stride 2, results in 32 OFMs of size 112×112 pixels, as shown in figure 2.16.

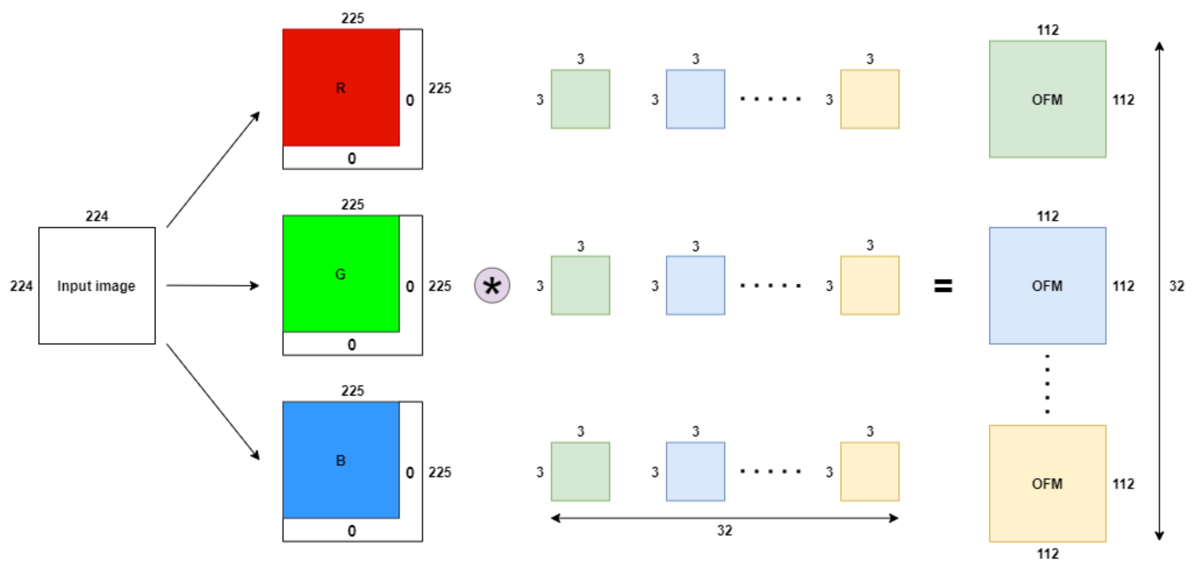


Figure 2.16: Standard convolution of MobileNets first layer

Considering K as the height/length size of the kernel, I the number of IFMs, N as the height/length size of the OFMs and P as the number of filters, the total number of multiplications and accumulations (MAC) of a standard layer is given by (2.8):

$$K \cdot K \cdot I \cdot N \cdot N \cdot P \tag{2.8}$$

Therefore, the total number of operations on the first layer of MobileNets model is:

$$Total\ standard\ layer\ MAC\ operations\ =\ 3 \times 3 \times 3 \times 112 \times 112 \times 32 = 10{,}838{,}016 \tag{2.9}$$

2.5.2. Depthwise convolutional layer (2-Dimension convolution)

This layer receives the IMFs and performs a 2D convolution. The size of the kernels in these layers is always $3\ \times\ 3$, however, the number of kernels increases from 32 (first depthwise layer) to 1024 (last depthwise layer). Some layers use stride 1, and other layers use stride 2. When stride is set to 2, a half zero padding is applied which results in an OFM with half the size of the IFM, as shown in figure 2.17.

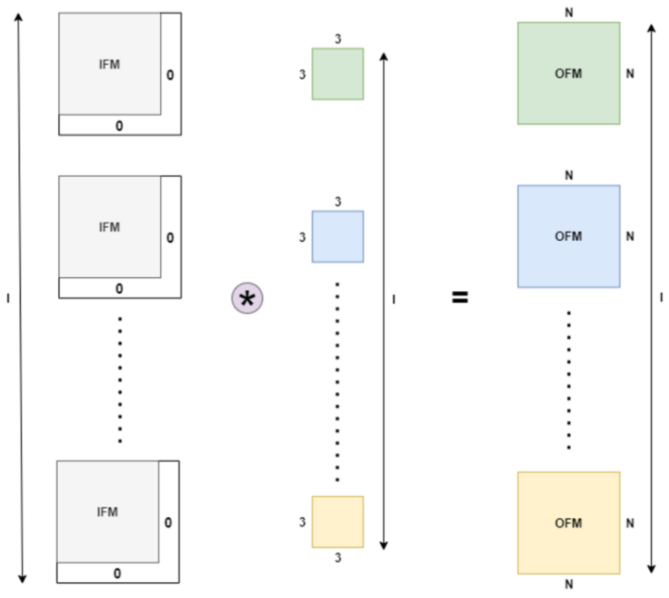


Figure 2.17: MobileNets depthwise convolution with stride 2.

When stride is 1, a complete zero padding is applied, which leads to an IFM of size $(\text{Input} + 2) \times (\text{Input} + 2)$ pixels, and results in an OFM with the same size of the IFM (figure 2.18).

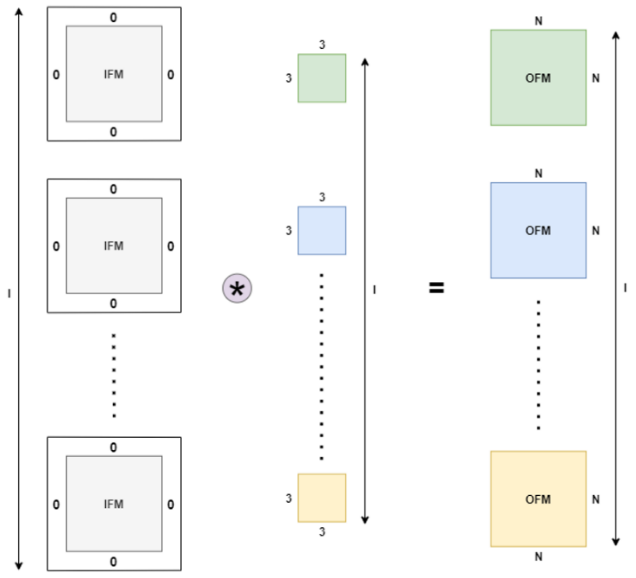


Figure 2.18: MobileNets depthwise convolution with stride 1.

Again, considering K as the height/length size of the kernel, N as the height/length size of the OFMs and I as the number of kernels, the total number of MAC operations of a depthwise layer is given by equation (2.10):

$$K \cdot K \cdot N \cdot N \cdot I \tag{2.10}$$

and, for the particular situation of MobileNets, the total number of operations is (2.11):

$$Total\ depthwise\ MAC\ operations\ =\ 3 \times 3 \times N \times N \times I \tag{2.11}$$

2.5.3. Pointwise convolutional layer (1-Dimension convolution)

The pointwise layer performs a depth convolution over all the OFMs of the depthwise layer, using a $1 \times 1 \times I$ filter size. Since all the kernels are composed of only one element, the size of the OFMs of this layer is always the same size of the IFMs (figure 2.19).

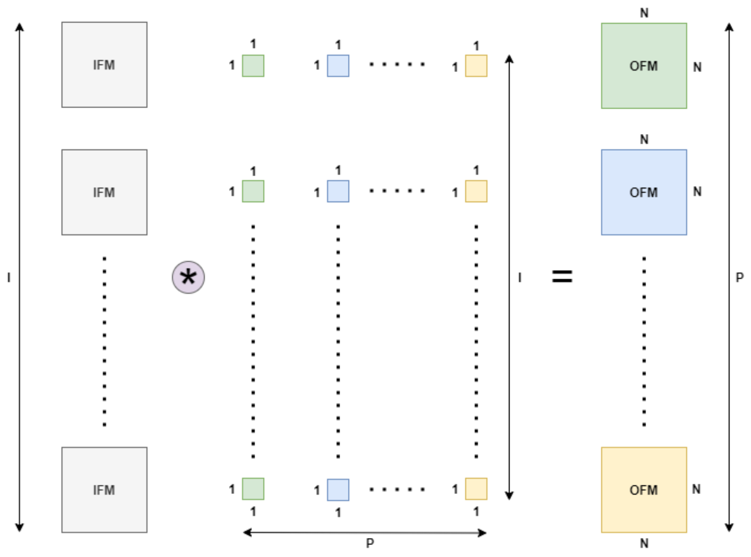


Figure 2.19: MobileNets pointwise convolution.

Considering K as the height/length size of the kernel, N as the height/length size of the OFMs, I as the number of IFMs, and P as the number of filters, the total number of MAC operations of a pointwise convolution is given by (2.12):

$$K \cdot K \cdot I \cdot N \cdot N \cdot P \tag{2.12}$$

From figure 2.19 is possible to conclude that the total number of MAC operations for each pointwise layer is:

$$Total\ pointwise\ MAC\ operations\ =\ 1 \times 1 \times I \times N \times N \times\ P = \ I \times N \times N \times\ P \tag{2.13}$$

The decomposition of a standard layer into depthwise and pointwise convolutions makes MobileNets a lighter model compared to traditional CNN models. The number of operations of a depthwise separable convolution is obtained combining the number of operations of a depthwise and a pointwise convolution:

$$K \cdot K \cdot N \cdot N \cdot I + N \cdot N \cdot I \cdot P \tag{2.14}$$

Observing figure 2.14, the number of operations for a standard convolution is:

$$K \cdot K \cdot I \cdot N \cdot N \cdot P \tag{2.15}$$

The ratio between the number of operations of a standard convolution and a depthwise separable convolution, equations (2.15) and (2.14) is:

$$\frac{K \cdot K \cdot I \cdot N \cdot N \cdot P}{K \cdot K \cdot N \cdot N \cdot I + N \cdot N \cdot I \cdot P} = \frac{K \cdot K \cdot P}{K \cdot K + P} \tag{2.16}$$

Considering $P \gg K$, equation (2.16) can be approximated:

$$\frac{K \cdot K \cdot P}{K \cdot K + P} \approx \frac{K \cdot K \cdot P}{P} \approx K \cdot K = K^2 \tag{2.17}$$

For kernels of size $K = 3$, the depthwise/pointwise decomposition uses approximately 9 times fewer operations to process the same layer than a standard CNN model.

2.5.4. Batch Norm and ReLU

Batch Norm and ReLU are performed after each of the described layers before. The first one is given by algorithm 1, which normalizes the input layer by adjusting and scaling the activations.

Algorithm 1 - Batch Norm transform (adapted from [12]).

```
Input: Values of $x$ over a mini-batch: $B = \{x1 \dots m\}$;
        Parameters to be learned: $\gamma, \beta$
Output: $\{y_i = BN_{\gamma,\beta}(x_i)\}$

$\mu_\beta \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                                  // mini-batch mean
$\sigma^2{}_\beta \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\beta)^2$                 // mini-batch variance
$\widehat{x_i} \leftarrow \frac{x_i - \mu_\beta}{\sqrt{\sigma^2{}_\beta + \epsilon}}$                        // normalize
$y_i \leftarrow \gamma\widehat{x_i} + \beta \equiv BN_{\gamma,\beta}(x_i)$                  // scale and shift
```

The values of variables μ_β, $\sigma^2{}_\beta$, γ and β are parameters from the MobileNets model, and for ϵ, the value used in this work is 0.001, the same as in Tensorflow [17]. Algorithm 1 can be rearranged and written as (2.18):

$$y_i \leftarrow \gamma \frac{x_i - \mu_\beta}{\sqrt{\sigma^2{}_\beta + \epsilon}} + \beta = \gamma(x_i - \mu_\beta)\frac{1}{\sqrt{\sigma^2{}_\beta + \epsilon}} + \beta = \gamma(x_i - \mu_\beta)V + \beta \tag{2.18}$$

Considering N as the height/length size of the OFM and P as the number of OFMs, the total Batch Norm operations can be calculated by equation (2.19):

$$Total\ Batch\ Norm\ operations\ = N \times N \times P \times 4 \tag{2.19}$$

The second function, ReLU6 is the activation function that preserves the values between 0 and 6, and assign 0 to all negative values and 6 to all values bigger than 6 (see algorithm 2).

Algorithm 2 – ReLU6 activation function.

```
Input: Values of $x \in \mathbb{R}$
Output: $y = [0, 6]$

$\boldsymbol{if}\ x < 0\ \boldsymbol{then}$
  $y \leftarrow 0$
$\boldsymbol{else\ if}\ x > 6\ \boldsymbol{then}$
  $y \leftarrow 6$
    $\boldsymbol{else}$
      $y \leftarrow x$
$\boldsymbol{end\ if}$
```

Since the number of operations (comparisons) of this function depends on the input value, the total number of operations in the worst case (the process performs two comparisons) is given by (2.20):

$$Total\ ReLU6\ operations\ = N \times N \times P \times 2 \tag{2.20}$$

where N is the height/length size of the OFM and P the number of OFMs.

2.5.5. Global average pooling layer

The last pointwise stage of MobileNets outputs 1,024 OFMs of size 7×7 and is followed by a global average pooling. This pooling method consists of summing all the values of each of the 7×7 OFMs and then perform a division by the matrix size $(7 \times 7 = 49)$ as presented in equation (2.21):

$$Pooling = \sum_{0}^{i=7} \sum_{0}^{j=7} X_{i,j} \times \frac{1}{N} \tag{2.21}$$

where N is the matrix size. From (2.21) the total number of operations (multiplications + sums + divisions) in the pooling layer is (2.22):

$$Total\ Pooling\ operations\ = \big((7 \times 7) + 1\big) \times\ OFMs = \ 50 \times 1{,}024 = 51{,}200 \tag{2.22}$$

2.5.6. Fully connected layer

The FC layer receives 1,024 input values from the global average pooling layer, and the output values are equal to the number of classes for which the model was trained. In the MobileNets model used in this work, there are 1,000 classes, which means that the model is trained to identify 1,000 different image categories. Therefore, the FC layer has 1,000 outputs.

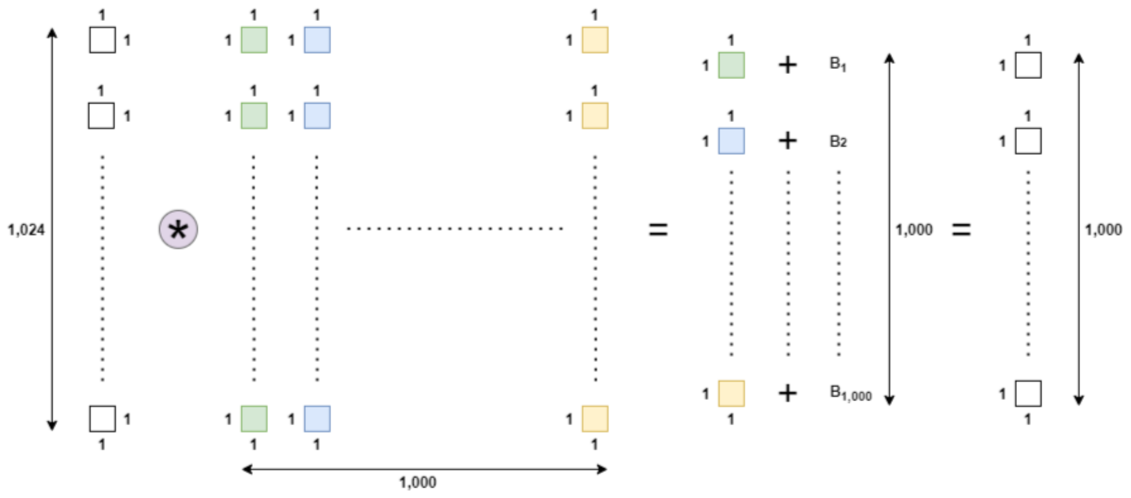


Figure 2.20: MobileNets Fully Connected layer. B_i represents the bias values where $i \in [1, 1{,}000]$.

This way, the total number of MAC operations in this layer is (2.23):

$$Total\ \ fully\ connected\ operations\ = (1{,}024 \times 1{,}000) \times 2 + 1{,}000 = 2{,}049{,}000 \qquad (2.23)$$

2.5.7. SoftMax layer

The SoftMax [18] function is used when there are more than two classes to classify and assigns to each result a probability so that the sum of all values is equal to 1. This function is defined as a generalization of the logistic function and is given by (2.24):

$$SoftMax = \sigma(\boldsymbol{Z})_j = \frac{e^{Z_j}}{\sum_{k=1}^{K} e^{Z_k}} \qquad (2.24)$$

where $\boldsymbol{Z}$ is a K-dimensional vector of arbitrary real values and $\sigma(\boldsymbol{Z})$ a K-dimensional vector of real values.

2.5.8. Smaller MobileNets models

There is the possibility to optimize MobileNets using two hyperparameters in order to create smaller and faster models. The first parameter is the width multiplier α, used to adjust the network uniformly at each layer, reducing the number of input and output channels. With this, the number of input channels I and output channels P, becomes respectively, αI and αP. Applying α to equation (2.14) the number of operations is given by (2.25):

$$K \cdot K \cdot \alpha I \cdot N \cdot N + \alpha I \cdot \alpha P \cdot N \cdot N \tag{2.25}$$

where $\alpha \in]0, 1]$, being the typical values used 1, 0.75, 0.5 and 0.25. The standard MobileNets model is represented by $\alpha = 1$ while the reduced MobileNets models by $\alpha < 1$. The application of the width multiplier to the model reduces the computation cost and the number of parameters approximately by α^2. This means that the width multiplier can be used to define smaller models with acceptable accuracy, latency and size trade-off. Table 2.1 presents some examples of the application of the width multiplier on MobileNets model.

Table 2.1: MobileNets width multiplier (adapted from [3]).

Width multiplier	ImageNet accuracy	Million mult-adds	Million parameters
1.00 MobileNets-224	70.6%	569	4.2
0.75 MobileNets-224	68.4%	325	2.6
0.50 MobileNets-224	63.7%	149	1.3
0.25 MobileNets-224	50.6%	41	0.5

In the Width Multiplier column, four MobileNets models are presented, with the value for the α on the left, and the input resolution on the right. The main objective of this procedure is the reduction of the parameters to obtain smaller models, as presented in the last column of table 2.1. As shown, this parameter reduction may significantly impact the accuracy of the model.

The other hyperparameter is β, which is defined as a resolution multiplier, is applied to reduce the resolution of the input image and therefore to reduce the computational cost of the neural network. Applying β to equation (2.25), the number of operations is reduced to equation (2.26):

$$K \cdot K \cdot \alpha I \cdot \beta N \cdot \beta N + \alpha I \cdot \alpha P \cdot \beta N \cdot \beta N \tag{2.26}$$

where $\alpha, \beta \in]0, 1]$. The values typically used for β are set implicitly to reflect an input resolution of the network of 224, 192, 160 or 128. The standard MobileNets model is represented by $\beta = 1$ while the reduced computation MobileNets by $\beta < 1$. The application of the resolution multiplier to the model reduces the computation cost by β^2. Table 2.2 presents some examples of the application of the resolution multiplier on MobileNets model. In the Resolution Multiplier column, the same MobileNets model is used ($\alpha = 1.00$), only the input resolution changes. Since the model is always the same, the number of parameters remains constant. The variation of the input resolution only affects the number of operations required, while the model parameters remain the same. However, using a lower input resolution reduces the model accuracy.

Table 2.2: *MobileNets resolution multiplier (adapted from [3]).*

Resolution multiplier	ImageNet accuracy	Million mult-adds	Million parameters
1.00 MobileNets-224	70.6%	569	4.2
1.00 MobileNets-192	69.1%	418	4.2
1.00 MobileNets-160	67.2%	290	4.2
1.00 MobileNets-128	64.4%	186	4.2

2.5.9. CNN models comparison

Table 3 compares some popular CNN models, including two MobileNets models, a standard and a reduced model with width multiplier $\alpha = 0.5$ and resolution multiplier $\beta = 0.714$ (input image resolution of 160×160). As shown, the standard MobileNets model has higher accuracy than GoogleNet which has about 2.6 million more parameters than MobileNets, and less 0.9% accuracy than the VGG 16 which has (more than) 32x more parameters. The reduced MobileNets model achieves a reduction in more than half of the parameters and using a lower input resolution of 160×160 produces an accuracy higher than the Squeezenet model which has about 100x more operations. Moreover, the reduced MobileNets model obtains an accuracy higher than the AlexNet model with 45x fewer parameters.

Table 2.3: MobileNets comparison to popular models (adapted from [3]).

Model	ImageNet accuracy	Million mult-adds	Million parameters
1.0 MobileNet-224	70.6%	569	4.20
GoogleNet	69.8%	1,550	6.80
VGG 16	71.5%	15,300	138.00
0.50 MobileNet-160	60.2%	76	1.32
Squeezenet	57.5%	1,700	1.25
AlexNet	57.2%	720	60.00

2.6. Conclusion

In this chapter, the concept of CNN was introduced as a variant of ANNs, suited and trained to evaluate a specific type of data, for instance, the classification of images. Some popular CNNs were introduced to understand how these neural networks work, particularly the MobileNets model that will be used in this work. Analyzing in detail the MobileNets model in this chapter allows understanding the architecture of the network, computational effort, memory requirements and benefits comparing to others. The strategy to reduce the MobileNets model to achieve smaller models and fewer will not be used due to accuracy loss. Thus, the focus of this work will be on the standard MobileNets model.

Chapter 3

CNN models on FPGAs

This chapter presents a set of model optimization and hardware design techniques that have been used to accelerate CNN models on an FPGA. CNN models can be compressed in order to reduce the use of memory, communications and operations while minimizing accuracy loss. From the hardware perspective, specific architecture modules are designed to reuse data, accelerate convolution operations, and efficiently use all available resources. Also, when porting a CNN model to an FPGA device, the bit widths of operators and weights are often reduced.

3.1. CNN model optimization

As referred, CNNs require a large number of parameters that consume considerable storage and memory bandwidth, making it impossible to fit them all in on-chip memory. One way to reduce the CNNs parameters is to apply a pruning strategy [19] [20] [21]. A simple technique of pruning starts by learning the connectivity during training. Thereafter, the connections with weights below a threshold are removed, and the network is retrained to learn the final weights for the remaining sparse connections (figure 3.1). Applying this pruning method to AlexNet and VGG-16 model, work [19] was able to reduce the number of parameters in 9x and 13x, respectively.

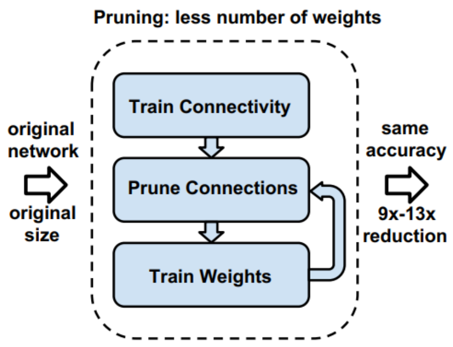


Figure 3.1: Network pruning method applied to AlexNet and VGG-16 (adapted from [19]).

The pruning proposed on [20] consists of successive iterations to identify and remove the least essential parameters and re-tune the network. The iterations stop when a target trade-off between accuracy and pruning objective is achieved (figure 3.2).

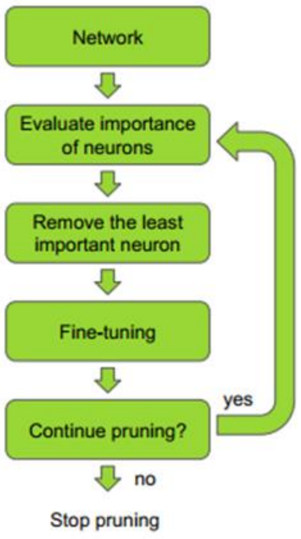

Network
Evaluate importance of neurons
Remove the least important neuron
Fine-tuning
Continue pruning?
yes
no
Stop pruning

Figure 3.2: Network pruning as a backward filter (from [20]).

Three types of pruning are proposed in [21]: *channel-wise*, which prunes all the incoming and outgoing kernels if a feature map in a layer is removed (red connections on figure 3.3 a)); *kernel wise*, which deletes kernels where each kernel corresponds to one full convolution (blue connections on figure 3.3 a)); and *intra-kernel sparsity*, which forces some of the kernel weights into zero-valued zones (figure 3.3 b)).

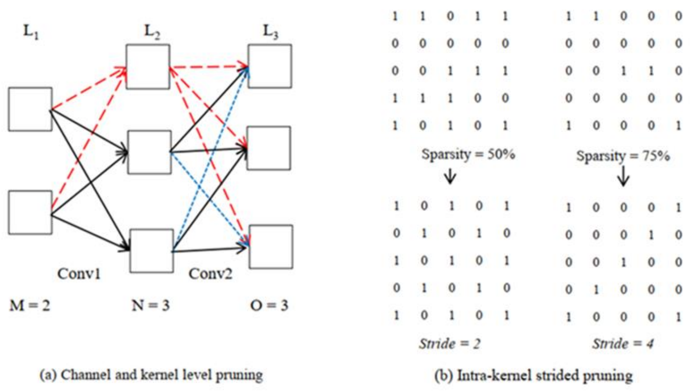


Figure 3.3: Three types of pruning (from [21]).

The application of a correct pruning strategy reduces the model and keeps the same accuracy level, which in practice represents a smaller and faster version of the model with the same classification rigour.

Another strategy to optimize CNN models is quantization [22] [23] [24]. Data quantization defines how data values are represented and how many bits are used. The use of short fixed-point numbers is sufficient for CNN inference, but not for training. CNN models can be quantized using one of the two strategies described in [22]. The first strategy is post-training quantization, which simply converts the 32 bits floating-point parameters of the model usually to 16 or 8 bits fixed-point because these are the minimum values that can keep an acceptable accuracy of the model. The other is quantized aware training, where the model is normalized and converted from 32 bits floating-point to 8 bits fixed-point and then retrained. In this situation, since the network is retrained, the use of 8 bits is usually sufficient to obtain an accuracy near to the original, or in some cases, even better. Some tests conducted on a trained CNN [23] are presented in table 3.1 which show that if a 16-bit fixed-point is used only for inference, the impact on error is not significant, but if it is also used for training, then there is no convergence.

Table 3.1: Impact of fixed-point computations on error (from [23]).

Inference	Training	Error
Floating-point	Floating-point	0.82%
Fixed-point (16 bits)	Floating-point	0.83%
Fixed-point (32 bits)	Floating-point	0.83%
Fixed-point (16 bits)	Fixed-point (16 bits)	(no convergence)
Fixed-point (16 bits)	Fixed-point (32 bits)	0.91%

Work [24] proposes a dynamic-quantization strategy and an automatic workflow. The objective is to convert floating-point numbers into fixed-point ones, with the highest possible accuracy. The difference between the widely used static-precision quantization strategies and the proposed dynamic-quantization strategy is that the fractional length (f_l) is dynamic for different layers and feature map sets, and static in one layer, to minimize the truncation error of each layer. To obtain the final quantization value, the process is divided into two phases: weight quantization and data quantization. The weight quantization phase aims to find the optimal f_l for weights in one layer and can be expressed by equation (3.1):

$$f_l = argmin \sum \left| W_{float} - W(bw, f_l) \right| \tag{3.1}$$

where W is a weight and $W(bw, f_l)$ the fixed-point format of W, given a bw and f_l. The data quantization phase aims to find the optimal f_l for a set of feature maps between two layers and can be expressed as (3.2):

$$f_l = argmin \sum \left| x^+_{float} - x^+(bw, f_l) \right| \tag{3.2}$$

where x^+ is the result of a layer, considering the computation of a layer as $x^+ = A \cdot x$. To analyze this quantization strategy in the inference process, some tests were performed under Caffe deep learning framework, and three networks were evaluated: CaffeNet, VGG16 and VGG16-SVD (table 3.2). Observing table 3.2, the first columns of each network (Exp 1, Exp 4 and Exp 7) are the standard CNN models which use the 32-bit floating-point representation, and the others are the respective quantization experiments. The results indicate that, in some cases, dynamic quantization using a smaller number of bits can be applied without significant accuracy loss (e.g Exp3 use 8-bit weights and activations with an accuracy loss lower than 0.9%).

Table 3.2: Exploration of different data quantization strategies with state-of-the-art CNNs (adapted from [24]).

Network	CaffeNet			VGG16			VGG16-SVD		
Experiment	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Exp 6	Exp 7	Exp 8	Exp 9
Activation Bits	32	16	8	32	16	8	32	16	8
Weight Bits	32	16	8	32	16	8 or 4	32	16	8 or 4
Activation Quantization	-	Dynamic	Dynamic	-	2^{-2}	Dynamic	-	Dynamic	Dynamic
Weight Quantization	-	Dynamic	Dynamic	-	2^{-15}	Dynamic	-	Dynamic	Dynamic
Top 1 Accuracy	53.9%	53.9%	53.0%	68.1%	68.0%	67.0%	68.0%	64.6%	64.1%
Top 5 Accuracy	77.7%	77.1%	76.6%	88.0%	87.9%	87.6%	88.0%	87.0%	86.3%

1 The weight bits "8 or 4" in Exp6 and Exp9 means 8 bits for CONV layers and 4 bits for FC layers.

For the particular case of MobileNets, this model is commonly quantized using 8-bit fixed-point and then retrained [25] [26]. Three contributions to improve the latency-vs-accuracy trade-off of MobileNets on common mobile hardware are presented in [25]. The first one is to provide a quantization scheme adopted from Tensorflow Lite [27], which quantizes the weights and activations to 8-bit integers and the biases to 32-bit integers. Another contribution of [25] is a framework to implement efficiently quantized inference on integer-arithmetic-only hardware. The last contribution of [25], is a training framework to retrain the quantized network, and therefore minimize the loss of accuracy caused by the quantization. To evaluate the efficiency of the strategy, the quantized MobileNets model was evaluated with the ImageNet validation dataset, using two Qualcomm [28] Snapdragon cores. These Snapdragon cores are suited for SoC semiconductor products with processors based on Advanced RISC Machine (ARM) [29] architecture, used on mobile devices. As shown in the following two figures, integer-only quantized MobileNets achieve higher accuracy than the floating-point implementation, for the same execution time. Using the Snapdragon 835 LITTLE, a power-efficient core, the accuracy gap is about 10% at 33 ms latency needed for real-time (30 fps) operation (figure 3.4).

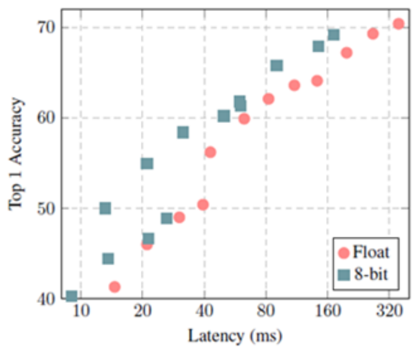


Figure 3.4: Latency-vs-accuracy trade-off of float vs. integer-only MobileNets on ImageNet using Snapdragon 835 LITTLE cores (from [25]).

Using the Snapdragon 821 big core (high-performance core) the quantized models do not show any significant latency advantage, since floating-point computation is better optimized in this core (figure 3.5).

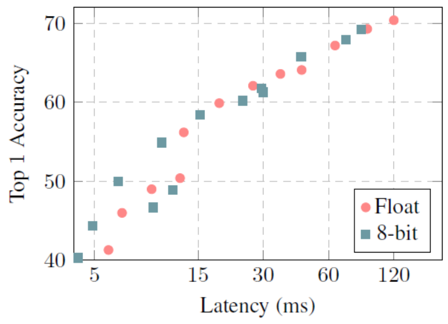
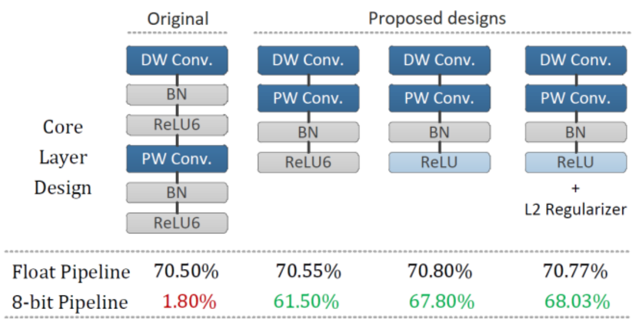


Figure 3.5: Latency-vs-accuracy tradeoff of float vs. integer-only MobileNets on ImageNet using Snapdragon 821 big core (from [25]).

The rearrange of MobileNets architecture is proposed in [26] to become more quantization friendly, namely the depthwise separable convolution layers, which the authors state are the causes of large quantization loss. Figure 3.6 provides an overview of the progress in developing this new architecture. First, the standard MobileNets model is quantized using 8-bit fixed-point, which practically results in an inaccurate model, since the Top 1 accuracy is only 1.8%. After performing a quantization loss analysis on MobileNets model, Batch Norm and ReLU6 were identified as the causes of this issue. More specifically, the problem arises because the depthwise convolution is performed independently in each channel, but, the minimum and maximum values used to quantize the weights parameters are taken collectively from all channels. Since the depthwise separable convolution splits a standard convolution into a depthwise layer for filtering and a pointwise layer for combining, the Batch Norm and ReLU6 between the depthwise and pointwise layers were removed.



	Original	Proposed designs		
Float Pipeline	70.50%	70.55%	70.80%	70.77%
8-bit Pipeline	1.80%	61.50%	67.80%	68.03%

Figure 3.6: Top-1 accuracy with different core layer designs on ImageNet2012 validation dataset (from [26]).

The model trained without both the Batch Norm and the ReLU6 between the depthwise and pointwise convolution achieves a 70.55% Top-1 accuracy for the floating-point implementation (practically the same as the original model). However, the accuracy of the 8-bit quantized model improved from 1.8% to 61.5%. Different activation functions were tested in all pointwise layers, and the conclusion was that

ReLU has a better performance than ReLU6. The improvements from replacing ReLU6 by ReLU in every pointwise layer are shown in the second proposed design of figure 3.6. Finally, to avoid increasing the quantization range, the L2 regularization method was used to distribute the weights parameters better, leading to an additional 6.5% improvement on the Top-1 accuracy of the quantized model (third proposed design of figure 3.6).

The last work [30] combines pruning and quantization during the training of MobileNets. First, the quantization training process described in algorithm 3 is conducted, followed by an iterative pruning and retraining process.

Algorithm 3 Quantization Training Process for A L-layer neural network

Require: Inputs a_0, labels a^*, kernel parameters W, batch normalization parameters θ, maximum iteration number *MaxIter*, lower bound value min, upper bound value max.
Ensure: W and θ at *MaxIter* iteration.

```
for $iter = 1$ to $MaxIter$ do
    // Forward Propagation
    $a_0^Q \leftarrow Quantize(a_0)$
    for $l = 1$ to $L$ do
        $W_l^Q \leftarrow Quantize(W_l)$
        $a_l \leftarrow layer\_forward(a_{l-1}^Q, W_l^Q)$;
        $a_l^Q \leftarrow Quantize(a_l)$;
    end for

    // Backward Propagation
    for $l = L$ to $1$ do
        $g_{a_{l-1}}, g_{W_l^Q} \leftarrow layer\_backward(g_{a_l}, W_l^Q)$
        $\theta_l \leftarrow Update(\theta_l, g_{\theta_l})$
        $W_l \leftarrow Clip(Update(W_l, g_{W_l^Q}), min, max)$
    end for
end for
```

In each iteration, $Prune(*)$ reduces the number of filters resulting in fewer OFMs and in a smaller memory requirement. However, in this strategy, the filters are removed or kept as a whole, according to the summation of its values, instead of making them sparse. Although only less important filters are removed, some information is lost, so the model is retrained to compensate for the accuracy loss. During the execution of algorithm 3, in forward propagation, weights W and activations a are quantized before actual computations during inference. Then the $Quantize(*)$ function converts real values to a pre-defined fixed-point representation. The inference computation is conducted by the $layer\_forward\ (*)$ function. In backward propagation, the updating is applied to the real-valued weights W rather than their quantized alternatives W^Q, which keeps a higher precision during training.

3.2. Hardware design

Many aspects must be considered when implementing a trained CNN model on an FPGA. Since CNN involve a lot of computation and parameters, the computational effort and memory accesses must be minimized. Also, all the available resources of the FPGA should be efficiently used to maximize the throughput. This section presents some efficient techniques that have been proposed to implement CNN models on FPGAs.

3.2.1. Parallelism

CNN computation can take advantage of different parallelization methods to accelerate the inference process. However, a standard CNN is composed of multi-layers with data dependencies between them, which preclude the execution of all the layers simultaneously. This restricts the use of CNN parallelization to each layer at a time, but even in this situation, there are some impediments. Hardware resources and other limitations such as memory bandwidth also determine the maximum computation that a specific CNN layer can perform. The design of CNN architectures mainly explores three types of parallelism [31]:

1. Operator-level (fine-grained): In the convolution of a single input image of size $m \times m$ with a kernel of size $k \times k$, each output pixel requires $k \times k$ MACs operations, which can be executed in parallel.
2. Intra-output (Coarse-grained): The computation of each pixel in a single OFM can be done in parallel since it is the sum of independent input-kernel convolutions.
3. Inter-output (Coarse-grained): Multiple OFMs can be computed in parallel by multiple processing elements (PEs).

The following works of CNN implementations on FPGA described in the next sections explore these three parallelism types.

3.2.2. System architecture

Commonly, FPGA implementations have on-chip memory and off-chip memory. All information can only be stored on the FPGA before computation begins if the CNN model is small enough to fit on the available on-chip memory [30]. Thus, the most common CNN accelerators read or write information on the external memory during computation and use on-chip buffers to save intermediate results [24] [32] [33]. Regarding computation, most common CNN models only perform standard and FC convolutions [24] [33]. However, with the rise of new models such as MobileNets, new implementations need to use a more elaborated type of PE to process different more types of convolutions [30] [32].

Figure 3.7 presents a complete CPU + FPGA heterogeneous architecture to accelerate CNNs using a Zynq device [24]: The computing module consists of PEs units, implemented on the Programmable Logic (PL), which perform most of the computation tasks, including convolutional layers, pooling layers and FC layers. Input and output buffers receive data from external memory to be used by the PEs and save the results, respectively. The Processing System (PS) includes the general-purpose processor and the external memory controller.

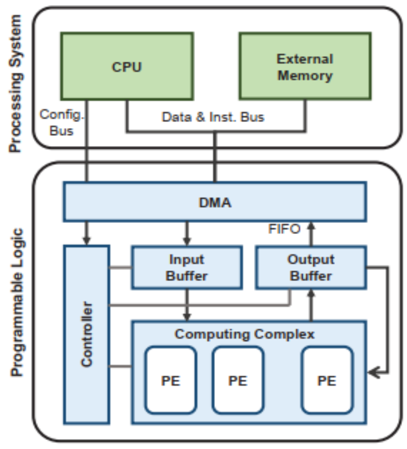


Figure 3.7: CPU + FPGA heterogenous architecture (from [24]).

The inference process begins with data preparation, where all the CNN model parameters, image data and control data are stored in the external memory. Control data includes the buffer descriptors (BD) used by the Direct Memory Acess (DMA) and instructions used by the controller. Then, the CPU hosts start to configure the DMA with the BDs. After that, the DMA loads data to the buffers and instructions to the controller, and the computation begins. Figure 3.8 a) shows the PE architecture, which consists of a convolver, an adder tree, a non-linearity, a max-pooling and a bias and data shift module. The convolver module uses a classical line buffer design, presented in figure 3.8 b).

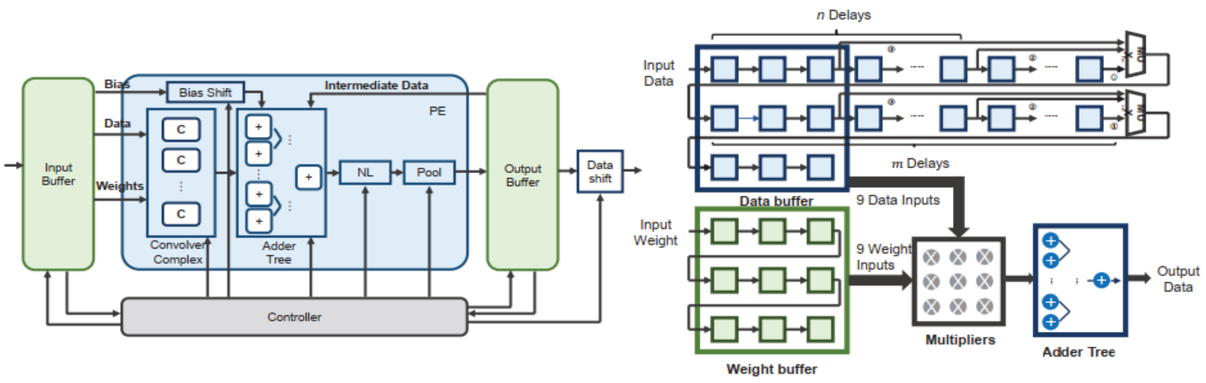


Figure 3.8: (a) the processing element; (b) the convolver in the processing element (from [24]).

The data buffer and weight buffer work like a FIFO design. While input data goes through the data buffer, the line buffer releases a window selection function on the input image. Then, multipliers and an adder tree will compute the convolution, obtaining one result per cycle. This module also performs the matrix-vector multiplications for FC layers by using a multiplexer at the end of the buffer lines to reduce the line buffers to the 3×3 window selection. The use of this strategy implies introducing a delay to allow the window selection to be fully loaded. For a 3×3 window selection, when input data goes through the buffer, a new vector is produced every 9 cycles. The bias shift aligns the input bias according to the quantization result of the layer. After processing the convolution, the data flows to the non-linearity module to apply an activation function to the data stream.

The max-pooling module applies a 2×2 window to the input data and selects the maximum of the four values. Lastly, the data shift module shifts the output data and cut back to the original width. The inference process ends when the processor applies the SoftMax function to the final results from PEs. This SoftMax function is processed on the CPU because an FPGA implementation would bring unnecessary overhead to the design, since this function is only called once, on the last layer. The use of various PEs modules in this work allows computing different convolutions simultaneously (inter-output parallelism). Also, operator-level parallelism is realized with 2-D convolvers and intra-output parallelism with multiple convolvers working simultaneously in each PE.

Another work [33] proposes a pipeline architecture, with different parallelism strategies for different layers, to accelerate the AlexNet CNN with 5 convolutional layers and 3 FC layers. The 8 layers are executed in 8 modules working as a pipeline. To minimize the workload because of the enormous quantity of input data and weights, they use two ping-pong buffers for each stage, to store the intermediate results, which allows one layer to write or read from one buffer while the next layer writes or reads data from the other buffer. Figure 3.9 presents the described architecture.

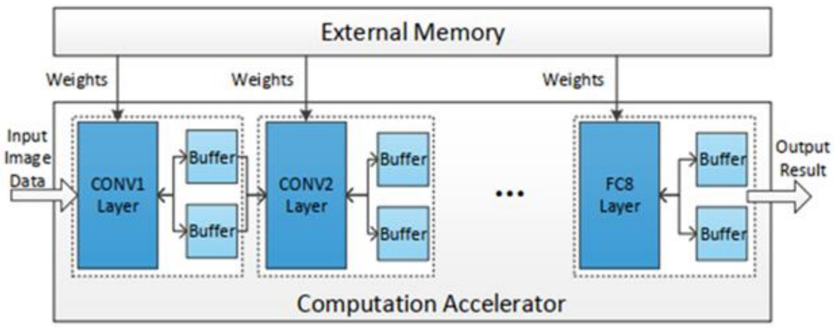


Figure 3.9: Pipelined architecture (from [33]).

The PE proposed in [33] is mainly composed of three accumulators, three multipliers and three shift registers, together with three multiplexers to choose the required weights. Figure 3.10 shows the 1-D PE applying a 3×3 kernel to a 5×5 IFM, producing a 3×3 OFM. The input data flows into the PE row by row and passes through three shift registers. When the first input x_1 reaches the rightmost multiplier, the multipliers start calculating the OFMs. After three clock cycles, the three multipliers generate the temporary results for y_1, y_2 and y_3 as shown by the red, blue and green colours. This convolution process is an example of intra-output parallelism since the three pixels are generated in parallel. The final results for y_1, y_2 and y_3 are generated when the calculating of the first three rows end. Since there are overlaps between the sliding windows, the rows need to be read three times (except the first and the last row) to produce all the OFM results. The pooling layer uses the same architecture as the convolution layer except for the operations, which are replaced by the max or average operations. Lastly, the FC layer is implemented as a matrix multiplication. For example, the first FC layer of the AlexNet model processes an input vector with 9,216 elements ($6 \times 6 \times 256$ feature maps flattened), which is multiplied by the $9{,}216 \times 4{,}096$ weights matrix, producing an output vector with 4,096 elements, as shown in figure 3.11 a). Due to limited hardware resources, this matrix multiplication is divided into several small-scale matrix multiplications.

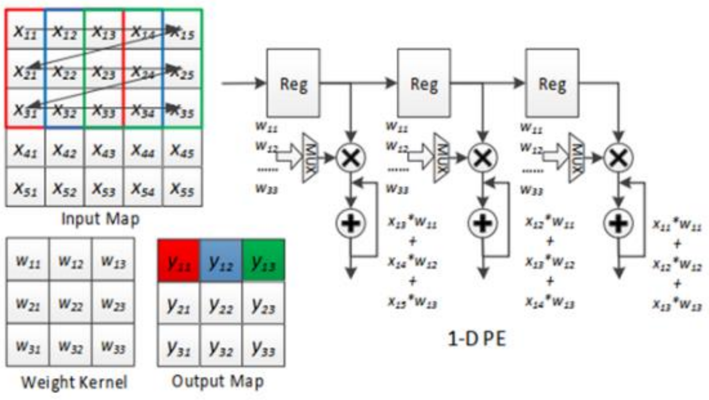


Figure 3.10: The 1-dimensional Processing Element (1-D PE) to implement (from [33]).

As shown in figure 3.11 b), the first small matrix are the values from x_1 to x_m which multiplies by $m \times n$ weights producing the temporary y_1 to y_n outputs. Then, the second small matrix are the values from x_{m+1} to x_{m2} which multiplies with a new $m \times n$ weights matrix updating the temporary y_1 to y_n outputs, and so on until the 9,216 data inputs and n rows, when the final y_1 to y_n are calculated. The other results are generated using the same procedure.

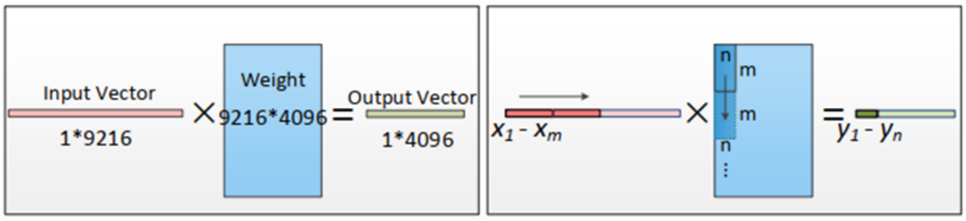


Figure 3.11: Matrix multiplication using the first FC layer of AlexNet model as an example (from [33]).

The following two works are implementations of the MobileNets model. In [30], all the network inputs are transferred from the external memory to on-chip buffers (figure 3.12).

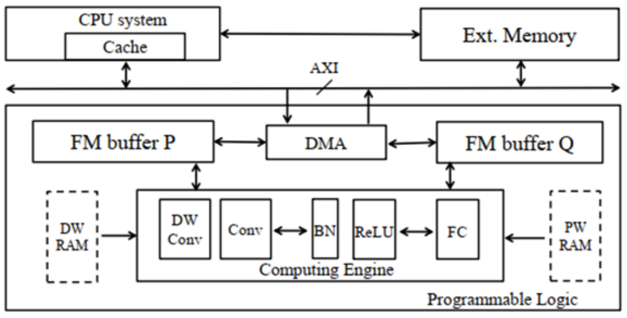


Figure 3.12: System Architecture Design for RR-MobileNets (from [30]).

The input images are stored in the FM buffer P and the network parameters in the DW RAM and the PW RAM. When all the initial data are stored in these buffers, the computation begins. Each layer is processed one at a time in the computation engine, which receives and stores the intermediate feature maps in the FM buffer P and the FM buffer Q, in an alternative manner for consecutive layers. The DW Conv module of the computation engine processes the depthwise convolutions, while the standard and pointwise convolutions are processed in the Conv module. BN and ReLU modules are common to these two convolution modules. Figure 3.13 a) presents the architecture of the DW Conv, BN and ReLU modules.

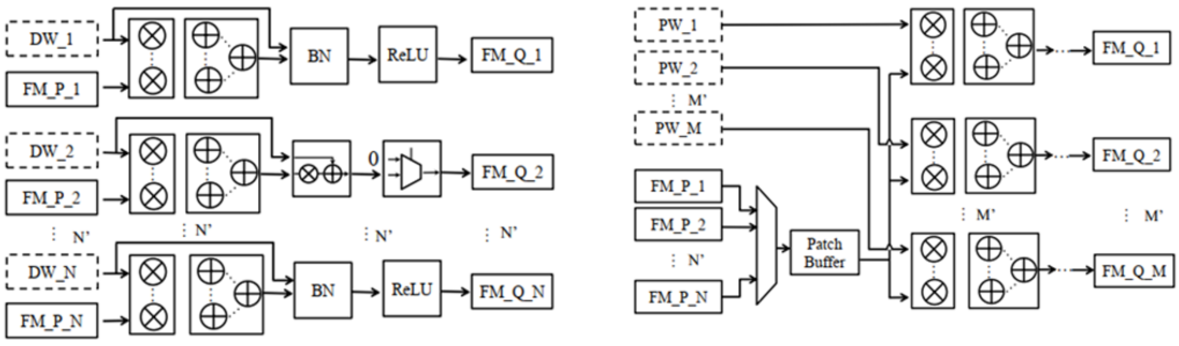


Figure 3.13: a) PE of DW module; b) PE of Conv module (from [30]).

The connection of these three modules forms a set of 32 PEs capable of processing 32 OFMs in parallel (inter-output parallelism). After processing the depthwise convolution, the BN module takes the outputs from the DW Conv and applies multiplication and addition for scaling operations. Then ReLU simply caps negative input values with zero. Similarly, the connection of the Conv module with BN and ReLU, also corresponds to a set of 32 PEs working in parallel (figure 3.13 b)). Lastly, the FC module is composed essentially of parallel multipliers and adder trees to execute de FC layer.

A variant of MobileNets [34] is used in work [32]. In this architecture, a matrix multiplication engine (MME) conducts all the CNN operations as presented in figure 3.14.

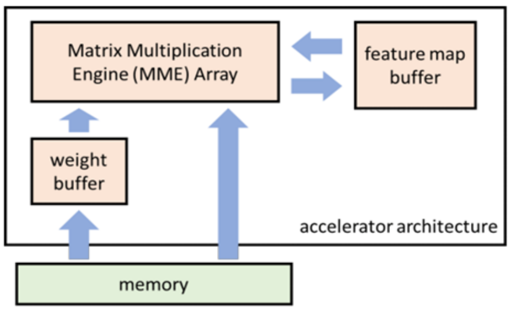


Figure 3.14: Block diagram of the accelerator system Module (from [32]).

All the input images and parameters are stored on the external memory. To avoid excessive latency, a ping-pong weight buffer is placed between the MME array and the external memory to maximize bandwidth. Biases are directly loaded to registers in MME array while the intermediate feature maps are stored on the feature map buffer. As shown in figure 3.15 a), to perform the computation of one convolution, first, the PE module (MME) loads the IFMs and the weights to line buffers. The Finite State Machine can adjust the length of the line buffer to work with different input sizes. Together with the weight buffers, these line buffers are connected to a multiplier array with 288 multipliers (32 slices with 9 multipliers) configurable to perform standard, depthwise or pointwise convolutions.

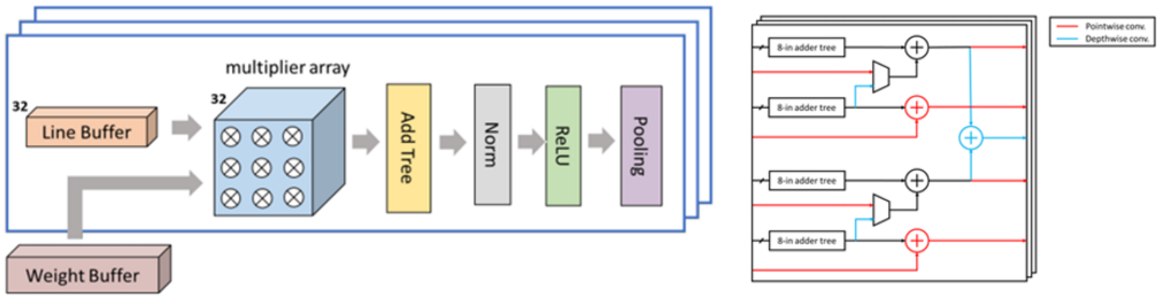


Figure 3.15: a) block diagram of an MME; b) block diagram of adder tree (from [32]).

To perform a standard convolution, the line buffer is configured to input 3 IFMs, and the multiplier array performs the multiplication of these IFMs with the respective weights. Only the results from the 3 slices from the multiplier array connected to the 3 line buffers will be considered in this situation. After the multiplication in multiplier array, the products are summed in an adder tree (figure 3.15 b)), configurable to perform depthwise or pointwise summing operation. When the depthwise convolution mode is selected, the adder tree sums up the products from each slice of the multiplier array in parallel. For one MME (figure 3.16 a)) the maximum output channels number is 32.

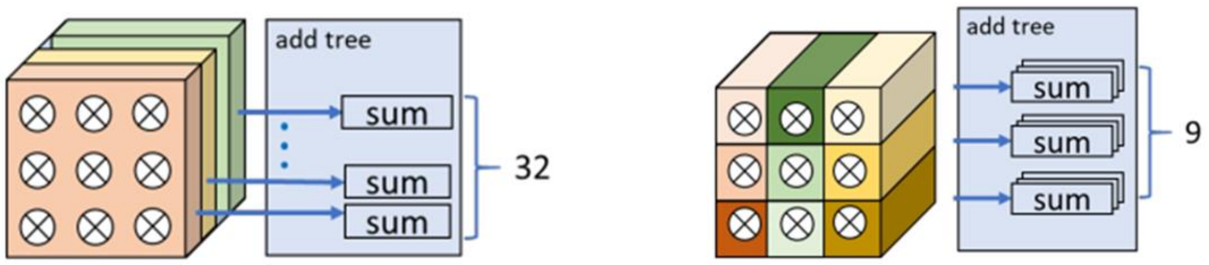


Figure 3.16: a) depthwise convolution in MME; b) pointwise convolution in MME (from [32]).

Pointwise convolution uses the divide and conquer algorithm in large matrix multiplication to divide the IFM into $M \times M \times 32$ submatrices, which are shifted into line buffers. This allows one PE to perform a maximum of $M \times M \times 32$ and 32×9 multiplications at once. The 32 products are then summed, and the output channels number is 9. The following steps perform the normalization, pooling and ReLU. ReLU has three selectable options: no ReLU, ReLU or ReLU6. Lastly, the pooling module can select between max and average pooling.

3.2.3. Memory management

The PE module accesses an external memory to get the necessary data to produce a result on FPGAs implementations. These communications between the PE and the external memory can considerably slow down computation if many communications are made. Therefore, it is desirable to minimize these communications or implement a system that does not stop PL computation while data is being loaded.

In [30], all the inputs are stored in external memory and loaded into the accelerator system by DMA through an Advanced eXtensible Interface (AXI) bus. When all the CNN layers are processed, the classification results are transferred back to the external memory. This is possible because, in this specific situation, all the intermediate results can be stored in the on-chip memory.

Other implementations access memory, depending on their needs [32] [33]. In [32], before starting a convolution, the needed parameters must be loaded to the accelerator. A ping-pong weight buffer is used to hide the latency caused by parameters loading (figure 3.17). When one of the buffers is outputting data for convolution, the other is loading data from the external memory and vice versa. A combined memory interface IP and DMA are used to connect the buffers in the CNN accelerator and the external memory.

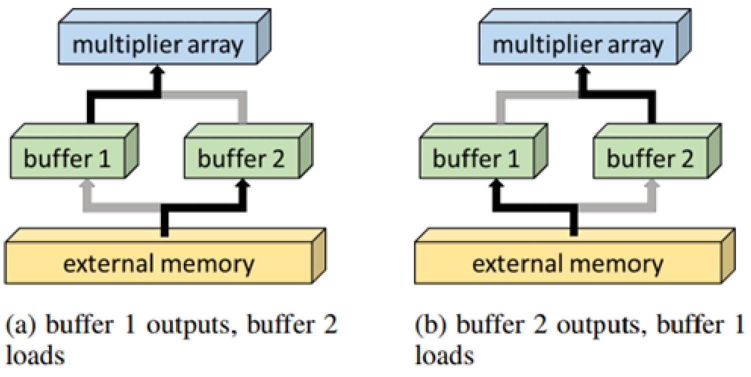


Figure 3.17: Weight buffer in the ping-pong structure (from [32]).

On work [33] the 8 convolutional modules read weights from off-chip memory simultaneously during computation. Since FC layers contribute to most data access, they use a batch-based computing method for FC layers, as shown in figure 3.18. In this batch-based computing, multiple-input feature vectors are computed as a batch in the FC layers. N (batch size) is the number of input vectors, so the operation amount can be increased by N times without increasing the number of data accesses for weights. In figure 3.19, m is the input parallelism and n the output parallelism. The number of $m \times n$ multiplications that can operate concurrently will depend on the available hardware resources. Therefore, to be possible to have the FC layers computing simultaneously without waiting for weights, $m \times n$ weights need to be ready for the next $N \times m \times n$ multiplication, where N should be no less than the number of required clock cycles to read $m \times n$ weights. Batch-based computing method can bring two issues. Although reducing data access, batch-based computing increases the latency for the first output result. The second issue is expanding the required on-chip buffers by N times for FC layers.

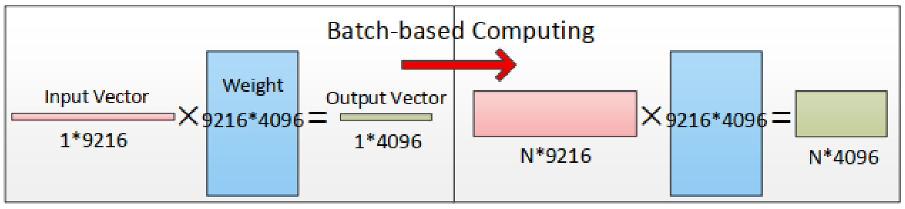


Figure 3.18: Batch-based computing method in the FC layer (from [33]).

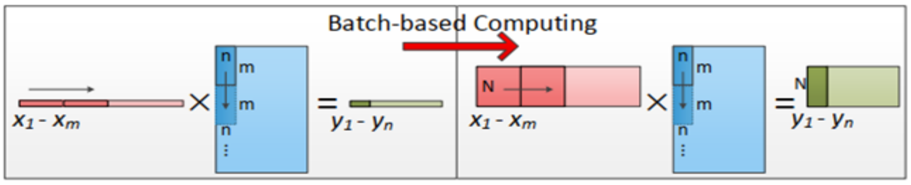


Figure 3.19: Computing pattern for the first and third FC layers; the weight window in the weight matrix shifts vertically (from [33]).

To eliminate the second issue, the computing pattern of FC layers is rearranged as presented in figure 3.20 to reduce the buffer requirements.

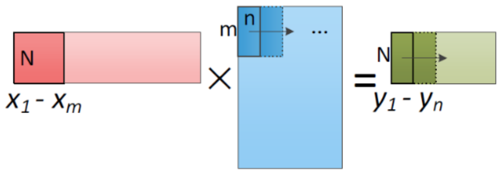


Figure 3.20: Computing pattern for the second FC layer; the weight window in the weight matrix shifts horizontally (from [33])

Firstly, this new pattern generates the temporary results for all the output neurons, using only x_1 to x_m as the input. Then, these temporary results are updated taking x_{m+1} to x_{2m} as the input. Contrary to the pattern of figure 3.19, in this one, the window of size $m \times n$ in the weights matrix shifts horizontally. These two patterns are used for the FC layers, in an alternative manner. The use of vertically shifting in the first FC layer generates the temporary results for y_1 to y_n. Therefore, the second FC layer starts with the inputs from y_1 to y_n using the horizontally pattern. The second FC layer starts before the generation of the whole input vector from the first layer. Thus, it is only necessary to store $N \times n$ neurons. However, in the second FC layer, there is still the need to store all the output neuron temporary results. The third FC layer uses the vertically shifting. Using this strategy, the required number of buffers can be significantly reduced.

3.2.4. Tilling and reuse

Due to the limited on-chip memory, some implementations use tilling and reuse of memory blocks [24, 30]. Tilling divides the IFM into smaller groups of pixels and sends these groups to the PE. This allows performing the convolution by parts when there is not enough memory to store all the IFMs at once. Since some pixels are needed more than one time and to avoid repeating data communications, some IFM pixels are stored on the PE to be reused when needed again. Figure 3.21 shows the tilling and reuse method used in [24]. For convolutional layers (figure 3.21 a)), the IFMs is tilled by the factor T_r for rows and T_c for columns. Each tiled block will be reused several times, defined by $reuse\_times$. The process begins by sending the tiled block (of the input image or IFMs) to the input buffers, together with some of the layer parameters. Since the number of parameters is too large to be all stored at once to on-chip buffers, these parameters are also tiled. To keep the PEs working without interrupts while the PE is processing one tile of the IFMs and weights, the other buffer is receiving another tile of weights, as shown in figure 3.21 b). This allows to hide the latency and reuse the IFMs tiles until all the weights are convolved with this block. After this, the PE receives a new tile of IFMs and restarts the process. For FC layers, the strategy is to apply tiling to each matrix to divide them into tiles of $T_i \times T_o$.

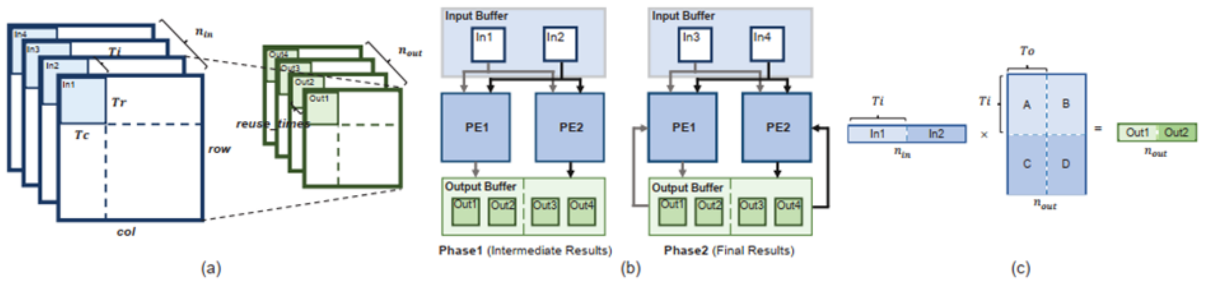


Figure 3.21: Workload schedule for CONV layers and FC layers: (a) Tilling and reuse of feature maps in CONV layers; (b) two phases in the execution of CONV layers; (c) workload schedule in FC layers (from [24]).

3.3. Conclusion

This chapter presented several works where the common objective was to implement CNNs in embedded systems. Two main approaches were aborded: optimization of the CNN model and design of the hardware. From the first one, in this work, pruning will not be considered because it implies changing the model architecture, which is not the objective for this project. On the other side, the quantization of the MobileNets model is essential due to memory limitations and to reduce the number of communications. Using 16 or 8 bits fixed-point to represent the model parameters, implies a reduction in the model of 50% and 75%, respectively. Besides, on FPGA embedded systems, fixed-point uses fewer resources and power consumption and offers better computational performance than floating-point.

Regarding hardware design, this work will use two PEs, one to perform the depthwise stage and the other to perform the pointwise stage. The standard convolution stage, pooling and FC layers will be performed by the software. Parallelism will also be optimized to allow multiple MAC in parallel and the use of several PEs working simultaneously to produce faster results. A pipeline implementation will also be used to produce a constant flow of results. Lastly, tiling and reuse will be explored for the depthwise convolution of the first 6 layers, which have the larger IFMs.

Chapter 4

MobileNets embedded software design

This chapter explains the entire developing process of the MobileNets inference model and the quantization strategy adopted. Firstly, an algorithm developed in C language is presented to test and understand how the inference process works. Then a study of the model is conducted to estimate the necessary memory, total operations and execution time of each convolution layer and activation function. Finally, a detailed analysis of fixed- and floating-point quantization schemes is performed to determine the most efficient for this project.

4.1. Algorithm implementation

The MobileNets model was developed using the Keras [35] deep learning framework with Python and TensorFlow backend. Algorithm 4 presents the pseudo-code of the MobileNets inference process. The input is a 224×224 size image in RGB format normalized between -1 and 1 and quantized with 8 bits (Q2.6). Then, the first stage of the inference process is performed using functions $Half\_Zero\_Padding()$, $3D\_Convolution()$, $BatchNorm()$ and $ReLU6()$. After this, the depthwise and pointwise stages are performed 26 times with different intermediate feature maps, weights and Batch Norm parameters. The algorithm ends with $Pooling()$, $Fully\_Connected()$ and $SoftMax()$. All the functions mentioned here are performed, as described in section 2.5.

The standard MobileNets is a 32-bit floating-point model, with 4,253,864 parameters, that can classify the 1,000 different classes of the ImageNet test dataset with a 70.6% top 1 and 89.9% top 5 accuracies. In this project, the dataset used is the ImageNet validation dataset, the same used by Keras developers, who report 70.4% top 1 and 89.5% top 5 accuracies [36]. Therefore, these will be the values used to compare with the accuracies achieved using the model developed in this work, including the quantization schemes analysed in 4.3 and the hardware/software implementation in section 5.

After developing the C program and before testing the final accuracy, the dataset, i.e., the images, needs to be normalized. The ImageNet validation dataset is composed of 50,000 images with 1,000 different categories, and these images have different sizes, and some are in a grayscale type. In the first place, the image must be loaded in a floating-point format. Then the smallest side of the image is resized to 256 pixels using bicubic interpolation over 4×4 pixels, and the largest size of the image is also resized to maintain the initial aspect ratio. The next step is to crop a central 224×224 window from the resized image. Lastly, the image is saved in RGB format. With the dataset prepared, the C program was tested and achieved a 70.5% top 1 and 89.6% top 5 accuracies, which is consistent with the values in the Keras project.

Algorithm 4 – MobileNets inference process

```
Input: Image $I$ in RGB format
Output: Values $S$ of the activation function softmax

$I_N \leftarrow Normalize(I)$
$Q_N \leftarrow Quantization(I_N)$
$C \leftarrow Half\_Zero\_Padding(Q_N)$
$C_{3D} \leftarrow 3D\_Convolution(C)$
$B \leftarrow BatchNorm(C_{3D})$
$R \leftarrow ReLU6(B)$

$\boldsymbol{for}\ iter = 1\ to\ Layers\ \boldsymbol{do}$    //Number of DW and PW layers
    $\boldsymbol{if}\ stride\ 1\ \boldsymbol{then}$
        $C \leftarrow Zero\_Padding(R)$
    $\boldsymbol{else\ if}\ stride\ 2\ \boldsymbol{then}$
        $C \leftarrow Half\_Zero\_Padding(R)$
    $\boldsymbol{end\ if}$

    // Depthwise stage
    $C_{2D} \leftarrow 2D\_Convolution(C)$
    $B \leftarrow BatchNorm(C_{2D})$
    $R \leftarrow ReLU6(B)$

    // Pointwise stage
    $C_{1D} \leftarrow 1D\_Convolution(R)$
    $B \leftarrow BatchNorm(C_{1D})$
    $R \leftarrow ReLU6(B)$
$\boldsymbol{end\ for}$

$P \leftarrow Pooling(R)$
$FC \leftarrow Fully\_Connected(P)$
$S \leftarrow Softmax(FC)$
```

4.2. MobileNets memory requirements, number of operations, and execution time

The evaluation of the memory requirements of the model and of the execution time of each layer and activation function allows understanding the layers and functions that should be implemented first in hardware and the maximum data sent in each communication to speed up the inference process. Table 4.1 presents the memory used, total operations and execution time for all the convolutional layers, as well as for the SoftMax function, of the MobileNets 32-bit floating-point model. The first column (left) shows the type of convolution, and the stride applied. They are performed in the sequence presented in the table, and the 5x means that there is a depthwise layer followed by a pointwise layer with the same characteristics that are performed five times. The next four columns present the size and quantity of the input kernels and images for each layer. Then the following two columns show an approximation in Kilobytes (KB) of the necessary memory for the kernels and images on each layer. The table ends with the total operations performed in each layer (multiplications and sums) and the respective execution time in milliseconds (ms). The model was tested on one ARM processor of the Zynq 7010 with the C program complied with the O3 optimization because this optimization proved to be the one with the least execution time.

Table 4.1: Memory required, number of operations and execution time for each convolutional, pooling and SoftMax layer of MobileNets standard model.

Type/Stride	Filter shape	Total filters	IFM size	Total IFMs	Kernel memory (KB)	Image memory (KB)	Total operations	ARM time (ms)
3D Conv/2	3 × 3 × 3	32	224 × 224	3	3	602	21,676,032	100.6
DW / 1	3 × 3	32	112 × 112	32	1	1,606	7,225,344	45.5
PW / 1	1 × 1 × 32	64	112 × 112	32	8	1,606	51,380,224	275.6
DW / 2	3 × 3	64	112 × 112	64	2	3,211	3,612,672	33.0
PW / 1	1 × 1 × 64	128	56 × 56	64	33	803	51,380,224	237.2
DW / 1	3 × 3	128	56 × 56	128	5	1,606	7,225,344	45.2
PW / 1	1 × 1 × 128	128	56 × 56	128	66	1,606	102,760,448	456.6
DW / 2	3 × 3	128	56 × 56	128	5	1,606	1,806,336	15.4
PW / 1	1 × 1 × 128	256	28 × 28	128	131	401	51,380,224	218.7
DW / 1	3 × 3	256	28 × 28	256	9	803	3,612,672	21.1
PW / 1	1 × 1 × 256	256	28 × 28	256	262	803	102,760,448	447.1
DW / 2	3 × 3	256	28 × 28	256	9	803	903,168	7.9
PW / 1	1 × 1 × 256	512	14 × 14	256	524	201	51,380,224	215.8
5x DW / 1	3 × 3	512	14 × 14	512	18	401	1,806,336	10.3
PW / 1	1 × 1 × 512	512	14 × 14	512	1,049	401	102,760,448	428.8
DW / 2	3 × 3	512	14 × 14	512	18	401	451,584	4.0
PW / 1	1 × 1 × 512	1,024	7 × 7	512	2,097	100	51,380,224	214.4
DW / 1	3 × 3	1,024	7 × 7	1,024	37	201	903,168	5.3
PW / 1	1 × 1 × 1,024	1,024	7 × 7	1,024	4,194	201	102,760,448	427.5
Avg Pool / 1	7 × 7	1	7 × 7	1,024	0	201	51,200	0.4
FC / 1	1 x 1 x 1,024	1,000	1 × 1	1,024	4,100	4	2,049,000	9.0
SoftMax / 1	Classifier	0	1 × 1	1,000	0	4	4,000	1.7

Table 4.2: Memory required, number of operations and execution time for Batch Norm and ReLU6 functions of MobileNets standard model.

Layer type	Parameters	IFM size	Total IFMs	Parameters memory (KB)	Image memory (KB)	Total operations	ARM time (ms)
BN (Conv)	128	112 × 112	32	0.5	1,606	1,605,632	13.6
ReLU6 (Conv)	0	112 × 112	32	0	1,606	802,816	7.4
BN (DW)	128	112 × 112	32	0.5	1,606	1,605,632	13.7
ReLU6 (DW)	0	112 × 112	32	0	1,606	802,816	7.5
BN (PW)	256	112 × 112	64	1	3,211	3,211,264	27.2
ReLU6 (PW)	0	112 × 112	64	0	3,211	1,605,632	17.3
BN (DW)	256	56 × 56	64	1	803	802,816	6.8
ReLU6 (DW)	0	56 × 56	64	0	803	401,408	4.4
BN (PW)	512	56 × 56	128	2	1,606	1,605,632	13.6
ReLU6 (PW)	0	56 × 56	128	0	1,606	802,816	8.2
BN (DW)	512	56 × 56	128	2	1,606	1,605,632	13.8
ReLU6(DW)	0	56 × 56	128	0	1,606	802,816	10.1
BN (PW)	512	56 × 56	128	2	1,606	1,605,632	13.6
ReLU6 (PW)	0	56 × 56	128	0	1,606	802,816	9.4
BN (DW)	512	28 × 28	128	2	401	401,408	3.4
ReLU6 (DW)	0	28 × 28	128	0	401	200,704	2.2
BN (PW)	1,024	28 × 28	256	4	803	802,816	6.8
ReLU6 (PW)	0	28 × 28	256	0	803	401,408	4.0
BN (DW)	1,024	28 × 28	256	8	803	802,816	6.9
ReLU6 (DW)	0	28 × 28	256	0	803	401,408	5.3
BN (PW)	1,024	28 × 28	256	8	803	802,816	6.8
ReLU6 (PW)	0	28 × 28	256	0	803	401,408	4.4
BN (DW)	1,024	14 × 14	256	8	201	200,704	1.7
ReLU6 (DW)	0	14 × 14	256	0	201	100,352	1.2
BN (PW)	2,048	14 × 14	512	8	401	401,408	3.4
ReLU6 (PW)	0	14 × 14	512	0	401	200,704	2.0
5 x **BN (DW)**	2,048	14 × 14	512	8	401	401,408	3.4
ReLU6 (DW)	0	14 × 14	512	0	401	200,704	2.7
BN (PW)	2,048	14 × 14	512	8	401	401,408	3.4
ReLU6 (PW)	0	14 × 14	512	0	401	200,704	2.2
BN (DW)	2,048	7 × 7	512	8	100	100,352	0.9
ReLU6 (DW)	0	7 × 7	512	0	100	50,176	0.5
BN (PW)	4,096	7 × 7	1,024	16	201	200,704	1.7
ReLU6 (PW)	0	7 × 7	1,024	0	201	100,352	1.1
BN (DW)	4,096	7 × 7	1,024	16	201	200,704	1.8
ReLU6 (DW)	0	7 × 7	1,024	0	201	100,352	1.3
BN (PW)	4,096	7 × 7	1,024	16	201	200,704	1.7
ReLU6 (PW)	0	7 × 7	1,024	0	201	100,352	1.0

Table 4.2 completes the analysis of the MobileNets model and the inference process showing the metrics for the remaining functions. Each pair of Batch Norm (BN) and ReLU6 on table 4.2 is preceded by the convolutional layer of the previous table, and once again, all the Batch Norm and ReLU6 functions are performed in the sequence presented in the table. The three columns next to the layer type indicate the size of the Batch Norm parameters and the size and number of the IFMs. Then, one column presents the memory necessary for the parameters, and the other, the memory necessary for the IFMs. Finally, the last two columns show the total number of operations and execution time. For the Batch Norm function, the operations are multiplications and sums. However, for the ReLU6, the operations are comparisons, and the total operations represent the worst case since the number of operations varies according to the input image.

Observing table 4.1 and table 4.2, the conclusion is that the pointwise convolutions dominate the execution time of the inference process of MobileNets. Table 4.3 groups all the operations and execution time of the same layers and functions. The pointwise convolution represents 92% of the total operations, which consume about 88% of the execution time of the inference process. The Batch Norm and ReLU6 functions combined have almost the same number of operations of the depthwise convolutions, and their execution time combined is more significant than the convolutional layer, which makes these functions also candidates for hardware implementation.

Table 4.3: Total parameters, operations and execution time by layer and function type of the MobileNets standard model.

Layer/function type	Total parameters	Total parameters (%)	Total operations	Total operations (%)	Total exec. time (ms)	Total exec. time (%)
Standard conv.	864	0.02	21,676,032	1.86	100.6	1.91
Depthwise conv.	44,640	1.05	34,771,968	2.98	228.6	4.35
Pointwise conv.	3,139,584	73.80	1,078,984,704	92.39	4636.5	88.17
Pooling	0	0.00	51,200	0.00	0.4	0.01
Fully Connected	1,025,000	24.10	2,049,000	0.18	9.0	0.17
SoftMax	0	0.00	-	-	1.7	0.03
Batch Norm	43,776	1.03	20,170,752	1.73	171.4	3.26
ReLU	0	0.00	10,085,376	0.86	110.7	2.10
Total	4,253,864	100.00	1,167,789,032	100.00	5,258.8	100.00

4.3. Quantization

As referred in the previous section, the MobileNets model is mostly composed of pointwise parameters. Thus, the focus of the following quantization strategies was in reducing the number of bits used by these parameters to the minimum acceptable, which for this work is a loss of accuracy smaller than 1%. During inference, the OFMs are also represented with the minimum of bits because some of these OFMs will be exchanged between the on-chip and off-chip memory, which means that the fewer bits are used to represent the values of these OFMs, the faster the communications will be, and less on-chip memory will be required to store these values. Since the 3D convolution, depthwise and Batch Norm parameters represent only about 2% of all the model parameters, these will be quantized only to 16 bits. Although FC parameters are about 24% of the model size, these parameters are also quantized to 16 bits because this convolution will be performed on the software application.

Two strategies of post-training quantization were developed and tested to evaluate the accuracy and behaviour of the model after quantization: fixed-point quantization of weights and activations and custom floating-point pointwise weights quantization. The goal was to find the best quantization scheme that preserves most of the model accuracy using less memory. This evaluation was performed using post-training quantization (section 3.1). Unlike quantized aware training, post-training quantization strategy does not need a new training of the model and therefore is very fast to implement. Also, the loss of accuracy is minimal when the strategy is carefully applied, as indicated by the results shown in this section below. The final hardware/software CNN application will equally accelerate the quantized model independently of whether post-quantization training or quantized aware training is used.

4.3.1. Fixed-point quantization of weights and activations

Since the objective of the quantization strategy is to reduce the size of the model, the first step of this quantization study was to convert the weights and Batch Norm parameters from 32-bit floating-point to 16 or 8-bit fixed-point, following the practices in state-of-the-art post-training quantization works [22] [24]. The study consists of evaluating the behaviour of the network when the parameters are quantized only with 16 bits, 8 bits or both without retraining the model. Dynamic quantization is used, where the number of integers and fractional bits of each layer is chosen according to the range of its weights and Batch Norm parameters. Figure 4.1 illustrates an example where layer 1 has weight values $a \in [-20.750, 31.750]$ and layer 2 has weight values $b \in [-1.675, 0.747]$. Applying the same 8-bit fixed-point quantization to both layers will result in all weights being represented as Q6.2 (6 bits for the integer part and 2 bits for the fractional part). As shown in figure 4.1 (left), the better approximation possible with two fractional bits of the maximum (absolute) of layer 2 is -1.500. Instead, applying 8-bit fixed-point quantization independently on each layer will result in the same Q6.2 representation for layer 1, but the weights of layer 2 will be represented in Q2.6 format (see figure 4.1 (right)). This strategy improves the precision of the parameters on layer 2 because now the fractional part can use six of the eight bits available. Consequently, now the better approximation of the maximum (absolute) value of layer two is -1.672.

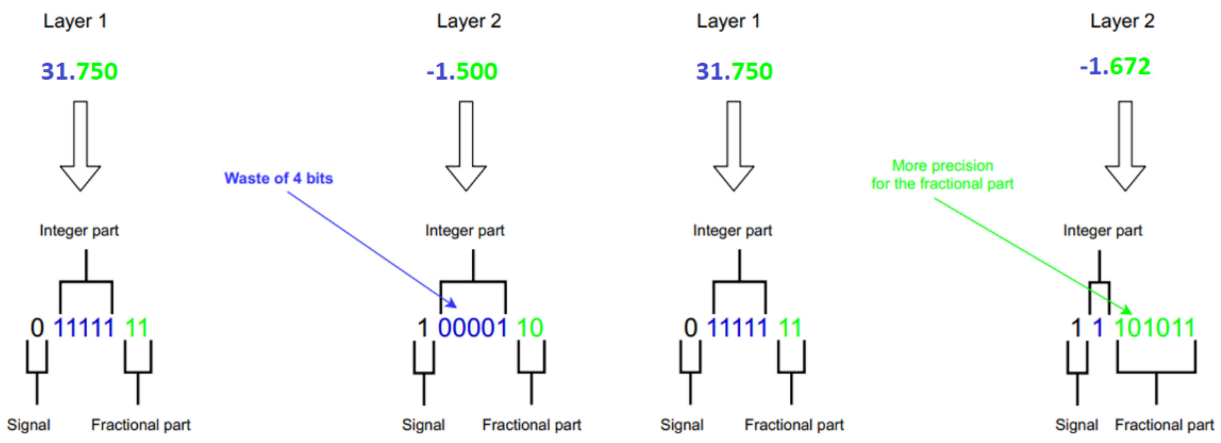


Figure 4.1: On the left: same quantization to both layers. On the right: different quantization by layer.

To perform the dynamic quantization, the quantization algorithm first searches the maximum absolute value of the model parameters, for each layer, to detect how many bits need to be used in the integer part of the number. All the layer parameters are quantized with the maximum bits obtained for the integer part, which will define the representation used for that layer. The described steps of the quantization process are summed up in algorithm 5.

Algorithm 5 – Quantization of MobileNets model

```
Input: MobileNets 32 floating-point parameters.
Output MobileNets quantized model.

for iter = 1 to Layers do                          //Number of layers
  max_bits ← Search_max(layer)                     //Searches on all layers
  Q_model ← Quantize(layer, max_bits )             //Quantizes values from the layer
end for

Q_binary ← Save_to_bin(Q_model)                    //Saves the model
```

For batch norm layers, and before the quantization of the Batch Norm parameters, an extra procedure is applied in order to combine the two multiplications in a single one. Thus, equation (2.18) is reduced to equation (4.1):

$$\gamma(x_i - \mu_\beta)V + \beta = (x_i - \mu_\beta)\gamma V + \beta = (x_i - \mu_\beta)P + \beta \tag{4.1}$$

where P is the multiplication of the gamma and variance parameters of the activation layer before the quantization process. Applying this procedure also represents a reduction of 25% in the Batch Norm parameters and therefore in the number of operations of this function.

The OFMs of the convolution layers are quantized with either 8 or 16-bits and the appropriate fixed-point scales are selected independently for each layer, following the same approached used for the weights. As the ReLU6 function bounds the values between 0 and 6, the pixels of the OFMs can be represented using Q3.5 (or Q3.13) unsigned. Therefore, after applying the Batch Norm function to the convolutional layer results, the number is aligned with the fractional bits used in ReLU6.

Table 4.4 presents the results obtained for the evaluated models. Model 1 corresponds to the standard MobileNets model that uses 32-bit floating-point precision. When the model is quantized with 16 bits (model 2), the size of the model is half of the standard and the accuracy is not affected However, when the weights of the pointwise layers are quantized using 8 bits (model 3), the accuracy drops more than 8%. Models 4 to 8 use the fixed-point quantization representation combined with dynamic OFM quantization. The strategy of reducing the precision of the images does not incur a significant accuracy loss (models 4, 5 and 6), but again, the accuracy drops significantly when the pointwise weights are quantized using 8 bits (models 7 and 8).

Table 4.4: MobileNets standard model vs fixed-point quantization accuracies and sizes.

Model	Input images	3D conv weights	DW weights	DW OFM	PW weights	PW OFM	BN param.	FC IFM	FC weights	Top 1 (%)	Top 5 (%)	Size (MB)
1	32F	32F	32F	32F	32F	32F	32F	32F	32F	70.54	89.58	17.0
2	8	16	16	16	16	16	16	16	16	70.56	89.50	8.5
3	8	16	16	16	8	16	16	16	16	62.20	83.95	5.4
4	8	16	16	16	16	8	16	8	16	69.77	89.05	8.5
5	8	16	16	8	16	16	16	16	16	69.63	88.93	8.5
6	8	16	16	8	16	8	16	8	16	68.94	88.51	8.5
7	8	16	16	16	8	8	16	8	16	61.91	83.65	5.4
8	8	16	16	8	8	16	16	16	16	60.73	82.85	5.4

4.3.2. Custom floating-point pointwise weights quantization

The experiments of the previous strategy showed that when the pointwise weights were quantized with 8 bits, the accuracy dropped roughly between 8 and 10%. To improve the accuracy without using 16 bits overall, a custom floating-point representation is proposed. This representation uses 12 bits distributed as follows: 8 for the quantized number and 4 for the exponent as presented in figure 4.2.

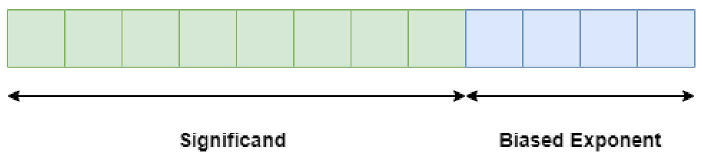


Figure 4.2: Distribution of the 12 bits of the proposed floating-point.

The real number can be represented by the proposed floating-point using equation (4.2) :

$$R = Significand \times 2^{-(Biased\ Exponent\ +\ Bias\ )} \quad (4.2)$$

where R is the real value to be quantized and $Bias$ the number of bits of the fractional part of the base representation obtained using the $Search\_\max$ function of algorithm 5. Since the Biased Exponent uses 4 bits and takes values between 0 and 15, the Exponent varies between $0 + Bias$ and $15 + Bias$. After obtaining the base representation, the algorithm selects the Exponent that allows each weight to be represented using the maximum significant bits. Using this floating-point representation, the memory requirements for the parameters is reduced in 20% compared to the 16-bit fixed-point (model 2), and 60% compared to the standard model. In terms of hardware requirements, there is only the need to insert a right shift operation after the multiplication, as shown in equation (4.3):

$$Conv_{pointwise} = \sum_{1}^{I} [(Qp_i \times Qw_i) \gg Biased\ Exponent_i] \tag{4.3}$$

where Qp_i is the ith quantized pixel, Qw_i is the ith quantized weight from the ith pixel, and $Biased\ Exponent_i$ the Based Exponent from the ith quantized weight.

Table 4.5 compares the results obtained with the custom floating-point quantization with those obtained with the fixed-point quantization (in the previous section). Comparing model 9 with model 2 and model 10 with model 4, shows that using the custom floating-point with 12 bits to represent the pointwise parameters practically maintains the same accuracy than using a 16-bit fixed-point representation, while requiring 25% fewer bits. As an example, the last pointwise layer has $1 \times 1 \times 1024 \times 1024 = 1{,}048{,}576$ parameters (table 4.1), which means that to perform the last layer on-chip, roughly 2.1 MegaBytes (MB) must be transferred to the FPGA local memory, when using a 16-bit fixed point. Using the 12-bit custom floating-point implementation, this value is reduced to 1.6 MB. The last models of table 4.5 are two variants of the custom floating-point strategy. Model 11 uses 3 bits to represent the exponent (8Exp3), and model 12 uses 2 bits (8Exp2). This reduces the data required to perform the last pointwise layer, which is approximately 1.4MB for model 11 and 1.3MB for model 12, but slightly reduces the accuracy.

Table 4.5: MobileNets standard model vs fixed and floating-point quantization accuracies and sizes.

Model	Input images	3D conv weights	DW weights	DW OFM	PW weights	PW OFM	BN param.	FC IFM	FC weights	Top 1 (%)	Top 5 (%)	Size (MB)
1	32F	32F	32F	32F	32F	32F	32F	32F	32F	70.54	89.58	17.0
2	8	16	16	16	16	16	16	16	16	70.56	89.50	8.5
4	8	16	16	16	16	8	16	8	16	69.77	89.05	8.5
9	8	16	16	16	8Exp4	16	16	16	16	70.50	89.54	6.9
10	8	16	16	16	8Exp4	8	16	8	16	69.76	89.02	6.9
11	8	16	16	16	8Exp3	8	16	8	16	69.72	89.02	6.5
12	8	16	16	16	8Exp2	8	16	8	16	69.56	88.98	6.2

4.3.3. Quantization - Final considerations

Figure 4.3 summarizes all the information of table 4.4 and table 4.5 on a common graph. The blue bars represent the model size with all the parameters while the red and purple bars represent the total size of the depthwise and pointwise OFMs, respectively. The yellow line shows the top 1 and the green line shows the top 5 accuracies.

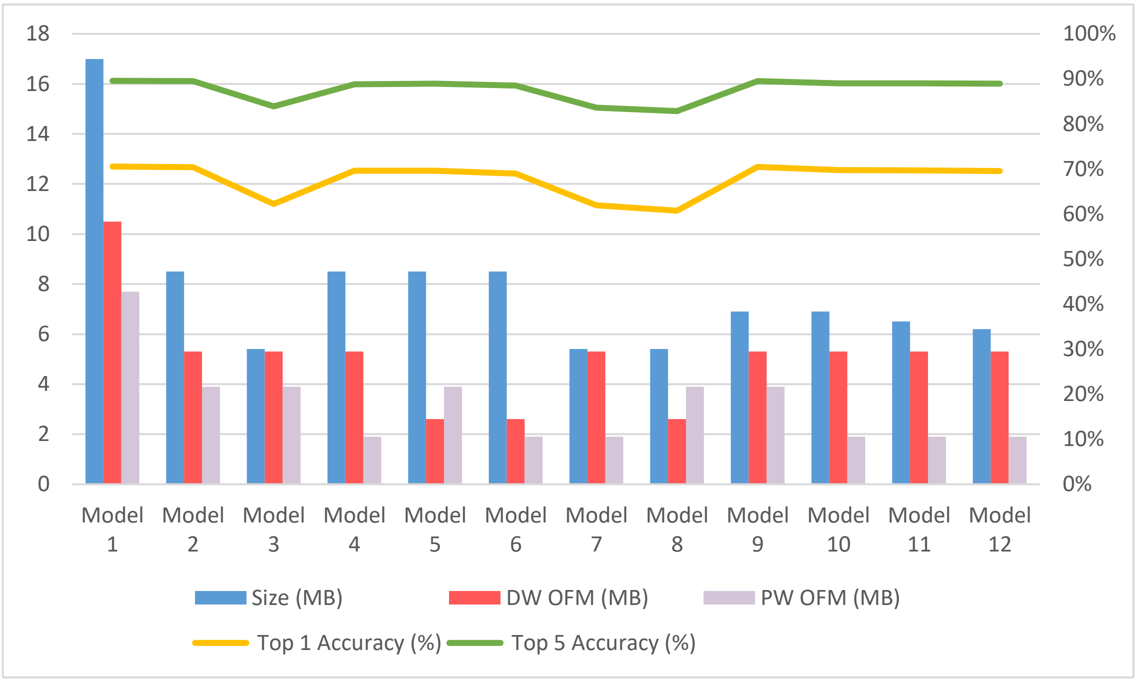


Figure 4.3: MobileNets model size vs top 1 and top 5 accuracies.

Models 10, 11 and 12 are the best for this project because they are the smallest models that use dynamic OFM quantization and meet the maximum accuracy loss of 1%. From these three, model 10 is selected because it has a slightly better accuracy (0.2% better than model 12). Also, model 11 uses 3 bits for the exponent, which is somewhat less efficient to group than using multiples powers of two.

Table 4.6 details the quantization process of the pointwise parameters of model 10 with the custom floating-point. For each pointwise layer it presents the base representation used for the maximum absolute value. Columns 3 and 4 show the maximum exponent used in the parameter representation in each layer (bias + exponent) and the respective minimum and maximum non zero absolute values. The last two columns indicate the total layer parameters and the percentage of zero parameters recovered using the custom floating-point representation, compared to the 16-bit fixed-point. Figure 4.4 compares the number of zero values after quantization for each pointwise layer when 16-bit fixed-point and the proposed 12-bit floating-point are used. The conclusion is that the custom floating-point is able to represent more effectively the very small (near zero) weights, as it reduces by almost 14% the number of pointwise parameters equal to zero after quantization.

Table 4.6: 12-bit custom floating-point applied to all pointwise layers of the MobileNets model.

Pointwise layer	Custom floating-point base representation	Maximum exponent	Minimum and maximum non zero absolute values	Total parameters	Percentage of zero parameters recovered (%)
1	Q2.6	21	~0.00000024, 1	2,048	21.1
2	Q1.7	22	~0.00000012, 0	8,192	22.9
3	Q2.6	21	~0.00000024, 1	16,384	1.85
4	Q2.6	21	~0.00000024, 1	32,768	18.5
5	Q1.7	22	~0.00000012, 0	65,536	100.0
6	Q1.7	22	~0.00000012, 0	131,072	97.9
7	Q1.7	22	~0.00000012, 0	262,144	98.9
8	Q1.7	22	~0.00000012, 0	262,144	14.2
9	Q1.7	22	~0.00000012, 0	262,144	20.7
10	Q1.7	22	~0.00000012, 0	262,144	12.6
11	Q2.6	21	~0.00000024, 1	262,144	99.3
12	Q1.7	22	~0.00000012, 0	524,288	7.5
13	Q1.7	22	~0.00000012, 0	1,048,576	9.0

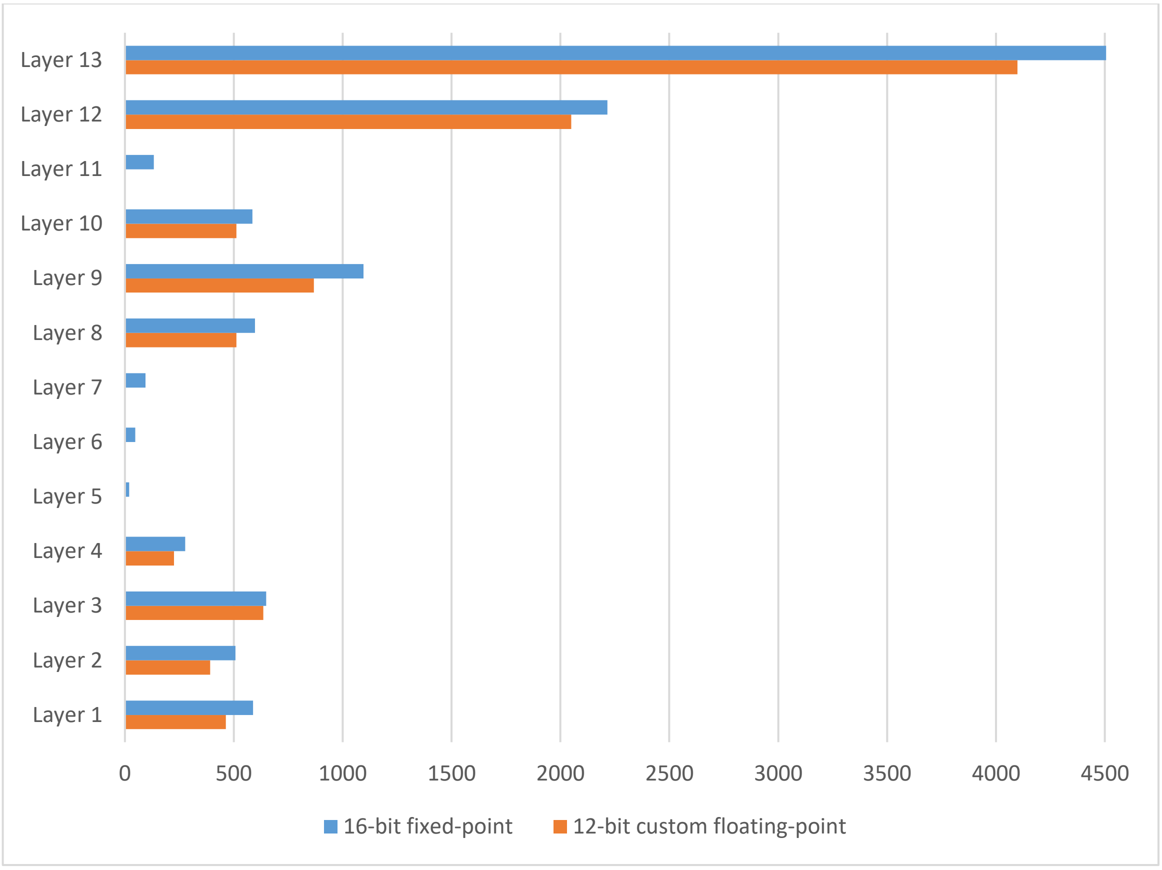


Figure 4.4: Total number of parameters equal to zero after quantization when 16-bit fixed-point and 12-bit custom floating-point are used.

4.4. Conclusion

This chapter started by introducing an algorithm developed in C language to perform the inference process of MobileNets model. An analysis of the memory requirements, number of operations and execution time identified the layers and functions that consume more memory and take more time to execute. This study concluded that the number of operations and memory requirements are dominated by the parameters of the pointwise layers. Therefore, the focus of the quantization approach was quantizing these parameters with the minimum possible bits. The accuracy and memory requirements of 12 models, using 16 and 8-bit representations for the OFM, and different fixed- and floating-point quantizations for the weights, were evaluated on the ImageNet validation dataset using the same conditions as the standard MobileNets model. The result showed that using a custom 12-bit floating-point representation for the pointwise weights parameters achieves a better trade-off between accuracy and model size than the corresponding 16-bit fixed model. Thus, this work will use the 12-bit floating-point quantized model (model 10) presented in section 4.3.3. This allows having a MobileNets model about 60% smaller than the standard with only a very small accuracy reduction of 0.78%.

Chapter 5

Hardware design

The following chapter describes in detail the hardware implementation to speed up the inference process of the MobileNets model. Firstly, an overview of the Zynq-7000 SoC FPGA is introduced to understand what resources are available on the target device to start outlining the hardware design. Then the bus protocols and standard Intellectual Propertys (IPs) from Xilinx are presented and its use in this project is explained. After that, the custom IPs are described individually, detailing their structure, the functionalities and features used, and explaining the specific design options taken. Finally, this chapter ends with the final hardware/software scheme of the project and possible upgrades.

5.1. Zynq-7000 SoC FPGA

An SoC FPGA integrates both microprocessors and FPGAs architectures into a single device, which also include a set of peripherals, on-chip memory, an FPGA-style logic array, and high-speed transceivers. The main benefits of this integration are lower power consumption, smaller board size and higher-bandwidth communication between the microprocessor and the FPGA. Figure 5.1 presents a block diagram of the Zynq-7000 devices.

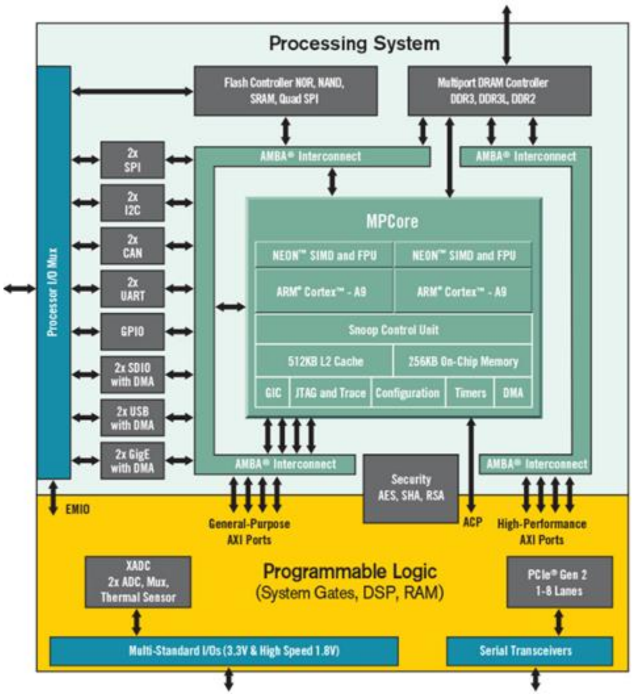


Figure 5.1: Zynq-7000 SoC main block diagram (from [37]).

Two main blocks are highlighted: the Processing System (PS) and the Programmable Logic (PL). The PS system is composed of the elements inside the light green block and the PL by the elements inside the yellow block. There is an Application Processing Unit (APU) that contains several modules, namely a dual-core ARM Cortex-A9 processor. Each ARM processor has a Level 1 (L1) cache memory with 32KB and a shared Level 2 (L2) cache memory with 512KB. A 256 KB On-Chip Memory is shared between the two cores, and each core has a floating-point unit (FPU) engine. A memory controller allows direct interaction with the external double data rate (DDR) memory.

The interface between the PS and PL can be performed through the following available ports: 4 general-purpose (GP) AXI 32-bit ports (2 masters and 2 slaves), 4 high-performance (HP) AXI slave ports (32 or 64 bits wide) and 1 accelerator coherency port (ACP) 64-bit slave port [38]. Each of these ports can operate up to 150 MHz. The GP ports have an estimated throughput of 600 MB/s and are used to control PL to PS functions or to access the PS I/O peripherals. For large DMA data transfers, the HP ports can support communication rates of up to 1,200 MB/s per interface. The ACP ports have also an estimated throughput of 1,200 MB/s but are used for smaller DMA coherent datasets.

The PL is based on the Xilinx Artix-7 and Kintex-7 FPGA series and is mainly composed of general-purpose FPGA logic fabric, which in turn are composed of slices and Configurable Logic Blocks (CLBs). There is also Input/Output Blocks (IOBs) for interfacing (figure 5.2). A CLB is a small group of logic elements that when linked together by routing resources, execute complex logic functions, implement memory functions, and synchronize code on the FPGA. This small group of logic includes slices, Lookup Tables (LUTs) and Flip-flops (FF). Slice components have the resources to implement combinatorial and sequential logic circuits. A LUT allows projecting a logic function of up to six inputs, a small read-only memory (ROM) or random access memory (RAM), or a shift register. LUTs can be combined to form larger variants of the basic possibilities described.

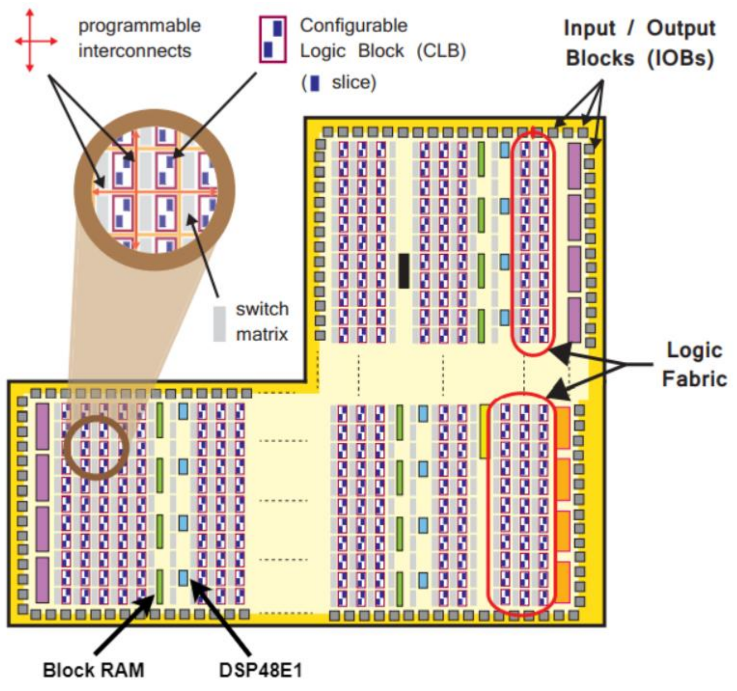

programmable interconnects
Configurable Logic Block (CLB)
(▮ slice)
Input / Output Blocks (IOBs)
switch matrix
Logic Fabric
Block RAM
DSP48E1

Figure 5.2: Zynq 7000 PL diagram (adapted from [38]).

The FPGA Block RAMs can efficiently implement local storage and can be configured to provide different width/depth memory organizations. DSPs provide efficient multiply-add arithmetic using for that, the main features such a pre-adder/subtractor of 25 bits, a 25x18 multiplier and a 48-bit adder/subtractor or logic unit. Figure 5.3 presents the basic functionality of the DSP48E1 block which omits low-level implementation details.

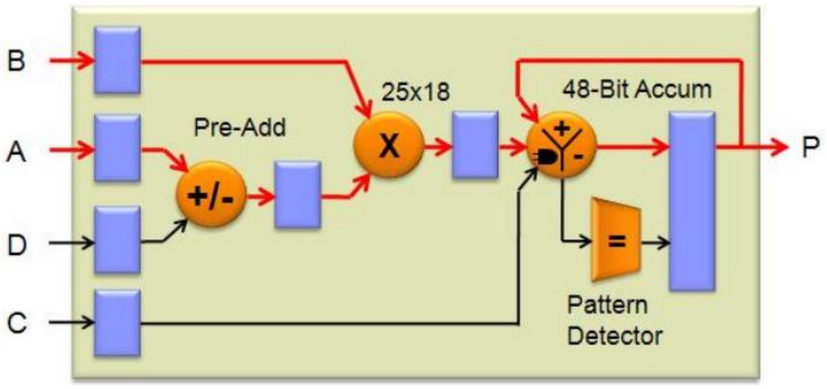


Figure 5.3: DSP48E1 main functions (adapted from [39]).

Observing the inputs A, B, C and D of the figure above, one of the functions that can be implemented is $P = B \times A + P$ (represented with the red arrows) which is the basic set of operations presented in CNNs, the MAC.

An example of an SoC FPGA is the Xilinx Zynq-7010 found in the Zybo Z7-10 development board and used in this work. This device integrates a 667MHz dual-core ARM Cortex-A9 processor with Xilinx 7-series FPGA logic. Other relevant characteristics of the Zybo Z7-10 are summarized in table 5.1.

Table 5.1 Zybo Z7-10 FPGA part and available LUTs, Flip-flops, BRAM and DSPs:

Development board	**Zybo Z7-10**
FPGA part	XC7Z010-1CLG400C
Memory RAM	1GB DDR3L with 32 bits bus @ 1066MHz
FPGA	**Zynq-7010**
Look-up tables	17,600
Flip-Flops	35,200
DSPs	80
Block RAM	270 KB

5.2. Hardware communications

One of the aspects that influences the performance of a hardware system is the number of communications carried out between the elements that compose it. CNN algorithms deal with high transfers of data, and this can affect more the execution time than all the other tasks involved. As presented in section 3.1, the CNN models can be optimized to obtain smaller models, therefore fewer data to be transfer during communications. However, there will be always the need to perform communications, whether to send or receive data, thus it is fundamental to take advantage of the performance-oriented features.

5.2.1. The AXI interconnect system

As said in the previous chapter, the data exchanged between the PS and PL is performed through the AXI interfaces. The AXI interfaces used by the ARM are part of the Advanced Microcontroller Bus Architecture (AMBA) specification [38]. This interconnect uses two interfaces to exchange data, a master and a slave. To ensure the reliability of the transactions, the AXI interface makes use of five different channels namely, Read Address, Write Address, Read Data, Write Data, and Write Response, and each of these channels has its own signals. The channels operate in a handshake mode, established from the master to the slave using valid and ready signals. To perform a transaction, the sender activates the valid signal, which indicates to the receiver that the source is ready to send data. After this, the receiver activates the ready signal, informing the sender that the transaction can take place. When all the data is transferred, the sender deactivates the valid signal and consequently, the ready from the receiver is also deactivated. However, the ready signal may also be deactivated if the receiver is not able to receive more data.

Figure 5.4 presents the functionality behind a read transaction. The master starts by providing the initial address of the data and the control information in the read address channel. When the information is received by the slave, this one responds by sending the data requested through the read data channel.

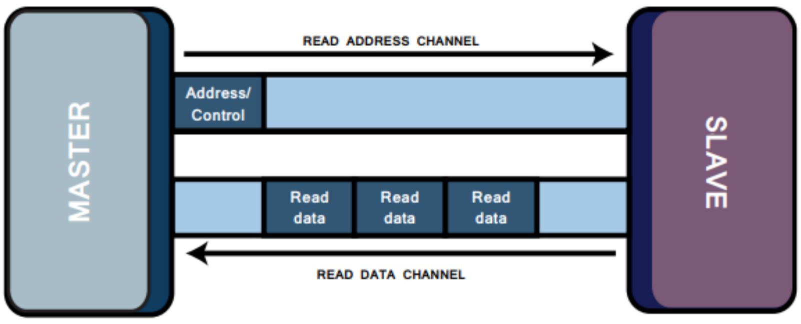


Figure 5.4: AXI read transaction from [38].

A write transaction, which is represented in figure 5.5, also starts with a request by the master, however, this time the request is made in the write address channel and the data is sent in the write data channel. The slave must validate the request to the transaction to occur but also needs to issue a confirmation through the write response channel to inform the success or not of the requested transaction.

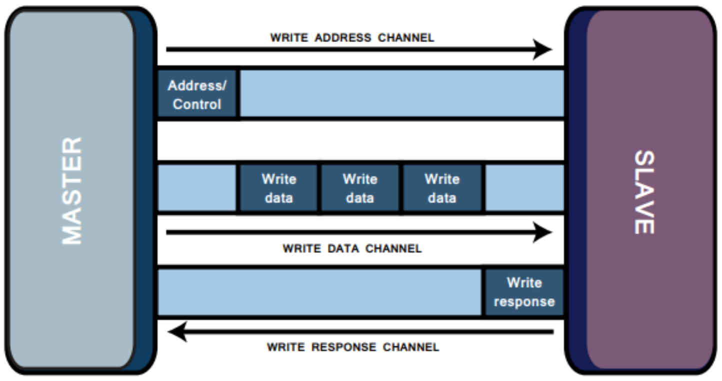


Figure 5.5: AXI write transaction from [38].

5.2.2. AXI4 variants: Full, Lite and Stream

The AXI4 interconnect system includes three AXI4 interface variants which are suitable for different types of applications:

- AXI4-Full: this variant provides the highest performance and is used for memory-mapped links. Uses the five channels and read and write methods described in the previous section, allowing a data burst transfer of up to 256 data beats.
- AXI4-Lite: works the same way that the AX4-Full, although in this variant there is only the possibility of one data transfer per connection (no bursts). Suitable for transfers that do not require large amounts of data.
- AXI4-Stream: is the simplest to implement because there is no address mechanism and uses only one channel. The transfer is always from master to slave, supporting data bursts of unlimited size. This interface is ideal for high-speed streaming data.

This work uses the AXI4-Stream interface to implement the large data communication between the external memory (DDR) and the PL. The parameters of the MobileNets model and the IFMs are sent from the DDR to the PL and the OFMs are sent back from the PL to the DDR.

An AXI4-Stream interface includes the data, the valid, the ready and the last signal. This last signal is activated by the source when the last group of data bits is sent to the receiver, informing that the transaction will end. The state machine that controls the receiver functions, may use this last signal to identify the end of the stream. This allows the loading process to be independent of the size of the data.

5.2.3. AXI Interconnect IP

Xilinx provides an IP that manages and directs the traffic between the attached AXI interfaces. The interconnect can assume simple configurations, where a single master connects with a single slave (conversion only configuration), and more complex configurations where a single master is connected to several slaves (1-to-N Interconnect configuration), or where several masters are connected to a single slave (N-to-1 Interconnect configuration), or yet, where several masters are connected to several slaves (N-to-M Interconnect configuration).

5.3. Memory management

This section describes how the data is transferred between the memories in the system, the organization of the data and the respective access, to maximize the memory bandwidth using the minimum resources required.

5.3.1. DMA controller

This work uses the "AXI Direct Memory Access" from the Xilinx IP catalog to manage the data exchange between the PS and PL (figure 5.6). To start the memory transaction, the DMA controller needs to be provided with the address from where the data will be read (source address), the address from where the data will be written (destination address) and the number of bytes of the transfer (transfer length).

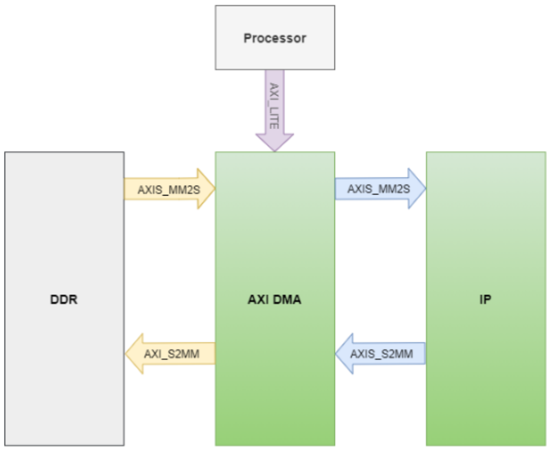


Figure 5.6: Data exchange between PS and PL using DMA adapted (from [40]).

The processor is connected to the DMA through an AXI-lite interface to prepare, start and monitor data transfers. The AXI_MM2S and AXI_S2MM are two memory-mapped AXI4 interfaces that allow the DMA to access directly to the DDR memory. Lastly, AXIS_MM2S and AXIS_S2MM are the AXI4_Stream interfaces used to send (or receive) a continuous stream of data to (from) the IP.

5.3.2. Block RAM

The data sent to the PL can be processed immediately or temporary stored on specific memory elements. Since in this project the AXI4-Stream interface is used to exchange data from the PS and the PL, the most efficient approach is to store the maximum possible data on Block RAMs (BRAMs) (see section 5.1). The BRAMs in the Zynq 7000 series devices can be configured as Single-port Block RAM, Dual-port Block RAM or Simple dual-port Block RAM [38].

All the BRAMs used in this project are configurated in simple dual-port. This allows storing more data that comes from the DDR in each clock cycle but also read data size different from the write port. There is even the possibility to write and read data from the same BRAMs at the same time, as long as the writing and reading addresses are different, otherwise, a collision will occur.

5.3.3. AXI4-Stream Data Width Converter

The read port of the BRAM in the simple dual-port configuration can have different sizes. However, the bus size of the DMA can be configured only with 32 or 64 bits, therefore, the bus of the DMA cannot be connected directly to the input of the data port of the BRAM. One solution is to use the “AXI4-Stream Data Width Converter” IP available on the Xilinx library which converts the input stream width to a higher or lower size, depending on how it is configured. In this project, the bus size of the DMA is configured with 64 bits and the BRAMs have an input of 72 bits wide to store the depthwise weights, 64 bits wide to store the pointwise weights/shifts and IFMs, and 96 bits wide to store the Batch Norm parameters.

5.4. MobileNets padding and tiling

Figure 5.6 shows the PS, the PL, and the communication interfaces between them. The PS is the “ZYNQ7 Processing System” and is responsible, among other tasks, for the execution of the MobileNets software application. As concluded in chapter 4 the depthwise and pointwise stages will be developed on hardware because they represent about 97% of the total execution time of the software application (table 4.3). Stages 7 to 13 are completely performed on hardware, however, stages 1 to 6 require a tiling procedure before starting the stage processing and a map reorganization after the stages are completed. These functions and also the half and complete zero-padding (see section 2.5.2) are performed on the software application, due to the cost it would bring to the hardware applying this procedure on tiled blocks, incurring only a small penalty in the final execution time of the inference process. After performing the half or complete zero-padding (depending on the stride applied) another function performs a tiling process before sending the IFMs to the PL. After receiving the data processed on the PL, the software application reorganizes the tiled blocks, which is the inverse process of the tiling function.

The tiling procedure needs to be done because of memory limitations. Before choosing the size of the tiled block, some aspects were analysed. Firstly, the ideal depth would be 32 because the first depthwise stage has 32 IFMs and the following are all multiple of 32, allowing the pointwise layer to perform 32 MACs at a time. Then, from stages 7 to 12 the size of the depthwise IFMs is 14×14 pixels, which after a complete zero-padding becomes 16×16 pixels. This means that in these stages the tiling process is not required if the size of the IFM is smaller than 16×16 pixels. Lastly, the transfer of a $(16 \times 16 \times 32) \times 8\ bits$ tiled block needs about 6.3KB, which is only 2.3% of the total BRAMs available. The described tiling process is presented in figure 5.7. A group of $16 \times 16 \times 32$ pixels is selected from the IFMs when the stride is 1 (or $15 \times 15 \times 32$ pixels when the stride is 2). The last column between two tiles is sent twice to the hardware and the same happens with the last rows between two tiles, avoiding the extra complexity of implementing the padding in hardware which does not compensate for the small gain in memory and time communication. Also, since the IFMs are stored on a ping-pong buffer memory, the loss on the communication is reduced as while the depthwise convolution is executing, the other memory is loading the new IFMs.

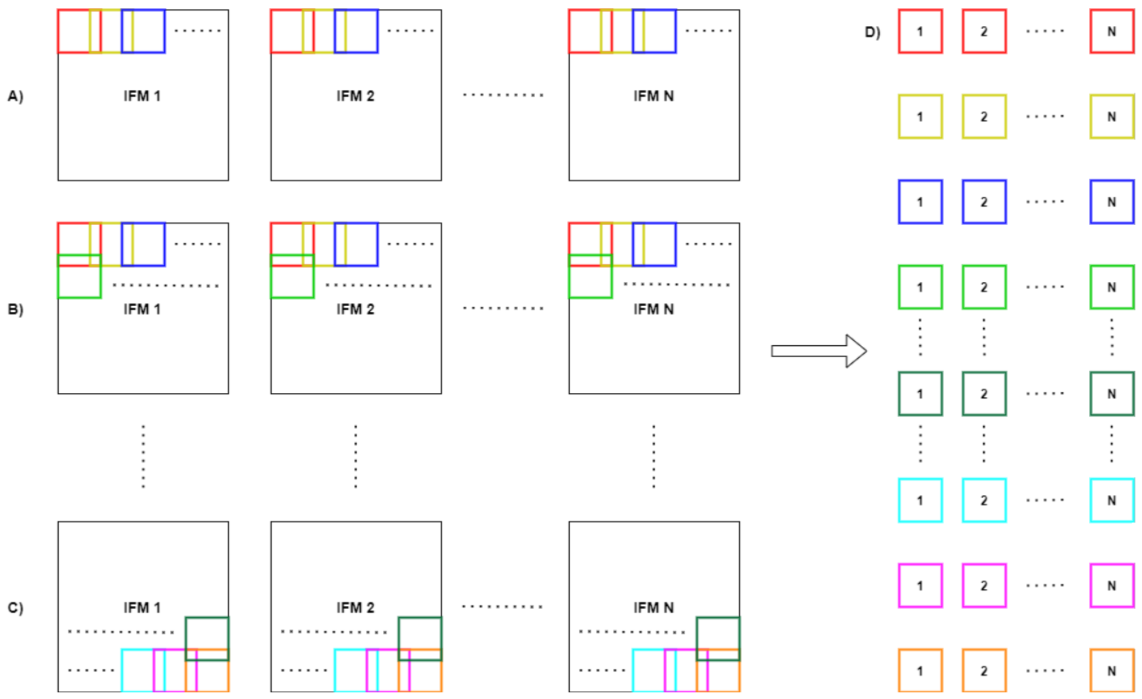


Figure 5.7: a) Tiling applied to the first 16 rows of the IFMs. b) Tiling applied to the next 16 rows of the IFMs. c) Tiling applied to the last 16 rows of the IFMs. D) Batches resulting from the tiling process, grouped by colours.

5.5. MobileNets hardware implementation

This section describes the standard and custom hardware blocks used in this project which combined with the software application compose the hardware/software heterogeneous system. Since the aim of this work was to implement the MobileNets inference process on a low-cost device, the main challenge was to allocate efficiently the resources available, namely the 270KB of BRAMs and the 80 DSPs, to reduce the number of data transfers and perform the maximum number of operations per clock cycle. As the pointwise layers represent about 94% of the MACs performed on the inference process, the hardware implementation was designed to prioritize the pointwise operations, allowing them to work continuously from the beginning until receiving the last IFM from the current stage. To keep the pointwise layer performing the MACs without interruption, the depthwise stage does not require to work continuously, there is only the need to produce new results right before the pointwise layer finishes the convolutions with the previous group of filters. Although the hardware resources to perform the MACs from each stage are independent, the BRAMs where the depthwise OFMs are stored are shared between both stages. To avoid memory collisions, this project uses a ping-pong local memory between these stages. Therefore, when the pointwise stage is reading from one of the shared IP memories, the depthwise stage can produce new OFMs and store them on the other shared IP memory. Other ping-pong IP memories are also used before the depthwise stage to store the IFMs, after the pointwise stage to store the OFMs, and to receive the pointwise weights. This allows the exchange of data between the DDR and the on-chip memory during the execution of the depthwise and pointwise stages.

The first custom IP is the “MobileNets Stream Connector”, designed to demultiplex the weights, Batch Norm parameters and IFMs which are stored on BRAMs. These BRAMs are within the “MobileNets Depthwise” and “MobileNets Pointwise” IPs (the other custom IPs) and are configured in simple dual-port mode with different port widths, depending on the data type. A simple scheme of the proposed MobileNets hardware design can be observed in figure 5.8.

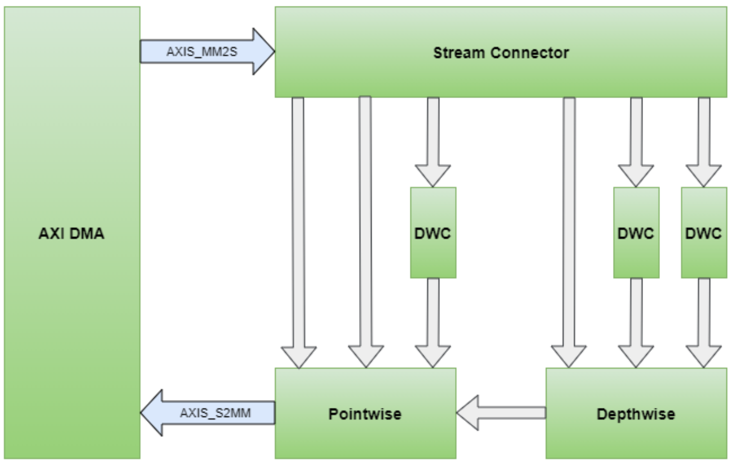


Figure 5.8: Simple scheme of the proposed MobileNets hardware design.

As shown in figure 5.8, the Stream Connector demultiplex all the data received from the DDR to the “AXI-Stream Data Width Converter” (DWC) or directly to the correct IP port. The pointwise module also receives the depthwise intermediate results and interacts with the DMA to send the final OFMs pixels to the DDR. The following sub-sections explain in detail the architecture and function of these three custom IPs, and all the interconnections used to form a unique system.

5.5.1. MobileNets Stream Connector

The “MobileNets Stream Connector” module represented in figure 5.9 is composed of a demultiplexer (demux) and a state machine. Each signal and data from the DMA are connected to the input of the demux, which is then forwarded to the correct output port that connects to an “AXI-Stream Data Width Converter” or directly to a “MobileNets Depthwise” or “MobileNets Pointwise” input port. This process is controlled by the state machine and depends on the valid and last signals from the DMA to maintain or change to the next state. When the state machine detects that the valid from the DMA is asserted, the desired output port is selected until the last signal is asserted. Thereafter, the state changes to the next. However, there are cases where the change of state is evaluated by an extra signal, provided by a counter. For instance, the IFMs from a single-stage cannot be sent at once from the DDR to the PL due to memory limitations, thus, they are divided and sent in groups. In this situation, a group of pixels is received and after the last signal is asserted, the state machine verifies the signal from the counter. If the counter signal is not asserted, this means that there are more IFMs from the current stage to be sent by the DMA, but it is necessary to wait for the current ones to be processed, thus, the state does not change and waits for a new group of images.

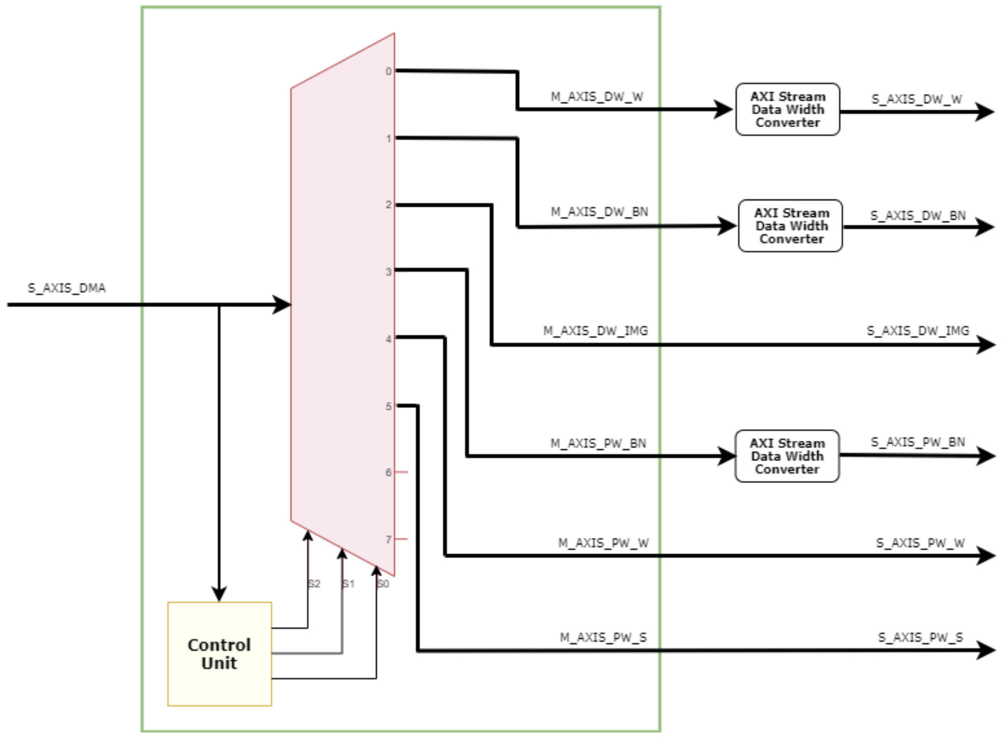


Figure 5.9: Stream Connector to demultiplex the input data to the different ports.

5.5.2. MobileNets Depthwise Stage

This module implements the depthwise convolution, Batch Norm and ReLU6. Before sending the OFMs to the pointwise module, the output data is specifically organized to allow an easy interface to the pointwise convolution, explained in more detail further ahead in this section. To keep data processing continuously, the depthwise stage IP sub-modules were designed to work with a pipeline structure, producing one result of the OFMs every clock cycle, after the initial delay. Since the depthwise kernels have a 3×3 size, the depthwise layer was implemented to perform 9 MACs in each clock cycle, and the Batch Norm and ReLU6 perform the required operations in the same conditions.

The distribution of the IP memory sizes to each type of data is such that all the depthwise weights and Batch Norm parameters are stored at once for each stage to reduce the number of communications, which is possible because these weights and parameters represent a small size compared to the IFMs and pointwise weights. For the IFMs, the memory should have a minimum size, able to store $16 \times 16 \times 32$ IFM pixels due to the tiling process described in section 5.4.

Figure 5.10 presents the block diagram of the depthwise module. The black lines represent the AXI-Stream bus with data, valid, last and ready signals.

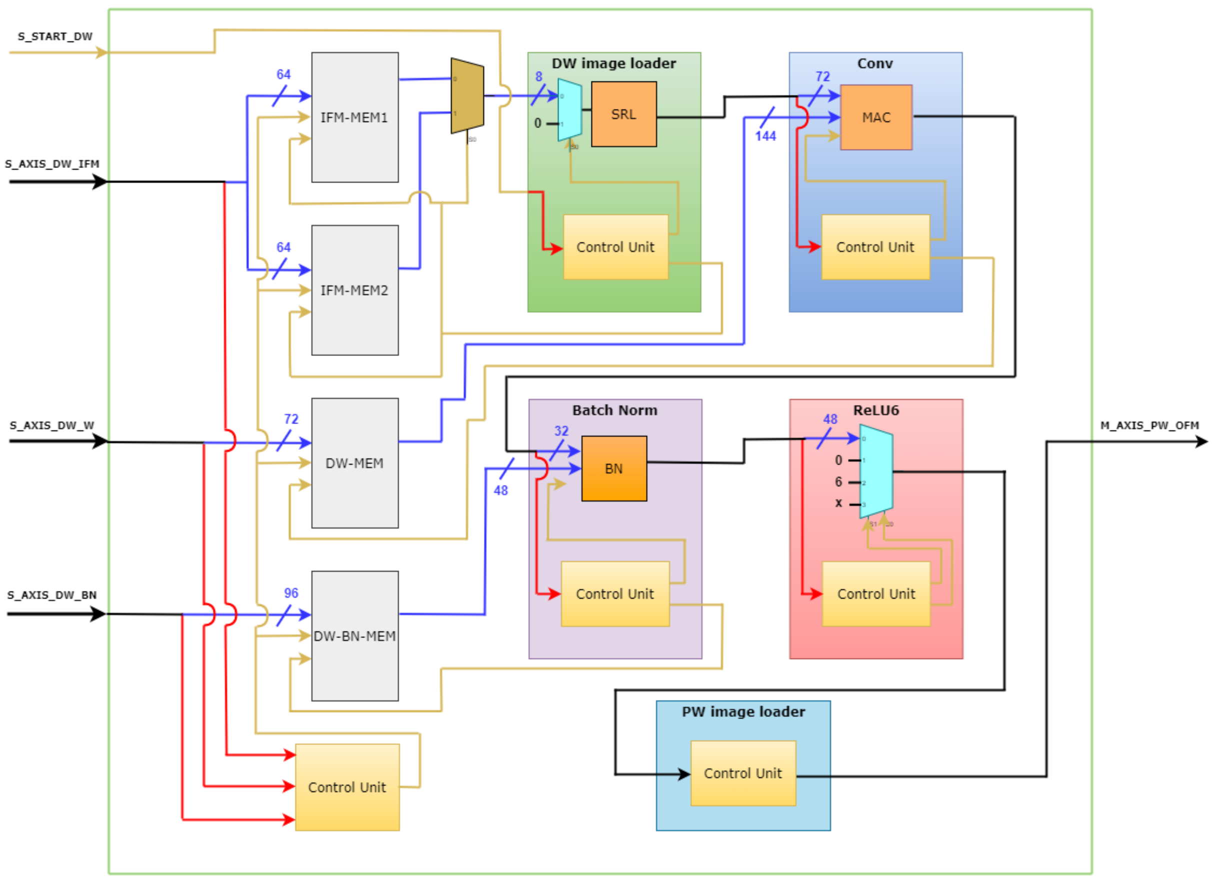


Figure 5.10: Depthwise convolution module.

Blue lines represent the buses used to load the BRAMs with the parameters. The red lines represent the input signals of the state machine, namely the valid and last signal, used to load the data and begin the process of a particular module. Lastly, the yellow lines are the output signals from the control unit to control the module and the readings from the BRAM memories.

IFM-MEM1 and IFM-MEM2 are formed each one by 2 BRAMs allowing to store a maximum of $16 \times 16 \times 32$ pixels at once. The input data port is 64 bits wide and the output 8 bits, this last one corresponding to the size of a single pixel. When the Conv sub-module is performing the depthwise convolution using a group of pixels from the IFMs stored in IFM-MEM1, IFM-MEM2 is receiving another group of pixels of the IFMs, and vice-versa, when the depthwise convolution is using the pixels stored in IFM-MEM2, IFM-MEM1 is receiving the next group of pixels. DW-MEM is formed by 4 BRAMs which allows storing all the depthwise weights in each stage. The input data port is 72 bits wide and the output 144 bits. This way, a single read of the DW-MEM, outputs 9 weights of 16 bits $(9 \times 16 = 144)$. DW-BN-MEM uses 1.5 BRAMs and stores the depthwise Batch Norm parameters in each stage. The input ports are 96 bits wide and the output 48 bits, which allow to output 3 Batch Norm parameters of 16 bits each $(3 \times 16 = 48)$ bits, with a single read. The depthwise module execution begins by receiving the depthwise weights and Batch Norm parameters and stores them on the DW-MEM and DW-BN-MEM respectively. After that, the DMA starts sending the IFMs pixels, as needed, in several streams. When the IFM-MEM1 (or IFM-MEM2) loading process is complete, the last signal from the stream is asserted and detected by the control unit from the depthwise convolution module, which informs the DW image loader that this sub-module can start the input of the pixels. At the same moment, IFM-MEM2 (or IFM-MEM1) is receiving another group of IFMs pixels, that will be ready to use when the depthwise stage process ends, and so on. The S_START_DW is also used to inform the DW image loader to start the input of the pixels. This process is controlled by the pointwise module and happens in each stage after finishing the first pointwise convolution.

DW image loader sub-module

Figure 5.11 shows the initial step of a 3×3 kernel convolving with a 7×7 IFM.

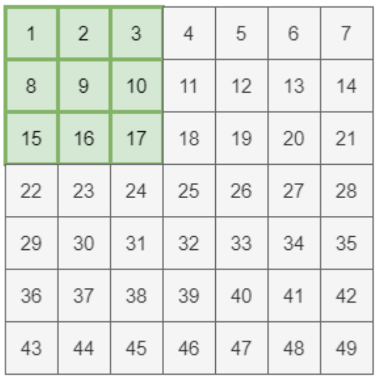


Figure 5.11: A 3×3 kernel convolving with a 7×7 IFM (2D perspective).

The pixels from the IFM can be aligned in one dimension as presented in figure 5.12 A), where the MAC of the pixels with the kernel (shown in green) produces the first result of the convolution.

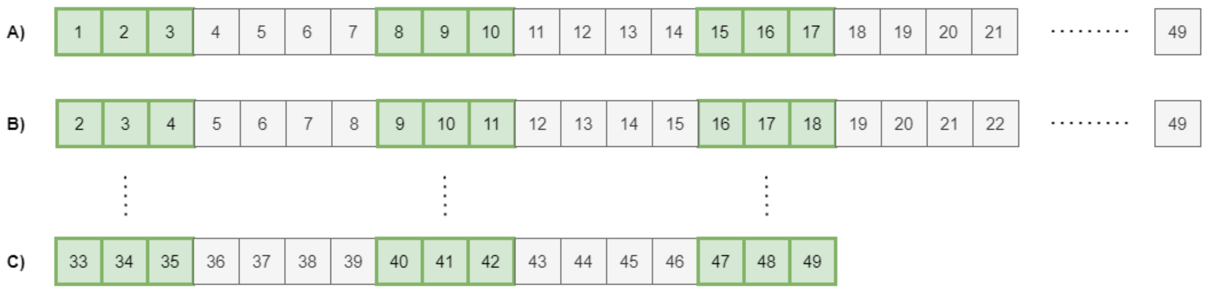

A) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 49
B) 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 49
C) 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49

Figure 5.12: A 3×3 kernel convolving with a 7×7 IFM (1D perspective).

Sliding by one unit the kernel to the right (figure 5.11) the second result of the convolution is obtained, which is equal to what happens in figure 5.12 B), where the pixels shift to the left once. Going further with the convolution process, the last result is presented in figure 5.12 C), having the pixels shifted 25 times to the left. Thus, fixing the kernel to certain registers, a Shift Register (SR) can be used to perform the convolution of an image with a 3×3 kernel. Moreover, there is only the need to store a constant number of 17 pixels, which corresponds to the size of 2 IFM rows plus 3.

The depthwise convolution can be performed using an SR. An SR can be efficiently implemented on an FPGA using LUTs, which is commonly referred to as an SRL. Figure 5.13 shows the SRL used in this project, represented in 2D for easier observation. To save resources, the proposed SRL is reconfigurable, allowing to perform convolutions of different IFMs sizes or strides. Since the maximum tiled block from the IFMs is 32 sets of pixels of size 16×16, the SRL is composed of a total of 35 registers $(16 + 16 + 3)$.

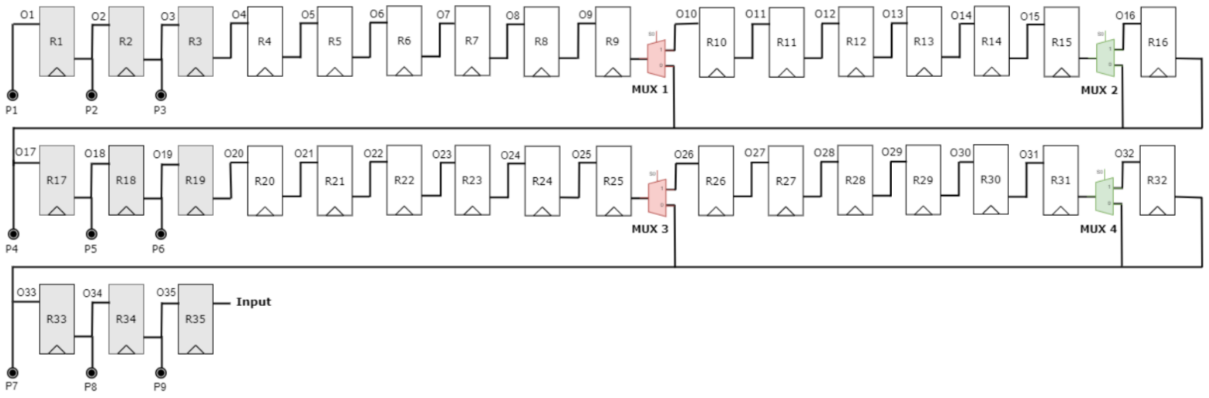


Figure 5.13: Reconfigurable SRL.

The grey registers correspond to the pixels that are going to be convolved with the kernel, thus, P1 to P9 are connected to the Conv sub-module. Four multiplexers (mux) control how the data flows through the SRL. As an example, on stage 13, the 7×7 pixels from a single IFM plus a complete zero-padding resizes the IFM to 9×9 pixels. To perform this convolution, the control unit selects in mux 1 the output from register 17 instead of register 10, and in mux 3 the output from register 33 instead of register 26. After indicating the desired configuration to the SRL, the control unit starts reading the pixels of the IFMs from IFM-MEM1 or IFM-MEM2 and stores them on register 35, one by one. The output of the SRL is then propagated to the Conv sub-module.

<u>Conv sub-module</u>

This sub-module is composed of a MAC and a control unit. The MAC architecture of the Conv sub-module is pipelined, as shown in figure 5.14, and executes the MAC of 9 pixels with 9 weights with a latency of 6 clock cycles. The 9 multipliers (shown in blue) receive the pixels from the SRL stored in the grey registers P1 to P9, and the 9 depthwise weights stored on the grey registers W1 to W9. These 9 weights, corresponding to one kernel, are loaded by the control unit from DW-MEM in a single reading. After that, the multiplication results are then summed and accumulated using the intermediate grey registers and the adders (shown in red). To control the process, the green registers indicate what are the valid results and the yellow registers what is the last result of the convolution process. These two signals together with the pixel generated (pixel out) are then propagated to the Batch Norm sub-module.

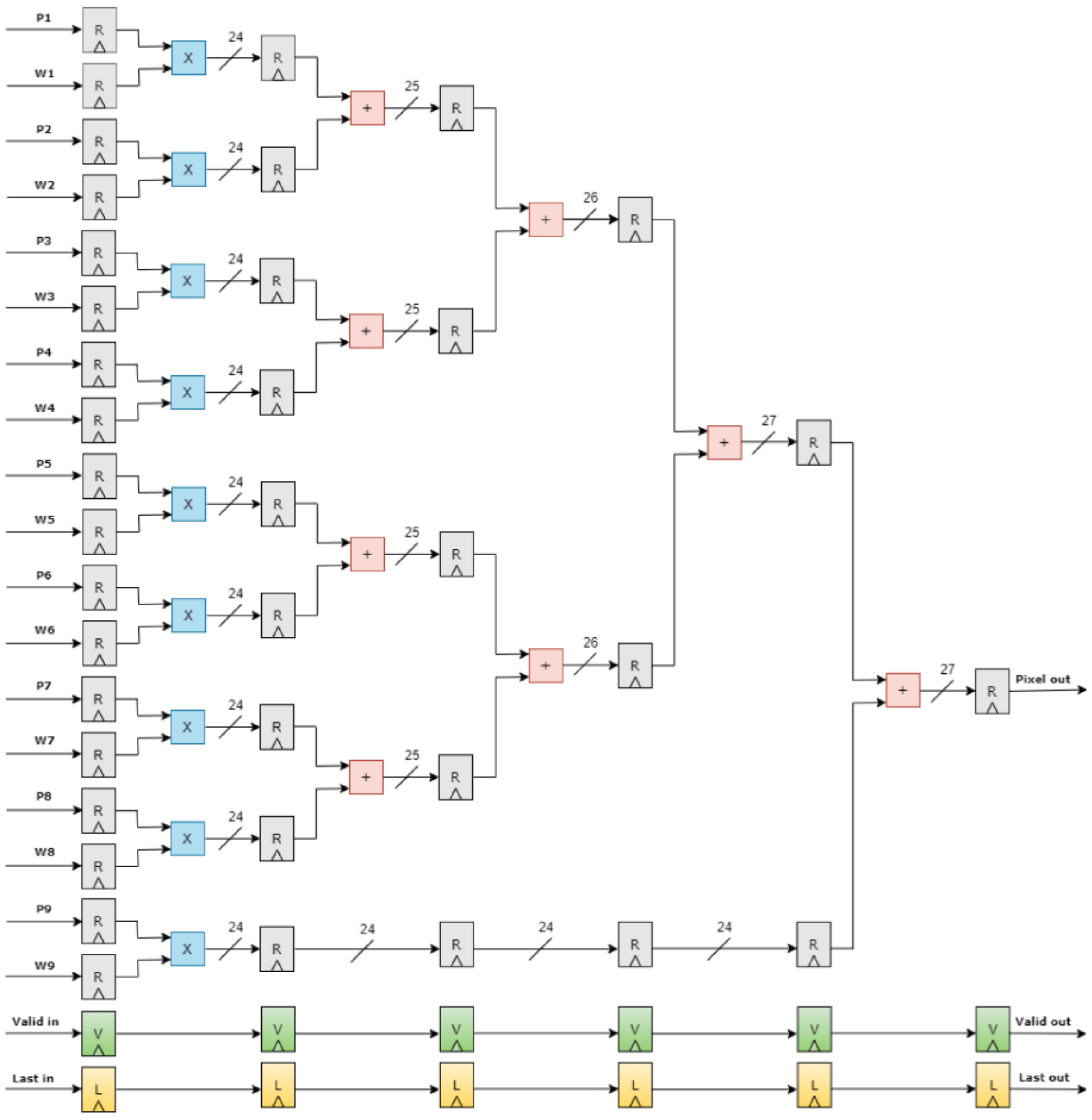


Figure 5.14: MAC architecture of the depthwise convolution module.

<u>Batch Norm sub-module</u>

The Batch Norm sub-module is composed of a control unit and a BN sub-module. This BN sub-module performs all the arithmetic involved in the Batch Norm function and the respective architecture is presented in figure 5.15. Before performing the add operation or sending the pixel out to the ReLU6 sub-module, the number is aligned, using the shift right operation (see section 2.5.4). The first three grey registers store the values used for these shifts and they change every stage. The fourth grey register stores the pixel that comes from the Conv sub-module and the last three grey registers store the Batch Norm parameters from the DW-BN-MEM. The valid and last signals on the green and yellow registers are propagated to the ReLU 6 sub-module.

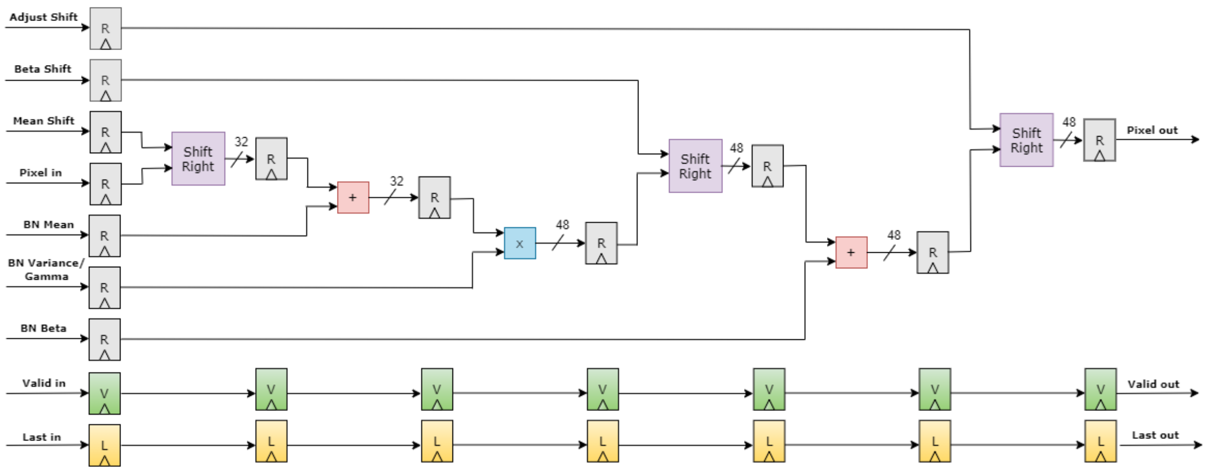


Figure 5.15: BN sub-module architecture.

ReLU6 sub-module

The ReLU6 is the last sub-module used to perform the depthwise convolution and the one with the simplest implementation. It tests if the number is smaller than zero; greater than zero and smaller than six; or greater than six, and selects the output of the mux according to the evaluation result (see section 2.5.4). Similar to the previous two sub-modules, ReLU6 has a line of valid and last registers that are used to propagate the valid and last signal to the next sub-module.

PW image loader sub-module

The PW image loader is responsible for organizing the OFM pixels before sending them to the pointwise module. Tiling the IFMs in height, width and depth, allows the depthwise stage OFMs to be stored on BRAMs before beginning the pointwise stage, instead of sending them to the DDR, reducing the number of communications between the PL and PS. However, while the depthwise convolution is performed in height and width, the pointwise convolution is performed in-depth. This requires grouping the pixels with the same index of the different depthwise OFMs while storing them on the IP memory. The control unit from the PW image loader executes this process by identifying the index from the valid input pixel and what OFM does it belong to. After that, an address is assigned to this pixel and the information is immediately sent to the pointwise convolution module, to store the pixel in the correct position of the IP memory. The process is repeated until the last pixel of the OFMs is sent to the pointwise module. This procedure makes the reading process of these pixels easier to perform when processing the pointwise convolution, since a single read allows to output the 32 pixels of the same index, from the different depthwise OFMs. Figure 5.16 presents an example of the described sorting process applied to 32 OFMs of size 7×7.

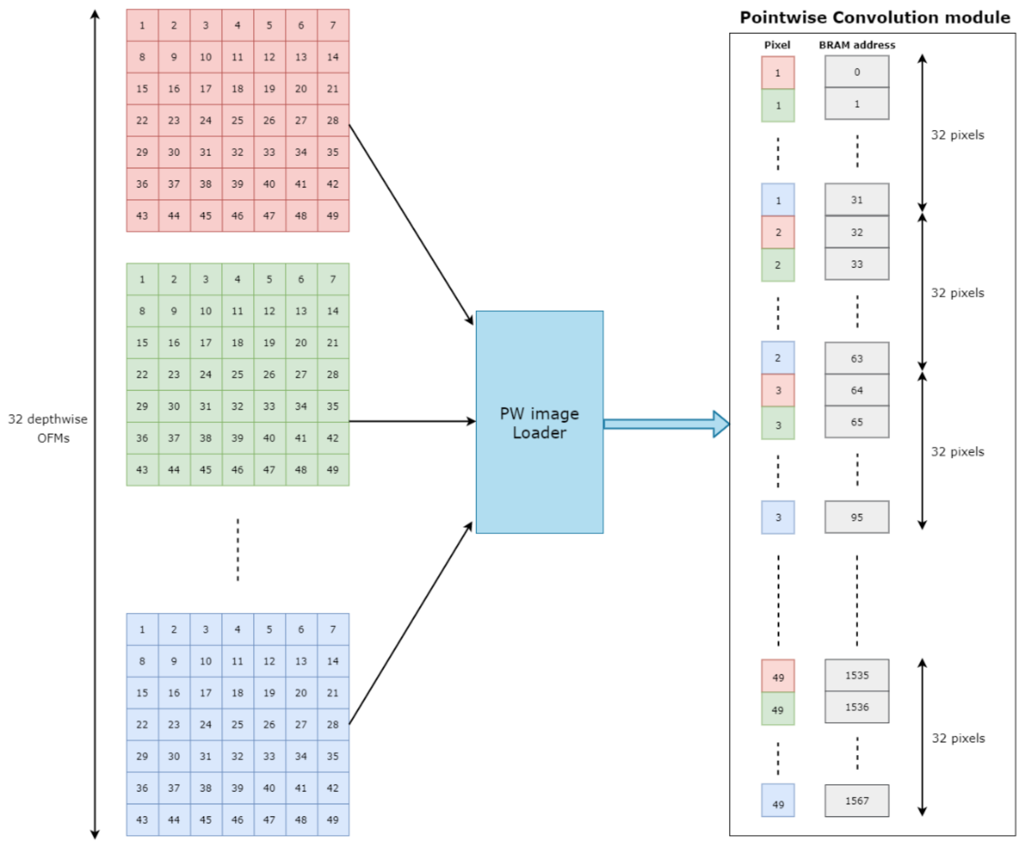


Figure 5.16: Sorting process of PW image loader.

All the pixels from the OFMs with the same indexes are grouped and stored in consecutive IP memory addresses. Therefore, the 32 pixels with index 1 are stored from address 0 to 31, the next 32 pixels with index 2 are stored from address 32 to 63, and so on.

Table 5.2 shows the number and percentage of the most important resources used by the MobileNets depthwise sub-modules. It is possible to observe that the resources used are about 5% for the LUTs and FFs, and about 15% for the BRAMs and DSPs, which is significantly low compared to the resources available.

Table 5.2: LUTs, FFs, BRAMs and DSPs used by the depthwise module.

Component	LUT	LUT (%)	FF	FF (%)	BRAM	BRAM (%)	DSP	DSP (%)
IFM-MEM1	8	0.1	1	0.0	2.0	3.3	0	0.0
IFM-MEM2	9	0.1	1	0.0	2.0	3.3	0	0.0
DW-MEM	0	0.0	0	0.0	4.0	6.7	0	0.0
DW-BN-MEM	0	0.0	0	0.0	1.5	2.5	0	0.0
DW Image Loader	224	1.3	336	1.0	0.0	0.0	0	0.0
Conv	230	1.3	353	1.0	0.0	0.0	9	11.3
Batch Norm	273	1.6	287	0.8	0.0	0.0	2	2.5
ReLU6	73	0.4	63	0.2	0.0	0.0	0	0.0
PW image loader	65	0.4	120	0.3	0.0	0.0	0	0.0
Others	140	0.8	178	0.5	0.0	0.0	0	0.0
Total	1,022	5.8	1,339	3.8	9.5	15.8	11	13.8

5.5.3. MobileNets Pointwise Stage

The last custom IP designed in this project performs the pointwise convolution, Batch Norm and ReLU6 functions from the pointwise stage. As the depthwise stage IP, this module was designed to work with a pipeline structure, however, in this case, the IP produces not one but two results every clock cycle, after the initial delay. The pointwise convolution was implemented to perform 32 MACs of 2 different filters each clock cycle, combined with 2 Batch Norm and 2 ReLU6 modules, to process both complementary operations separately.

The distribution of the IP memory sizes to each type of data was made such that all the Batch Norm parameters are stored at once for each stage, and the pointwise parameters are stored depending on the layer and on the need of the pointwise convolution, since in some layers, it is impossible to store all the pointwise parameters on the BRAMs. The module is designed to implement the accumulation of 32 filters at a time (i.e. before sending a group OFMs pixels to the DDR) when the layer stride is 1, or the accumulation of 128 filters at a time when the layer stride is 2. These are the values that maximize the use of resources of the device. The number of OFMs depend on the stride of the layer. When the stride is 1, the IMFs of the pointwise convolution have $14 \times 14 \times 32$ pixels size, leading to an output of the same size when performing the convolution with 32 filters. However, when the stride is 2, the IMFs of the pointwise convolution have $7 \times 7 \times 32$ pixels size, thus four times smaller, giving the possibility to use four times more filters to maximize the output. Therefore, to perform the pointwise convolution using this hardware implementation, the pointwise filters and Batch Norm parameters need to be reorganized to allow the accumulation of 32 filters at a time when the stride is 1, and the accumulation of 128 filters at a time when the stride is 2. This reorganization process can be applied to the MobileNets model before the inference execution.

Figure 5.17 presents the block diagram of the pointwise module. As in the depthwise module, the black lines represent the AXI-Stream bus with data, valid, last and ready signals. Blue lines represent the buses used to load the BRAMs with the parameters. The red lines represent the input signals of the state machine, namely the valid and last signal, used to load the data and begin the process of a particular module. Lastly, the yellow lines are the output signals from the state machine to control the module and the readings from the BRAMs. IFM-MEM3 and IFM-MEM4 are formed each one by 8 BRAMs allowing to store a maximum of $14 \times 14 \times 32$ pixels at once, with an input data port 16 bits wide and an output data port of 512 bits. These two BRAMs are connected to work as a ping-pong memory and each read of one of these BRAMs outputs 32 pixels of 16 bits $(32 \times 16 = 512)$. When the Conv sub-module is performing the pointwise convolution using a group of pixels from the IFMs stored in IFM-MEM3, IFM-MEM4 can receive another group of pixels of the IFMs, and vice-versa, when the depthwise convolution is using the pixels stored in IFM-MEM4, IFM-MEM3 is receiving the next group of pixels. These pixels come from the depthwise module and do not require any writing control of the control unit, since the information that comes from the depthwise module contains the pixel, destination address and the other control signals.

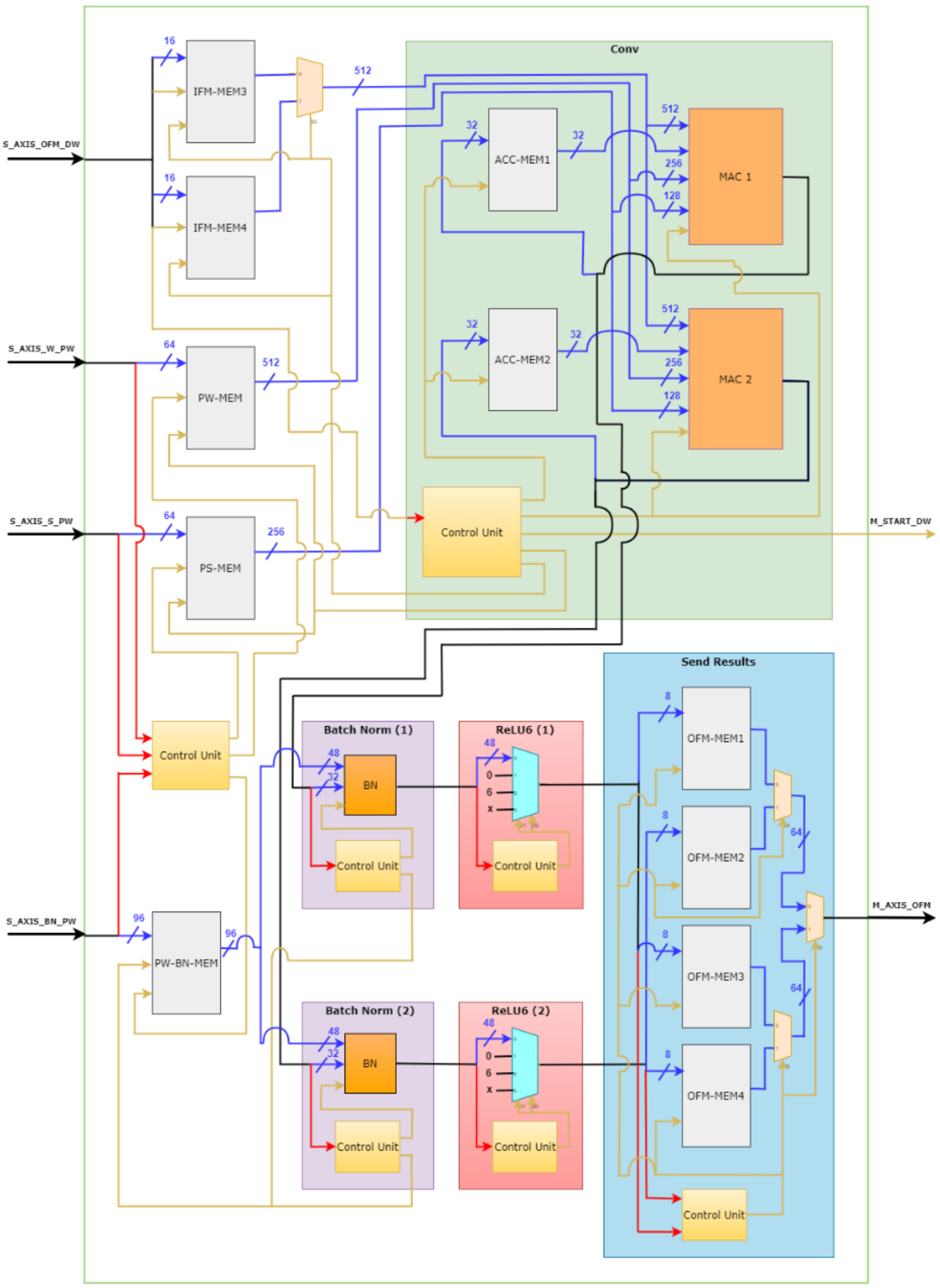


Figure 5.17: Pointwise convolution module.

PW-MEM is formed by 7.5 BRAMs allowing to store a maximum of 32,768 pointwise values in each stage. The input data port is 64 bits wide and the output 256 bits. This way, a single read of the PW-MEM, outputs 64 values of 8 bits $(64 \times 8 = 512)$. This unit can work as a ping-pong memory depending on the layer. When the number of values is fewer than 32,768, all the parameters can be stored in a single transfer on the 4,096 input addresses. On the other hand, when the number of values is higher than 32,768, the parameters are sent in blocks of 16,384. This allows PW-MEM to receive data on half the addresses while the pointwise convolution is reading other data from the other half. The choice of using a single memory instead of two memories (as IFM-MEM3 and IFM-MEM4) for this ping-pong buffer was because of two reasons: first, the amount of pointwise values in each transfer to on-chip memories, defines the number of filters that is possible to process at that moment, thus the number of filters can be sent gradually. However, even to perform the convolution of a single filter, there is always the need to have $14 \times 14 \times 32$ pixels (or $7 \times 7 \times 32$ depending on the layer) available. Therefore, the priority was on defining all the other memory sections and using the remaining to define the PW-MEM memory size. The second reason is that using an IP memory with the same configuration as PW-MEM but with only half of the addresses, would lead to the same 7.5 BRAMs, which in practice would require twice the BRAMs than the actual configuration. PS-MEM is implemented exactly as PW-MEM and receives the pointwise scale factors (see section 4.3.2).

The last BRAMs that receive data from an AXI-Stream interface is PW-BN-MEM, which uses 1.5 BRAMs and stores the pointwise Batch Norm parameters in each stage. Both input and output ports are 96 bits wide, which allow to output 6 Batch Norm parameters of 16 bits each $(6 \times 16 = 96)$ bits, with a single read.

Conv sub-module

This sub-module is composed of 2 MACs, 2 IP memories and a control unit, allowing the convolution of two pointwise convolutions simultaneously. Figure 5.18 represents the MAC architecture from the Conv sub-module where one pointwise convolution is performed, using a pipeline architecture that allows the MAC of 32 pixels with 32 weights with a latency of 8 clock cycles. To more clearly understand the description of the module, the image is divided into 8 groups. The first group is composed of 64 registers (represented by grey) that store the pixels P1 to P31 read from IFM-MEM3 or IFM-MEM4, and the pointwise values W1 to W32 read from PW-MEM. There are also 32 multipliers (represented by blue) allowing the multiplication of the input data simultaneously. The multiplications results are stored in the 32 registers of group 2, and after this, these results are shifted according to the scale factor used to quantize the respective weight. These scale factors are read from PS-MEM. Group 3 is composed also by 32 registers that store the shifted values, and by 16 adders, that begin the accumulation of the multiplication results. From here, the number of registers and adders drops by half in each group, until group 8. In group 8 there is one register and one adder. This adder sums the intermediate result from the MAC process with the intermediate value stored on the ACC-MEM1 or ACC-MEM2 and saves it on the same address. This address is propagated to ACC-MEM1 or ACC-MEM2, and the valid and last signals to the Batch Norm sub-modules.

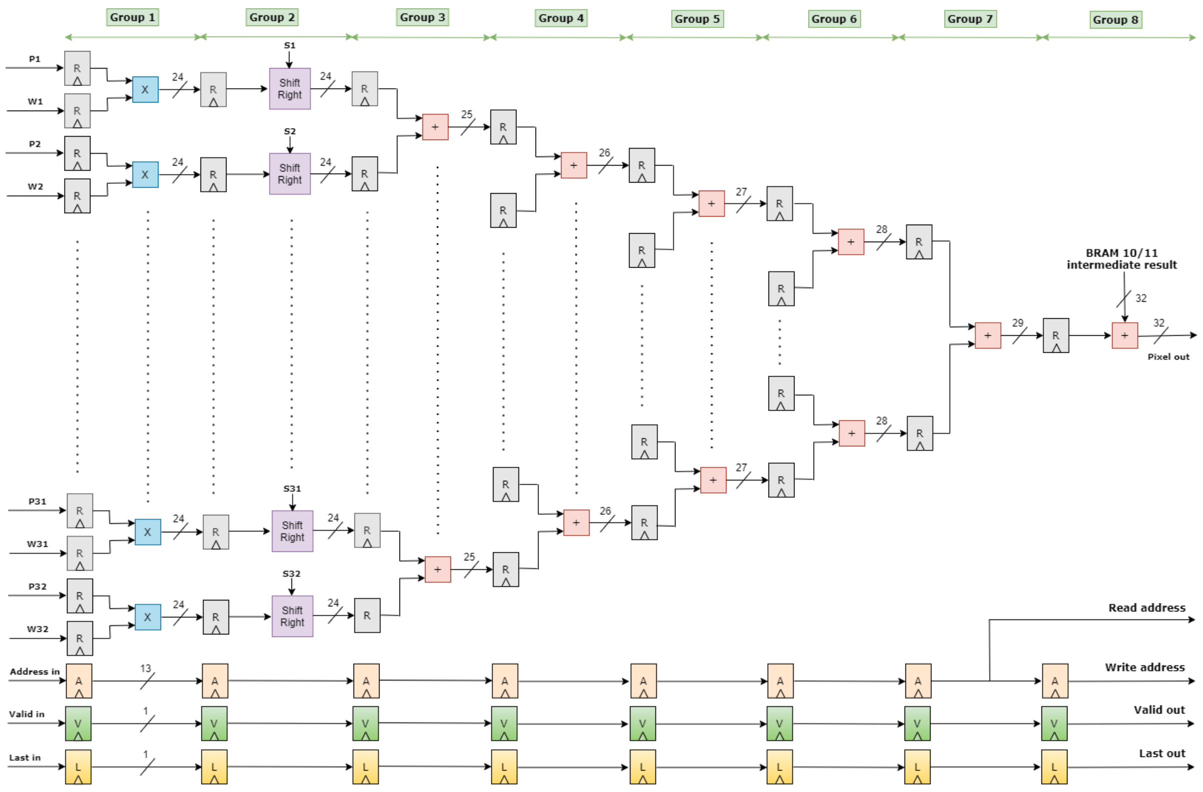


Figure 5.18: MAC architecture of the pointwise convolution module.

The option of having 2 MACs sub-modules allows performing two convolutions at the same time. However, the convolutions have to be processed in the following way: MAC 1 uses the first half of the filters and MAC 2 the second half. Therefore, ACC-MEM1 will accumulate the intermediate results from the first half of the OFMs, and ACC-MEM2 the intermediate results from the second half. To store these intermediate results, both ACC-MEM1 and ACC-MEM2 are formed by 3.5 BRAMs and have an input and output port of 32 bits wide. This way, each of these IP memories can store up to 3,136 results of 32 bits. The M_START_DW signal is used to start the depthwise convolution when the pointwise convolution finishes. This happens in each stage after the first pointwise convolution.

Batch Norm and ReLU6 sub-modules

These two sub-modules are implemented as in the depthwise convolution stage. Batch Norm (1) and ReLU6 (1) process the intermediate values stored in ACC-MEM1, while the other two sub-modules process the intermediate values stored in ACC-MEM2. Having two groups of these modules allows keeping the generation of two results at a time. The values are then propagated to the Send Results sub-module.

Send Results

This last sub-module precedes the transfer of the OFMs pixels to the DDR and is composed of 4 IP memories and a control unit. OFM-MEM1, OFM-MEM2, OFM-MEM3 and OFM-MEM4 are formed, each one, by 1 BRAM and have an input port of 8 bits wide and an output port of 64 bits wide, allowing each memory to store a maximum of 3,136 pixels of 8 bits. The function of this unit is to store the OFMs pixels before sending them to the DDR. The transfer can begin when the group of 32 or 128 filters (depending on the stride) finishes the convolution with the IFMs pixels and the pointwise sub-module starts the convolution with a new group of filters. This way, the OFMs pixels are sent while the pointwise convolution is producing new results, hiding the latency behind the communication. To perform this operation, the memories work as two ping-pong buffer memories. OFM-MEM1 or OFM-MEM3 stores the data from ReLU6 (1) and OFM-MEM2 or OFM-MEM4 the data from ReLU6 (2). The reason to have the memory system implemented this way is that the OFMs pixels are not sent immediately when the other pointwise starts. When the system is performing a pointwise convolution, this execution time is used to exchange data between the PS and the PL, namely IFMs and pointwise weights. Therefore, the Send Results sub-module can only start to transfer data when the other transfers end. However, during this wait, other OFMs pixels may need to be stored in this sub-module, and to avoid the corruption of the data, these new pixels are stored in the other IP memory. Besides organizing the OFMs pixels and synchronizing the transfers, the Send Results sub-module also allows sending 8 pixels at a time to the DDR, since the output data port of the IP memories is 64 bits wide.

Table 5.3 presents the number and percentage of the most important resources used by the MobileNets pointwise sub-modules. Comparing with the depthwise module, the pointwise implementation uses roughly 5 times more BRAMs and 6 times more DSPs.

Table 5.3: LUTs, FFs, BRAMs and DSPs used by the pointwise module.

Component	LUT	LUT (%)	FF	FF (%)	BRAM	BRAM (%)	DSP	DSP (%)
IFM-MEM3	0	0.0	0	0.0	8.0	13.3	0	0.0
IFM-MEM4	0	0.0	0	0.0	8.0	13.3	0	0.0
PW-BN-MEM	0	0.0	0	0.0	1.5	2.5	0	0.0
PW-MEM	0	0.0	0	0.0	7.5	12.5	0	0.0
PS_MEM	0	0.0	0	0.0	7.5	12.5	0	0.0
Conv	6.920	39.3	4,816	13.7	7.0	11.7	64	80.0
Batch Norm (1)	288	1.6	293	0.8	0.0	0.0	2	2.5
Batch Norm (2)	258	1.5	226	0.6	0.0	0.0	2	2.5
ReLU6 (1)	78	0.4	48	0.1	0.0	0.0	0	0.0
ReLU6 (2)	75	0.4	46	0.1	0.0	0.0	0	0.0
PW Send	120	0.7	63	0.2	4.0	6.7	0	0.0
Others	145	0.8	270	0.8	0.0	0.0	0	0.0
Total	7,884	44.8	5,498	15.5	43.5	72.5	68	85.0

5.6. Conclusion

This chapter begins by introducing the Zynq-7000 SoC FPGA and presents the main features and the resources available of the device. The BRAMs and DSPs are the crucial components for this project because as more are used, more data can be stored on-chip and more MACs can be performed in each clock cycle. Another aspect highlighted is the AXI interfaces available on the SoC

FPGA to exchange the data between the PS and the PL. For this project, since there are constantly data transfers to be performed, the AXI-Stream protocol is the one that better suits this purpose.

After evaluating the resources available for the project, the chapter describes the functions assigned to the software application and the ones that are implemented in hardware. Taking into account the execution time, the depthwise and pointwise stages are selected to be performed on the PL, and two custom IPs are designed and presented to execute each function. There is also presented another custom IP to multiplex the data to different ports of depthwise and pointwise IP.

Due to memory limitations and to reduce the communications between the PS and the PL, tilling is applied to the first six depthwise stages, making it possible to perform both depthwise and pointwise stages without the need to store intermediate data (pixels) on the DDR. Also, the implementation of ping-pong buffers hides the latency of most of the communications.

Chapter 6

SoC-FPGA MobileNets Results Analysis

This chapter describes the complete MobileNets accelerator architecture developed in this project and evaluates the respective performance compared to the software implementation. The first section shows the complete architecture of the accelerator as well as the hardware resources used. Also, an algorithm is presented and described to understand how the inference process is conducted by the system. The following section presents the experimental results where the communication and computation are explained, and some examples applied to certain layers. Lastly, the chapter ends with a comparison of the execution times for the software and hardware/software implementations and the respective speed up.

6.1. Hardware/Software architecture

Figure 6.1 presents the 11 blocks that compose the base hardware of the MobileNets accelerator. All the blocks were described in the previous chapter, except for the Processor System Reset, the AXI Interconnect and the AXI SmartConnect, which are automatically added, connected and configured by the development environment.

The resource utilization of all the main components used on the MobileNets accelerator is presented in table 6.1. One of the aims of this project was to design the accelerator taking full advantage of the available BRAMs and DSPs because these are crucial to perform the MACs. Observing the values, this was successfully achieved since almost all of these two components were used in this project.

Table 6.1: Distribution of the resources used in the MobileNets accelerator.

Module	LUT	LUT (%)	FF	FF (%)	BRAM	BRAM (%)	DSP	DSP (%)
DMA	1,599	9.1	2,316	6.6	3.0	5.0	0	0
AXI SmartConnect	2,238	12.7	3,053	8.7	0.0	0.0	0	0
Width converter	977	5.6	2,458	7.0	0.0	0.0	0	0
Stream Connector	426	2.4	574	1.6	0.0	0.0	0	0
Depthwise stage	1,022	5.8	1,339	3.8	9.5	15.8	11	13.8
Pointwise stage	7,884	44.8	5,762	16.4	43.5	72.5	68	85.0
Others	405	2.3	481	1.4	0.0	0.0	0	0
Total	14,551	82.7	15,983	45.5	56.0	93.3	79	98.8

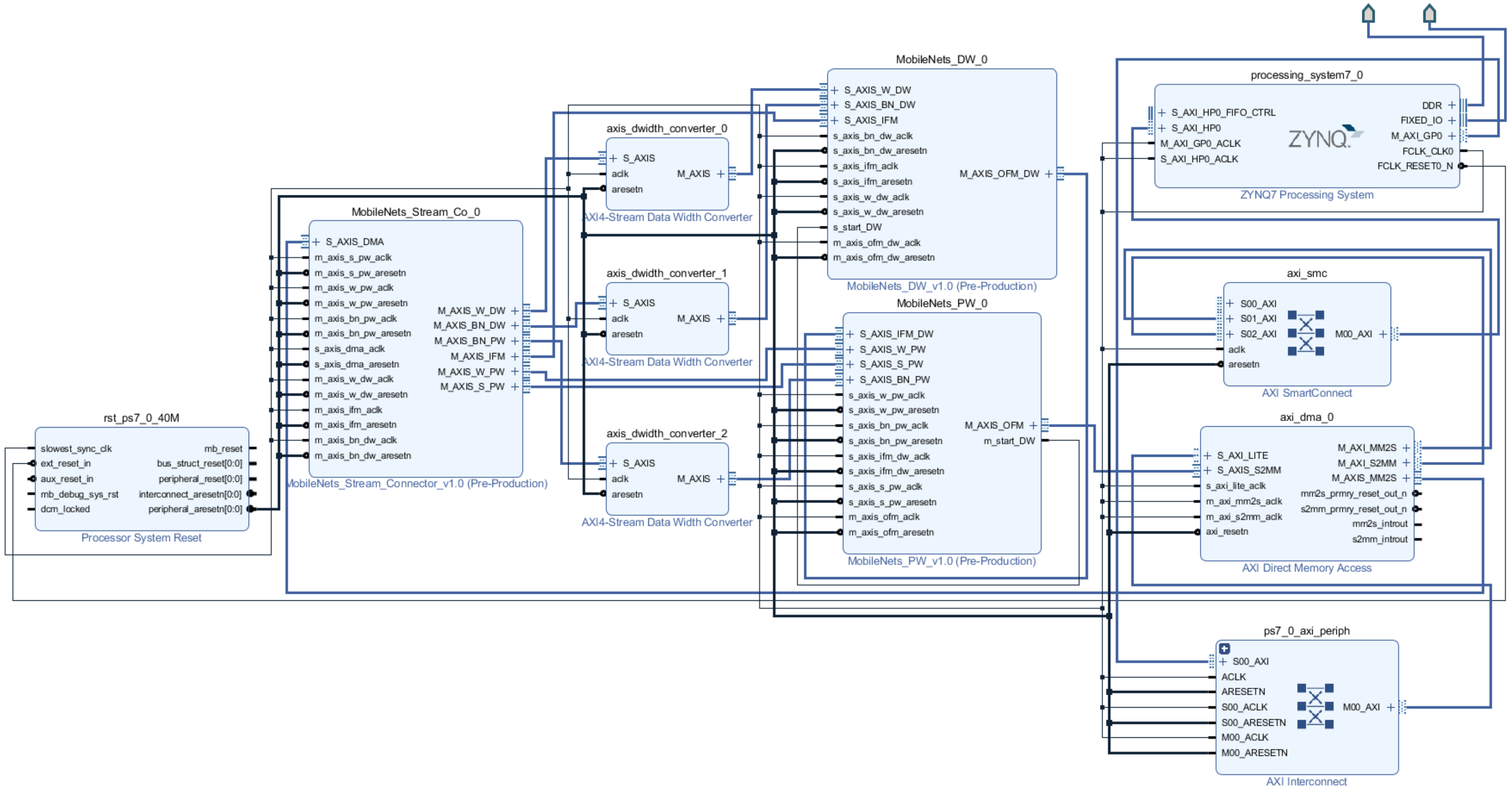


Figure 6.1: Block diagram of PS+PL system.

This system uses only one of the two ARM processors available, which is used to execute the software application. The MobileNets model and the input image are stored in the DDR memory and can be accessed by the ARM, and by the hardware blocks through the DMA. Algorithm 5 presents the pseudo-code for the hardware/software inference process. The ARM starts by processing the 3D stage which includes the normalization of the input image, half zero-padding, 3D convolution, Batch Norm and ReLU6.

Algorithm 5 – Software + Hardware inference process of MobileNets.

```
Input: Image $I$ in RGB format
Output: Values $S$ of the activation function softmax

$I_N \leftarrow Normalize(I)$
$C \leftarrow Half\_Zero\_Padding(I_N)$
$C_{3D} \leftarrow 3D\_Convolution(C)$
$B \leftarrow BatchNorm(C_{3D})$
$R \leftarrow ReLU(B)$

//Stages 1 to 4
$I \leftarrow Complete\_zero\_padding(R)$   // Stages 1 and 3
$I \leftarrow Half\_zero\_padding(R)$   // Stages 2 and 4
$I_n \leftarrow Tiling(I)$
$Send\_DW\_weights(W_D)$
$Send\_DW\_BN(BN_D)$
$Send\_PW\_BN(BN_P)$
$Send\_PW\_weights(W_P)$
$Send\_PW\_shifts(S_P)$
$Send\_IFM(O_n)$
for $iter = 1$ to $Iterations$ do   // IFM Iterations $= N_{IFM} \times S_{IFM} - 1$
    $Send\_IFM(O_n)$
    // $I \leftarrow$ performs depthwise and pointwise stages
    $Receive\_OFM(O_n)$
end for
$Receive\_OFM(O_n)$
$Reorganize\_pixel\_images(O)$

//Stages 5 to 6
$I \leftarrow Complete\_zero\_padding(O)$ // Stage 5
$I \leftarrow Half\_zero\_padding(O)$ // Stage 6
$I_n \leftarrow Tiling(I)$
$Send\_DW\_weights(W_D)$
$Send\_DW\_BN(BN_D)$
$Send\_PW\_BN(BN_P)$
$Send\_PW\_weights(W_P)$
$Send\_PW\_shifts(S_P)$
$Send\_IFM(I_n)$
for $iter = 1$ to $Iterations$ do   // IFM Iterations $= N_{IFM} \times S_{IFM} - 1$
    If $send_{PW} = TRUE$ do  // If a new group of pointwise parameters is required.
        $Send\_PW\_weights(S_P)$
        $Send\_PW\_shifts(S_P)$
    end if
    $Send\_IFM(I_n)$
    // $I \leftarrow$ performs depthwise and pointwise stage
    $Receive\_OFM(O_n)$
end for
```

```
$Receive\_OFM(O_n)$
$Reorganize\_pixel\_images(O)$

//Stages 7 to 13
$Send\_DW\_weights(\ W_D)$
$Send\_DW\_BN(\ BN_D)$
$Send\_PW\_BN(\ BN_P)$
$Send\_PW\_weights(\ W_P)$
$Send\_PW\_shifts(\ S_P)$
$Send\_IFM\ (I_n)$
for $iter = 1$ to $Iterations$ do    // IFM Iterations $= N_{IFM} \times S_{IFM} - 1$
    If $send_{PW} = TRUE$ do  //If a new group of pointwise parameters is required.
        $Send\_PW\_weights(\ S_P)$
        $Send\_PW\_shifts(\ S_P)$
    end if
    $Send\_IFM\ (I_n)$
    // $I \leftarrow$ performs padding + depthwise and pointwise stage
    $Receive\_OFM(O_n)$
end for
$Receive\_OFM(O_n)$

$P \leftarrow Pooling(O_n)$
$FC \leftarrow Fully\_Connected(P)$
$S \leftarrow Softmax(FC)$
```

Then, from stages 1 to 4, the software application performs the padding and tiling and sends to the PL the different types of weights and Batch Norm parameters required to process the current stage. The depthwise stage begins when the first block of IFMs pixels is sent ($Send\_IFM(O_n)$), and while these pixels are being processed, the software application sends another group of IFMs pixels. This way, the depthwise module have always IFMs pixels ready to be processed, which, consequently, keeps the pointwise module working without interruption. When the second group of IMFs is sent, the algorithm waits for the reception of a group of OFMs pixels or a synchronization signal ($Receive\_OFM(O_n)$) before sending new IMFs pixels. This process repeats $N_{IFM} \times S_{IFM} - 1$ times, where N_{IFM} represents the number of tiled blocks and S_{IFM} the number of times each of these blocks is sent. On the other hand, the pointwise convolution is performed in-depth, thus, except for the first stage, the generation of a group of OFM pixels require two or more group of IFM pixels. Therefore, when the OFM pixels are not generated yet, the synchronization signal is used to inform the software application to send more IFM pixels. Performing the iteration this way, the system is synchronized to have always a reading from one of the IFM memories and a writing on the other, until the last but one group of OFM pixels are received. The stage ends when the last OFM pixels are received and the OFM pixels are reorganized ($Reorganize\_pixel\_images(0)$).

For stages 5 and 6, the software application proceeds as described for the previous stages, except when sending the pointwise weights and shifts. In this situation, since the weights and shifts cannot be stored at once to on-chip memory, the algorithm first sends the maximum pointwise parameters that can be stored on the IP memories. Then, when half of these parameters are used, the software application selects a new group of weights and shifts and sends them to the PL, while the other half of the parameters are being used to process the current pointwise convolutions. The process continues until all the pointwise parameters are used, and applied to each IFM tile. This requires sending the different groups of pointwise weights and shifts, N_P, to on-chip memories, S_P times.

Stages 7 to 13 are performed completely in hardware, thus, in this situation, the software application only needs to control the data transfers, and the algorithm starts by sending to the PL the different types of weights and Batch Norm parameters required to process the current stage. After that, the transfer process is similar as in stages 5 and 6. However, here the padding is applied in the hardware when loading the pixels to the SRL. Also, when all the OFMs are received there is not necessary to reorganize them.

To observe how the data is managed to/from the PL, table 6.2 presents the number of transfers for each stage for the different types of parameters. The first three columns show the depthwise weights (DW), depthwise Batch Norm parameters (DW BN) and pointwise Batch Norm parameters (PW BN), which are all represented with 16 bits and sent to the IP memories in a single transfer. Then, the fourth column presents the pointwise values or scale factors (PW/PS) and this column is divided into 3 sub-columns. The P sub-column shows the size of the 8-bit block, tiled from the values (or scale factors), sent in each transfer. These blocks are represented in such a way that the left number indicates the size of the kernel, and the right number, the number of kernels.

Table 6.2: Number of tiles and transfers for each stage for the different type of parameters.

	DW	**DW BN**	**PW BN**	**PW/PS**			**IFM**			**OFM**	
Layer	P (16b)	P (16b)	P (16b)	P (8b)	N_P	S_P	P (8b)	N_{IFM}	S_{IFM}	P (8b)	R
1	9 x 32	3 x 32	3 x 64	32 x 64	1	1	16 x 16 x 32	64	1	14 x 14 x 32	128
2	9 x 64	3 x 64	3 x 128	64 x 128	1	1	15 x 15 x 32	128	1	7 x 7 x 128	64
3	9 x 128	3 x 128	3 x 128	128 x 128	1	1	16 x 16 x 32	256	1	14 x 14 x 32	256
4	9 x 128	3 x 128	3 x 256	128 x 128	2	1	15 x 15 x 32	64	2	7 x 7 x 128	32
5	9 x 256	3 x 256	3 x 256	256 x 64	4	4	16 x 16 x 32	32	8	14 x 14 x 32	32
6	9 x 256	3 x 256	3 x 512	128 x 128	8	4	15 x 15 x 32	32	4	7 x 7 x 128	16
7	9 x 512	3 x 512	3 x 512	512 x 32	16	1	14 x 14 x 32	16	16	14 x 14 x 32	16
8	9 x 512	3 x 512	3 x 512	512 x 32	16	1	14 x 14 x 32	16	16	14 x 14 x 32	16
9	9 x 512	3 x 512	3 x 512	512 x 32	16	1	14 x 14 x 32	16	16	14 x 14 x 32	16
10	9 x 512	3 x 512	3 x 512	512 x 32	16	1	14 x 14 x 32	16	16	14 x 14 x 32	16
11	9 x 512	3 x 512	3 x 512	512 x 32	16	1	14 x 14 x 32	16	16	14 x 14 x 32	16
12	9 x 512	3 x 512	3 x 1,024	128 x 128	32	1	14 x 14 x 32	16	8	7 x 7 x 128	8
13	9 x 1,024	3 x 1,024	3 x 1,024	128 x 128	64	1	7 x 7 x 128	8	8	7 x 7 x 128	8

The number of tiled blocks and the number of times each of these tiled blocks are transferred to the hardware are represented in the next two sub-columns, respectively by N_P and S_P. As an example, in stage 6, the 512 filters have a size of 256, the same size as the IFMs. These filters are tiled in 8 blocks of 128 filters, each one with size 128. Moreover, each of these blocks is transferred to the hardware 4 times. The IFM column has also 3 sub-columns. The P sub-column shows the size of the block, tiled from the IFM sent in each transfer. These blocks are represented in height, width and depth with 8 bits size. The next sub-columns represent the number of tiled blocks (N_{IFM}) of the IFMs and the number of times each of these tiled blocks are transferred to the hardware (S_{IFM}). Lastly, the OFM column has 2 sub-columns which show the size of the blocks represented in height, width and depth with 8 bits (P) and the number of blocks (R) received from the PL.

6.2. Experimental results

This section evaluates the MobileNets accelerator performance comparing the software and hardware/software implementation. The architecture was synthesized using the Vivado 2019.1 and executed using a clock frequency of 115MHz. To maximize the number of results per clock cycle, the architecture uses a pipeline implementation and two of the three types of parallelism presented in section 3.2.1. Operator level parallelism is used when performing the MAC in both depthwise and pointwise PEs, using the 79 DSPs at the same time to produce 2 OFM results per clock cycle. On the other hand, intra-output parallelism is used in the Conv sub-module of the pointwise PE since two different filters are applied simultaneously to the IFM pixels.

While the computation is performed, data is exchanged between the DDR and PL, loading the on-chip memories with new values, and the DDR with OFMs pixels. This process can be observed in figure 6.2 which presents an overview of how the data exchange and computation are performed during the MobileNets inference.

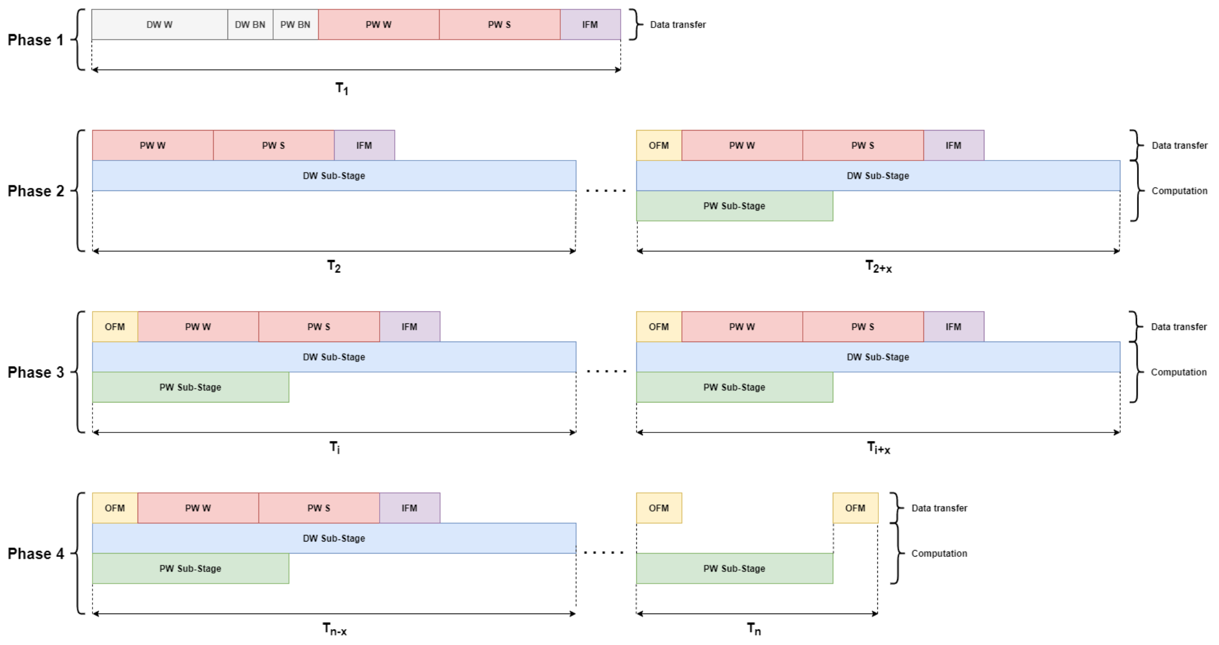


Figure 6.2: Overview of the four phases of the data exchange and computation.

In all layers and situations, the depthwise sub-stage takes 8,244 clock cycles to perform, while the pointwise sub-stage takes only 3,152 clock cycles. What varies is the number of depthwise and pointwise sub-stages applied to each layer and the number and size of different data transferred (see table 6.2). Therefore, figure 6.2 represents the worst case, i.e., the situation where a depthwise sub-stage is always followed by a depthwise and pointwise sub-stage (except for T_n) and each type of data transferred has the maximum size. In phase 1, only data is transferred to the PL, namely the depthwise weights (DW W), depthwise Batch Norm parameters (DW BN), pointwise Batch Norm parameters (PW BN), a group of IFM pixels (IFM), pointwise values (PW W) and pointwise scale factors (PW S). After that, the software application sends a group of pointwise values and scale factors to the PL followed by another group of IFM pixels, while the first DW sub-stage of the current layer is performed, as presented in the first step (T_2) of phase 2. This step is equal to all layers, the only difference is that in the first 3 layers all the pointwise parameters are sent in this transfer. From here, the next steps of phase 2 depend on the layer, thus, there can be performed a single pointwise sub-stage or both depthwise and pointwise sub-stages. Also, groups of IFM are always transferred on the following steps, however, not all of them receives OFMs. The main objective of phase 2 is to start and synchronize the computation to a point where the next phase can iterate the same process several times. Phase 3 is where most of the computation and data exchange occurs, thus, this phase has several configurations and iterations, depending on the layer. The process behind this phase is always the same: send the different types of data to process a batch of IFMs, synchronize the computation to receive new IFMs and perform the accumulation of the intermediate results, receive a group of OFMs pixels, and repeat the process until iteration $N_{IFM} \times S_{IFM} - 1$ (see table 6.2). Finally, in phase 4 the last sub-stages are performed. This phase is similar to the process iteration of phase 3, the only difference is the last step (T_n), where only pointwise sub-stages are performed and OFM pixels received.

The previous description shows that with the exception of the first and the last, for the worst case, all T intervals depend on the depthwise stage, making it possible to transfer all the data required in each T interval. However, neither of the layers require the transfer of pointwise values and scale factors during all sub-stages, and when required, these parameters are transferred during a depthwise sub-stage. The IFM and OFM pixels can be transferred during depthwise or pointwise computation without restrictions. Below is a general description of each of the phases applied to layer 1 and layers 7 to 11.

In the first phase of layer 1 (figure 6.3), the depthwise weights and all Batch Norm parameters are loaded into IP memories, as well as the first group of IFM pixels. The size of these data types can be observed in table 6.2. After that, in phase 2, the depthwise computation begins and at the same moment, all the pointwise values and scale factors are stored into IP memories. Then, the software application waits for the synchronization signal (S) to send another group of IMF pixels while the pointwise is performing the computation. A single depthwise sub-stage produces enough results to perform 2 pointwise sub-stages. Phase 3 is where the OFM pixels begin to be transferred to the DDR and this phase is where most of the computation and transfer of data is performed. Observing table 6.2 and since 2 IFMs are transferred in the previous phases, phase 3 require the transfer of 62 IFMs to the IP memories. All the other data are already stored on IP memories. Therefore, the process in phase 3 repeats 62 times before advancing to the next phase. Lastly, in phase 4 the last depthwise and last 3 pointwise sub-stages are performed, to produce the last 4 groups of OFM pixels.

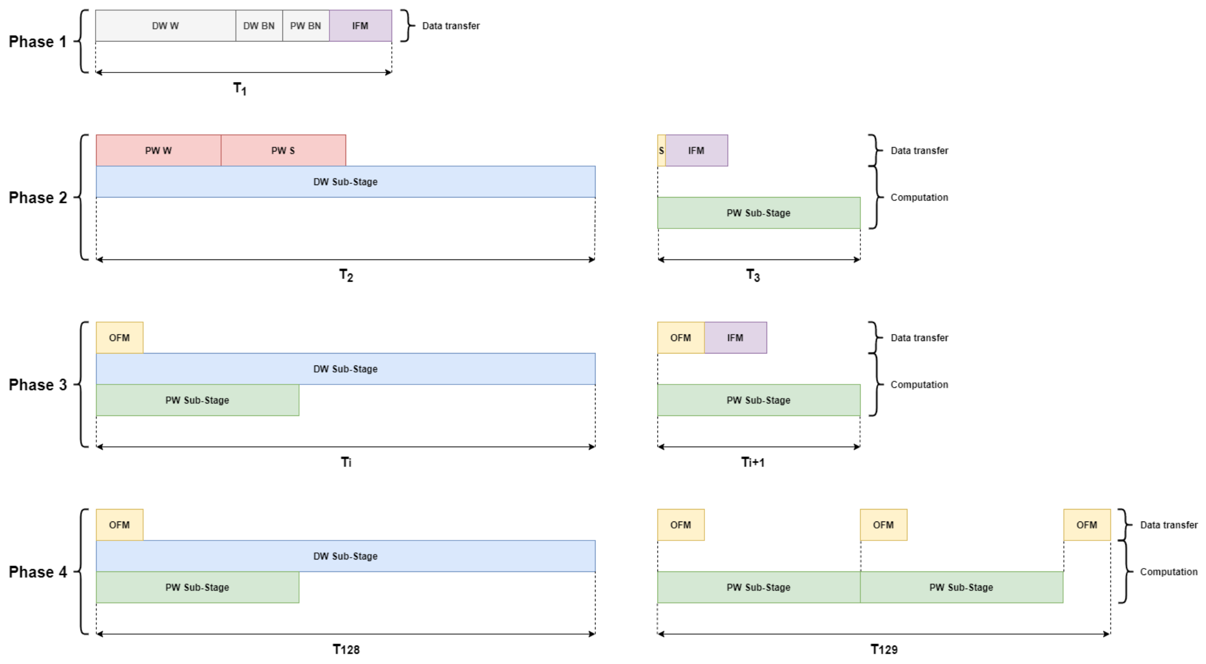


Figure 6.3: Four phases of the data exchange and computation in layer 1.

In phase 1 of layers 7-11 (Figure 6.4), the depthwise weights and all Batch Norm parameters, as well as the first group of IFM pixels and a group of pointwise values and scale factors, are loaded into IP memories. Here the pointwise values and scale factors are transferred before starting the computation because each group of pointwise values and scale factors are used to produce one group of OFM pixels. Therefore, to keep the computation without interruption, when a group of values and scale factors are completely used, the computation starts using the other group while new values and scale factors are being stored. After this, phase 2 begins with the storage of the second group of pointwise values and scale factors into IP memories while the depthwise computation is executing. This is the first step (T_2) of the 16 that compose phase 2. The following 14 steps $(T_3 \; to \; T_{16})$ are similar to the last step (T_{17}), however instead of receiving a group of OFM pixels, these 14 steps receive a synchronization signal (s) to inform the software application that the IP memories are ready to receive a new group of IFM pixels. This happens because in layers 7 to 11, the IFMs are $14 \times 14 \times 512$ pixels and each block sent to the PL has $14 \times 14 \times 32$ pixels, thus, to produce a group of $14 \times 14 \times 32$ OFM pixels, 16 groups of IFM pixels are necessary. Phase 3 is also composed of 16 steps. Observing table 6.2 and since this phase performs $N_{IFM} \times S_{IFM} - 2$ iterations, phase 3 need to repeat the process 14 times. The last phase performs the last iterations, similar to the previous one, however, it is not necessary to transfer more pointwise parameters and the last step only performs a pointwise sub-stage.

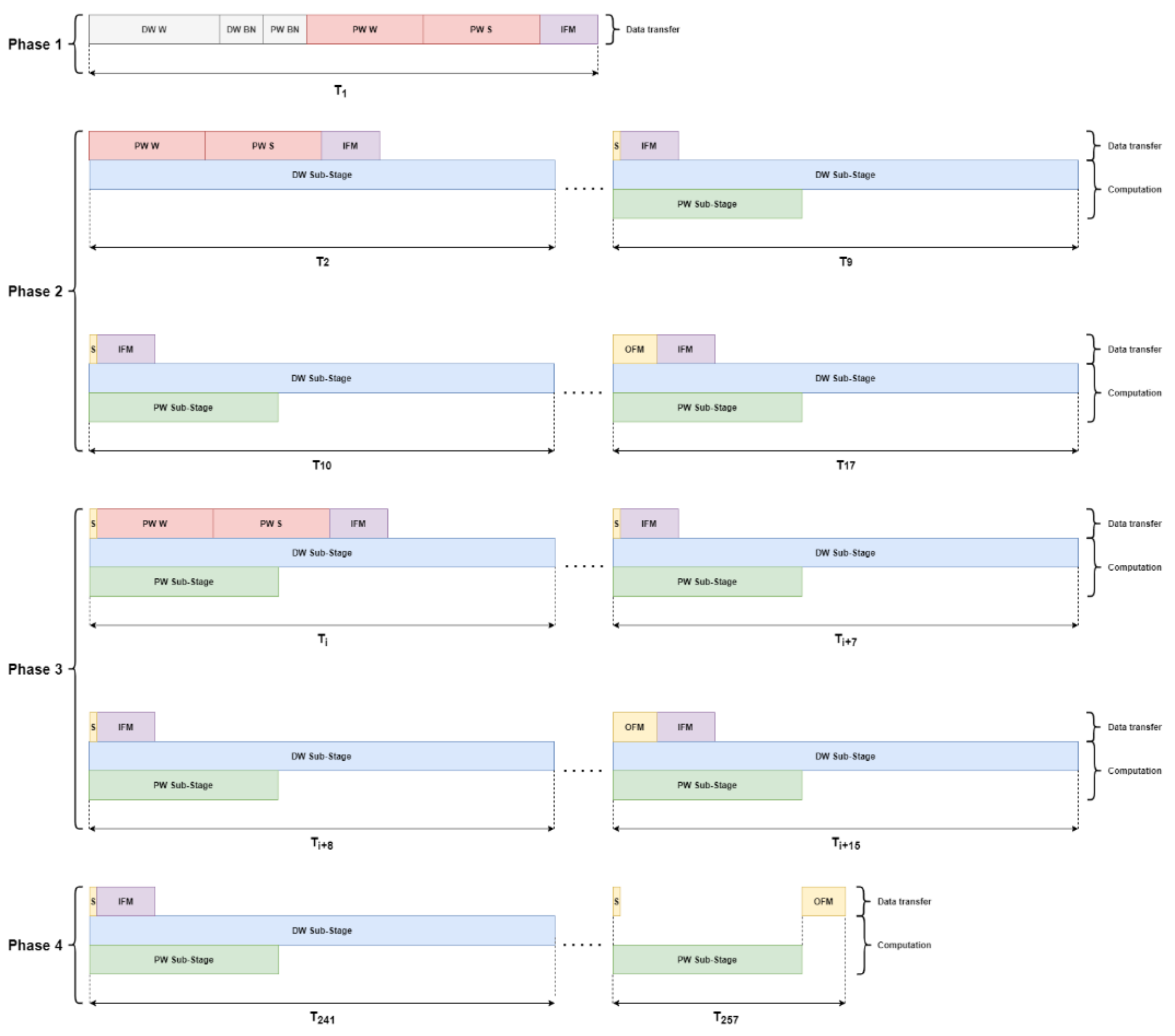


Figure 6.4: Four phases of the data exchange and computation in layer 7 to 11.

This section concludes by comparing the execution time of the software and the hardware/software implementation, using O2 and O3 software compilation optimizations. These two optimizations were chosen because they present the best performance. Table 6.3 presents the comparison results between algorithm 3 and algorithm 5. With the best optimization (-O3) the hardware/software implementation is 16 times faster than the software application, and considering only the last 7 stages (all implemented in hardware), the speedup is 18. Another particularity observed on the table is that the software optimization does not affect the execution time of the hardware/software implementation, despite using tiling and performing the reorganization of the OFM pixels in the first 6 layers using the software application.

Combining all the MobileNets layers and using the O3 software compilation optimization, the complete inference process execution time of the hardware/software implementation is 469 ms, representing a speed up of 11 compared to the software implementation.

Table 6.3: Time execution results for Software and Hardware/Software using optimization O2 and O3.

Layer	SW (02) (ms)	SW + HW (02) (ms)	Speed up	SW (03) (ms)	SW + HW (03) (ms)	Speed up
1	468	21	22	385	21	19
2	392	33	12	302	32	9
3	754	39	19	548	38	14
4	323	24	13	250	23	11
5	674	33	20	492	33	15
6	303	19	16	232	19	12
7	598	26	23	451	26	17
8	598	26	23	450	26	17
9	598	26	23	450	26	17
10	598	26	23	450	26	17
11	598	26	23	450	26	17
12	293	13	23	222	13	17
13	596	18	33	438	18	24
Total	6,793	330	21	5,122	328	16

6.3. Conclusion

This chapter begins by introducing the complete MobileNets accelerator developed in this project and the respective resources used. One objective outlined during the architecture design was to use the most number of BRAMs and DSPs, to develop a system able to produce the maximum possible results per clock cycle. Observing table 6.1, this requirement was accomplished since almost all BRAMs and DSPs were used in this project.

After showing the complete MobileNets accelerator and resources used, an algorithm was presented to understand how the hardware/software system performs all the functions of the inference process. Table 6.2 helps understanding how many times each function needs to be called to complete a certain stage and consequently the inference process.

The last section of the chapter presents the experimental results. To understand the workload of the system, the exchange of data and computation of the different depthwise and pointwise stages performed in parallel is fully analysed. As more hardware resources have been allocated to process the pointwise stages, these stages are now fully accelerated. Therefore, the execution time of the inference process is no longer limited by these stages, contrary to what happens on the software application. Finally, the results of table 6.3 show that the proposed hardware/software implementation is 16 times faster executing the 13 depthwise/pointwise stages, compared to the software implementation (with the O3 optimization)

Chapter 7

Conclusion

The work developed and presented in this master's thesis focused on the research and development of a custom hardware/software architecture to execute the MobileNets inference process on an SoC FPGA. The main requirement defined for this project was implementing efficiently the hardware/software architecture on a low-cost device making a minimal trade-off between accuracy and execution time.

Although MobileNets is a CNN model for image classification designed to run on embedded systems, the standard model may still be too computing intensive to run on low-cost devices such as the Xilinx Zynq 7010 or 7020. Due to this, a quantization analysis was conducted to evaluate the model when different representations are used for the model parameters, instead of the 32-bit floating-point. This allowed to reduce the MobileNets model using 16-bit fixed-point to quantize 26% of the parameters, and 12-bit custom floating-point to quantize the remaining 74%. The final result obtained was a MobileNets model 60% smaller than the standard with only a very small accuracy reduction of 0.78%, which is below the 1% of maximum stipulated for the accuracy loss.

The hardware was designed to execute the quantized MobileNets model resulting from the previous analysis. Two types of parallelism were used to process the MACs in parallel. Operator-level parallelism is used when performing the MAC in both depthwise and pointwise PEs, and intra-output parallelism is used in the Conv sub-module of the pointwise PE since two different filters are applied simultaneously to the IFM pixels. The architecture developed follows a pipeline structure to produce a valid result per clock cycle. Also, the use of ping-pong memories hides the latency and allows the exchange of data between the PS and the PL while the computation of the depthwise and pointwise is running.

Lastly, the proposed hardware/software was demonstrated and analysed on the Xilinx Zynq 7010 device. The design of the architecture was the main challenge of this project because of the limited number of resources of the device. Almost all of the BRAMs and DSPs were used, which was another requirement defined for this project. The proposed hardware/software achieved a speed up of 16 times for the 13 stages, compared to the software implementation (with the O3 optimization). Another requirement for this project was to reduce the execution time of the inference process below 1 second, which was also successfully achieved since the final execution time of the hardware/software implementation is 469 ms.

7.1. Future work

Despite all the project requirements being achieved there are some considerations that can improve this work. The architecture can be adapted to perform also the Conv3D and FC layer in hardware since now they represent about 30% of the execution time. Observing figure 2.16, the filter size of the Conv3D layer has a size of $3 \times 3 \times 3 = 27$. Therefore, the depthwise IP may be adapted to perform the Conv3D convolution since the module can perform $3 \times 3 = 9$ MACs every clock cycle. This means that the depthwise IP would need 3 clock cycles to produce one MAC of the Conv3D convolution. On the other hand, observing figure 2.20, the FC layer is similar to the pointwise convolution. The difference is the size of the IFMs and the bias sum after the convolution. Therefore, the pointwise IP may be adapted to perform the FC layer, using a similar process as in stage 12 or 13.

The architecture of this project may also be adapted to work in devices with more resources. As an example, the Xilinx Zynq 7020 device has more than twice the resources of the Xilinx Zynq 7010 device. Therefore, the pointwise module can be modified to perform 4 filters at a time, while the depthwise module may be able to perform 2 kernels at a time. However, in some stages, the data transfers can take more than the depthwise sub-stages to perform (see figure 6.2), which would corrupt the data and invalidate the inference process. One solution to this problem is to divide the pointwise parameters into smaller blocks and perform more transfers of these blocks, instead of sending them at the beginning of each iteration.

Lastly, the custom floating-point can be explored in future works, namely, quantizing all the MobileNets parameters with the custom floating-point strategy or extend the concept to other CNNs. Therefore, as more bits are used in the exponent, higher accuracies may be achieved.

References

[1] C. Gershenson, “Artificial Neural Networks for Beginners,” *arXiv:cs/0308031,* September 2003.

[2] K. O’Shea and R. Nash, "An Introduction to Convolutional Neural Networks".*arXiv:1511.08458.*

[3] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv:1704.04861,* 17 April 2017.

[4] K.-L. Du and M. N. S. Swamy, "Neural Networks and Statistical Learning," December 2014.

[5] R. Rosenblatt, "The Perceptron: A probabilistic model for information storage and organization in the brain.," *Psychological Rev.,* pp. 386-408, 1958.

[6] M. A. Nilsen, “Neural Networks and Deep Learning,” Determination Press, 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com/chap1.html. [Accessed 16 September 2020].

[7] "Capítulo 3 – O Que São Redes Neurais Artificiais Profundas ou Deep Learning?," Data Learning Book, 2018. [Online]. Available: http://deeplearningbook.com.br/o-que-sao-redes-neurais-artificiais-profundas/. [Accessed 16 September 2020].

[8] S. Saha, "A Comprehensive Guide to Convolutional Neural Networks - the ELI5 way," [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. [Accessed 16 September 2020].

[9] "Student Notes: Convolutional Neural Networks (CNN) Introduction," [Online]. Available: https://indoml.com/2018/03/07/student-notes-convolutional-neural-networks-cnn-introduction/. [Accessed 20 September 2020].

[10] A. Deshpande, "A beginner's guide to understanding convolutional neural networks," 2016. [Online]. Available: https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/. [Accessed 16 September 2020].

[11] I. Hadji and R. P. Wildes, "What Do We Understand About Convolutional Networks?," *arXiv:1803.08834v1,* 23 March 2018.

[12] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv:1502.03167,* February 2015.

[13] A. Fred and M. Agarap, "Deep Learning using Rectified Linear Units (ReLU)," *arXiv:1803.08375,* March 2018.

[14] M. Zahangir Alom, T. M. Taha, C. Yakopcic, S. Westberg, P. Sidike, M. S. Nasrin, B. C. van Essen, A. A. S. Awwal and V. K. Asari, "The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches," *arXiv:1803.01164,* March 2018.

[15] A. Canziani, E. Culurciello and A. Paszke, "An analysis of deep neural networks models for pratical applications," *arXiv:1605.07678,* April 2014.

[16] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 1MB model size," *arXiv:1602.07360,* February 2016.

[17] "TensorFlow," [Online]. Available: https://www.tensorflow.org/. [Accessed 16 September 2020].

[18] P. Sadowski, "Notes on Backpropagation".

[19] S. Han, H. Mao and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *arXiv:1510.00149,* October 2015.

[20] P. Molchanov, S. Tyree, T. Karras, T. Aila and J. Kautz, "Pruning Convolutional Neural Networks for Resource Efficient Inference," *arXiv:1611.06440,* June 2017.

[21] S. Anwar, K. Hwang and W. Sung, "Structured Pruning of Deep Convolutional Neural Networks," *arXiv:1512.08571,* December 2015.

[22] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *arXiv:1806.08342,* 21 June 2018.

[23] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu and N. Sun, "Dadiannao: A machine-learning supercomputer," *in Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO),* June 2018.

[24] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, Y. Wang and H. Yang, "Going deeper with embedded fpga platform for convolutional neural network," *in Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA*

’16. New York, NY, USA: ACM, February 2016.

[25] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam and D. Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," *arXiv:1712.05877v1,* December 2017.

[26] T. Sheng, C. Feng, S. Zhuo, X. Zhang, L. Shen and M. Aleksic, "A Quantization-Friendly Separable Convolution for MobileNets," *arXiv:1803.08607,* March 2018.

[27] "TensorFlow Lite," [Online]. Available: https://www.tensorflow.org/lite . [Accessed 18 November 2020].

[28] "Wireless Technology & Innovation | Mobile Technology | Qualcomm," [Online]. Available: https://www.qualcomm.com/. [Accessed 13 February 2021].

[29] "Architecting a Smarter World – Arm," 13 February 2021. [Online]. Available: https://www.arm.com/.

[30] J. Su, J. Faraone, J. Liu, Y. Zhao, D. B. Thomas, P. H. W. Leong and P. Y. K. Cheung, "Redundancy-reduced mobilenet acceleration on reconfigurable logic for imagenet classification," *in Applied Reconfigurable Computing. Architectures, Tools, and Applications,* April 2018.

[31] S. Chakradhar, M. Sankaradas, V. Jakkula and S. Cadambi, "A Dynamically Configurable Coprocessor for Convolutional Neural Networks," *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture,* p. 247–257, June 2010.

[32] L. Bai, Y. Zhao and X. Huang, "A CNN Accelerator on FPGA Using Depthwise Separable Convolution," *arXiv:1809.01536v2,* 6 September 2018.

[33] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou and L. Wang, "A high performance fpga-based accelerator for large-scale convolutional neural networks," *in Field Programmable Logic and Applications (FPL), 2016 26th International Conference,* 29 August 2016.

[34] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov and L.-C. Chen, "MobileNetV2 Inverted Residuals and Linear Bottlenecks," *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) arXiv:1801.04381,* pp. 4510-4520, 13 January 2018.

[35] "Keras," [Online]. Available: https://keras.io/api/applications/mobilenet/#mobilenet-function. [Accessed 18 November 2020].

[36] "Keras applications," [Online]. Available: https://keras.io/api/applications/. [Accessed 18 November 2020].

[37] "Zynq-7000 SoC," [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html. [Accessed 13 April 2021].

[38] L. H. Crockett, R. A. Elliot, M. A. Enderwitz and R. W. Stewart, The Zynq Book, Strathclyde Academic Media, 2014.

[39] S. Gallagher, "Mapping DSP algorithms into FPGAs," [Online]. Available: https://www.ieee.li/pdf/viewgraphs/mapping_dsp_algorithms_into_fpgas.pdf. [Accessed 13 April 2021].

[40] "FPGA Developer," [Online]. Available: https://www.fpgadeveloper.com/2014/08/using-the-axi-dma-in-vivado.html. [Accessed 13 April 2021].