

**École polytechnique de Louvain**

# **Accelerating Convolutional Neural Networks for FPGA using Depthwise Separable Convolution and Pruning**

Author: **Guillaume GHEYSEN**

Supervisors: **Prof. Jean-Didier LEGAT, Prof. Christophe DE VLEESCHOUWER**

Readers: **Victor JOOS DE TER BEERST, Martin LEFEBVRE**

Academic year 2019–2020

Master [120] in Computer Science and Engineering



# ABSTRACT

In the domain of Artificial Intelligence and Deep Learning, Convolutional Neural Networks (CNN) are powerful models used in image classification, speech recognition, medical image analysis, etc. However, those models require a large number of parameters and operations to be executed. Therefore, GPUs are the dominant platform to implement CNNs thanks to their huge computational resources and memory. However, such platforms are power-hungry and limit the implementation of CNNs on mobile and embedded devices. A solution would be to use, rather than GPU, FPGA which are more energy efficient but do not possess enough resources to implement the most performing models.

The analysis of the literature revealed a series of optimizations that could be applied to make it possible to efficiently implement CNNs on FPGA. From this analysis, two optimizations appeared to be most promising.

Therefore, this thesis aims at combining the advantages of those two optimizations, pruning, and depthwise separable convolution, to reduce the number of parameters and operations required by a CNN and make an efficient implementation of CNNs on FPGA possible. To demonstrate the performance of the pruning scheme, an FPGA-based accelerator was designed combining both optimizations.

This thesis shows that combining both pruning and depthwise separable convolution provides an efficient way to reduce both the number of weights and computational complexity. Moreover, the proposed pruning scheme can be efficiently handled on FPGA without any loss of performance when the sparsity increases.



# ACKNOWLEDGMENTS

I would like to acknowledge the help of some people who were crucial for the completion of this thesis.

First I would like to thank my two supervisors: Prof. Jean-Didier Legat and Prof. Christophe de Vleeschouwer for their support. They provided valuable advice helping me to define the scope of this work and guiding me through its completion.

Second, I would like to thank the two readers of this thesis: Martin Lefebvre and Victor Joos de ter Beerst for devoting their time to read this thesis and assess my work.

Third, I wish to acknowledge the assistance provided by Victor Joos de ter Beerst and Antoine Vanderschueren to accomplish this work, which was valuable for me. Their availability to answer my questions was a great help.

Fourth, I would like to thank my family and friends, who were always there to help me when I needed the most. I want also to acknowledge the support of Clara, who was a huge source of inspiration to outdo myself. Moreover, special thanks go to the members of the IngiGang for the time spent during those five years together and the support they provided me.

Finally, I would like to personally thank all the people who have proofread this thesis, in particular Martine, Julie, Florian, Martin, Maxime, and last but not least Guillaume for giving their precious time to improve the quality of this work.



# Contents

## List of Figures

## List of Tables

|                                                                                                                |           |
|----------------------------------------------------------------------------------------------------------------|-----------|
| <b>Introduction</b>                                                                                            | <b>1</b>  |
| Structure of the thesis . . . . .                                                                              | 3         |
| <b>1 Background</b>                                                                                            | <b>5</b>  |
| 1.1 Convolutional Neural Network . . . . .                                                                     | 5         |
| 1.1.1 Structure of a CNN . . . . .                                                                             | 5         |
| 1.1.2 CNN Models . . . . .                                                                                     | 15        |
| 1.1.3 Training . . . . .                                                                                       | 18        |
| 1.2 Field Programmable Gate Array (FPGA) . . . . .                                                             | 20        |
| <b>2 State of the Art</b>                                                                                      | <b>22</b> |
| 2.1 CNN optimizations for FPGA . . . . .                                                                       | 22        |
| 2.1.1 Algorithmic Optimizations . . . . .                                                                      | 22        |
| 2.1.2 Model Optimizations . . . . .                                                                            | 25        |
| 2.2 CNN optimizations on FPGA . . . . .                                                                        | 32        |
| 2.2.1 General CNN inference on FPGA . . . . .                                                                  | 32        |
| 2.2.2 DSC . . . . .                                                                                            | 39        |
| 2.2.3 Pruning . . . . .                                                                                        | 40        |
| <b>3 Accelerating Convolutional Neural Networks for FPGA using Depthwise Separable Convolution and Pruning</b> | <b>41</b> |
| 3.1 Design of an FPGA-based accelerator integrating structured sparse DSC . . . .                              | 42        |

|       |                                                                                          |           |
|-------|------------------------------------------------------------------------------------------|-----------|
| 3.1.1 | Pruning scheme . . . . .                                                                 | 42        |
| 3.1.2 | Compressed format . . . . .                                                              | 45        |
| 3.1.3 | Adding the pruning scheme to MobileNetV2 . . . . .                                       | 48        |
| 3.1.4 | Loop analysis . . . . .                                                                  | 51        |
| 3.2   | Implementing sparse DSC on FPGA . . . . .                                                | 56        |
| 3.2.1 | Overall architecture . . . . .                                                           | 56        |
| 3.2.2 | External memory . . . . .                                                                | 58        |
| 3.2.3 | Main controller . . . . .                                                                | 61        |
| 3.2.4 | DMA . . . . .                                                                            | 63        |
| 3.2.5 | 1×1 convolution PE . . . . .                                                             | 65        |
| 3.2.6 | DSC PE . . . . .                                                                         | 66        |
| 3.2.7 | Buffer . . . . .                                                                         | 67        |
| 3.3   | Results and discussion . . . . .                                                         | 69        |
| 3.3.1 | Experimental setup . . . . .                                                             | 69        |
| 3.3.2 | Results . . . . .                                                                        | 71        |
| 3.3.3 | Discussion . . . . .                                                                     | 74        |
|       | <b>Conclusion</b>                                                                        | <b>76</b> |
|       | Future Works . . . . .                                                                   | 77        |
|       | <b>Appendix A MobileNetV2</b>                                                            | <b>78</b> |
|       | <b>Appendix B Weight and operation reduction factors</b>                                 | <b>79</b> |
|       | <b>Appendix C Minimal pruning ratio required to reduce memory utilization</b>            | <b>81</b> |
|       | <b>Appendix D Details of the different signals of each component of the architecture</b> | <b>82</b> |

# List of Figures

|      |                                                                                                          |    |
|------|----------------------------------------------------------------------------------------------------------|----|
| 1    | CNN execution phases, inspired by [16]                                                                   | 2  |
| 1.1  | The Perceptron                                                                                           | 6  |
| 1.2  | Activation functions                                                                                     | 7  |
| 1.3  | A fully connected layer                                                                                  | 9  |
| 1.4  | Volumes involved in the convolution operations                                                           | 10 |
| 1.5  | Convolution operation                                                                                    | 11 |
| 1.6  | Depthwise Separable Convolution, inspired by [12, 39]                                                    | 13 |
| 1.7  | Examples of pooling layers                                                                               | 14 |
| 1.8  | Working principle of a CNN and the different layers which composing it                                   | 14 |
| 1.9  | An illustration of the architecture of AlexNet, from [25]                                                | 15 |
| 1.10 | An illustration of the architecture of VGG16, from [42]                                                  | 16 |
| 1.11 | ResNet building block, from [11]                                                                         | 17 |
| 1.12 | Bottleneck residual block, from [22]                                                                     | 18 |
| 1.13 | General FPGA layout, from [47]                                                                           | 20 |
| 2.1  | GEMM-based processing of a standard convolution layer, from [18]                                         | 23 |
| 2.2  | Convolutional block from different architectures                                                         | 27 |
| 2.3  | Ball chart reporting the Top-1 accuracy of various architectures vs. their computational complexity [67] | 28 |
| 2.4  | Structured pruning schemes, where the yellow weights are the pruned ones, inspired by [21]               | 30 |
| 2.5  | Accuracy-Sparsity Curve of AlexNet obtained by pruning [74]                                              | 30 |
| 2.6  | Pruning a depthwise separable convolution [79]                                                           | 31 |
| 2.7  | Illustration of the Q-format                                                                             | 31 |

|      |                                                                                                                                                       |    |
|------|-------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 2.8  | Static Systolic Arrays, inspired by [18]                                                                                                              | 33 |
| 2.9  | Graph representation of a convolution layer, inspired by [18]                                                                                         | 34 |
| 2.10 | FPGA memory hierarchy, from [34]                                                                                                                      | 35 |
| 2.11 | Loop Tiling, from [18]                                                                                                                                | 36 |
| 2.12 | Loop unrolling configurations, from [34]                                                                                                              | 36 |
| 2.13 | Design space of platform-supported designs, from [20]                                                                                                 | 37 |
| 2.14 | Design space exploration of: (a) total number of partial sums that need to be stored in memory; (b) the number of external memory accesses, from [34] | 38 |
| 2.15 | Weight buffer in ping-pong structure, from [39]                                                                                                       | 39 |
| 3.1  | Process of a pointwise convolution with a sparse pointwise kernel, where $N_{par} = 4$ and $N_{np} = 2$ , inspired by [75]                            | 43 |
| 3.2  | Evolution of the reduction factors depending on the pruning ratio and the number of output channels                                                   | 45 |
| 3.3  | A pointwise kernel after pruning, where black cubes are pruned weights                                                                                | 46 |
| 3.4  | Illustration of the compressed format, where $N_{par} = 4$ and $N_{np} = 2$                                                                           | 47 |
| 3.5  | Minimal ratio of pruned weights if $BW = 16$                                                                                                          | 47 |
| 3.6  | Illustration of the algorithm used to perform the convolutions                                                                                        | 48 |
| 3.7  | Representation of the sparse $1 \times 1$ convolution                                                                                                 | 49 |
| 3.8  | Representation of the sparse DSC convolution                                                                                                          | 50 |
| 3.9  | Tiling process on an input FM, where the green area is the input pixels that would be loaded more than once due to 2D convolution                     | 55 |
| 3.10 | Overall architecture of the accelerator                                                                                                               | 57 |
| 3.11 | overall structure of the external memory                                                                                                              | 59 |
| 3.12 | Maximum external memory required in $MB$ in relation with the bitwidth of each data $BW_{pixel}$ and $BW_{weight}$                                    | 60 |
| 3.13 | FSM of the main controller                                                                                                                            | 61 |
| 3.14 | Illustration of the concept of the pixel of reference and padding                                                                                     | 63 |
| 3.15 | FSM of the DMA                                                                                                                                        | 64 |

|      |                                                                                                              |    |
|------|--------------------------------------------------------------------------------------------------------------|----|
| 3.16 | FSM of the $1 \times 1$ convolution PE . . . . .                                                             | 65 |
| 3.17 | PE structure performing the convolution where $N_{par} = 4$ and $N_{np} = 2$ , inspired<br>by [75] . . . . . | 65 |
| 3.18 | FSM of the DSC PE . . . . .                                                                                  | 66 |
| 3.19 | PE structure performing the depthwise convolution, inspired by [39] . . . . .                                | 67 |
| 3.20 | General architecture of a buffer component . . . . .                                                         | 68 |
| 3.21 | Ratio of correct output depending on the number of partial sums per output pixel                             | 71 |
| 3.22 | Number of cycles and Reduction factor of the number of cycles . . . . .                                      | 72 |
| 3.23 | Number of cycles and Reduction factor of the number of cycles for an $N_p$ fixed .                           | 73 |
| 3.24 | Repartition of the latency across the different components for various pruning<br>parameters . . . . .       | 73 |
| A.1  | MobileNetV2 structure, from [22] . . . . .                                                                   | 78 |

# List of Tables

|     |                                                                                                    |    |
|-----|----------------------------------------------------------------------------------------------------|----|
| 1   | FPGA's Reduction Ratios to GPU, from [14] . . . . .                                                | 2  |
| 1.1 | Bottleneck convolution, from [22] . . . . .                                                        | 18 |
| 2.1 | Performance on ImageNet, comparison for different networks [22] . . . . .                          | 28 |
| 3.1 | Pruning parameters . . . . .                                                                       | 43 |
| 3.2 | $1 \times 1$ convolution loop dimensions and hardware design variables, inspired by [34] . . . . . | 52 |
| 3.3 | DSC loop dimensions and hardware design variables, inspired by [34] . . . . .                      | 52 |
| 3.4 | Parameters of the implemented layer . . . . .                                                      | 70 |
| 3.5 | Resource utilization of the network . . . . .                                                      | 74 |
| D.1 | Input and output signals of the main controller . . . . .                                          | 82 |
| D.2 | Input and output signals of the DMA . . . . .                                                      | 83 |
| D.3 | Input and output signals of the $1 \times 1$ convolution PE . . . . .                              | 84 |
| D.4 | Input and output signals of the DSC convolution PE . . . . .                                       | 85 |

# List of Abbreviations

**AI** Artificial Intelligence.

**ASIC** Application-Specific Integrated Circuit.

**BRAM** Block RAM.

**CNN** Convolutional Neural Network.

**CPU** Central Processing Unit.

**CSR** Compressed Sparse Row.

**DL** Deep Learning.

**DMA** Direct Memory Access.

**DSC** Depthwise Separable Convolution.

**DSP** Digital Signal Processing.

**DUT** Device Under Test.

**FFT** Fast Fourier Transform.

**FIFO** First-in First-out.

**FM** Feature Map.

**FPGA** Field-Programmable Gate Array.

**FSM** Final State Machine.

**GEMM** General Matrix Multiplication.

**GPU** Graphical Processing Unit.

**HDL** Hardware Description Language.

**HPS** Hard Processor System.

**IOE** input/output elements.

**LE** logic element.

**MAC** Multiply-Accumulate.

**ML** Machine Learning.

**MoC** Model of Computation.

**NAS** Neural Architectural Search.

**NN** Neural Network.

**OaA** Overlap-and-Add.

**PE** Processing Element.

**RNN** Recurrent Neural Network.



# Introduction

Since the last decade, Artificial Intelligence (AI) has become increasingly part of our daily lives: chatbot, self-driving car, or chess game are a few examples [1]. AI is indeed an “umbrella term” that includes multiple approaches and a large diversity of fields such as computer vision, natural language processing, voice recognition, etc. An interesting approach of AI is the ability of a machine to learn automatically. This field is called Machine Learning (ML) and it has revolutionized several technology domains such as data mining, classification of new astronomical structures, etc. [2, 3]. Mitchell [3] gives us a definition of what ML is: “*The concept of ML relates to the question of how to construct computer programs that automatically improve with experience*”.

Arnold et al. [4] report that one of the challenges of traditional machine learning, given a task of interest, is the handmade selection of the most relevant features. Automatic learning of these features using Neural Network (NN) can be used to tackle this issue. Deep Learning (DL) architectures are, based on this idea, composed of layers of non-linear processing. Since the previous decade, DL has made breakthroughs and its fields of application have multiplied [5]. The accuracy of the models using this paradigm has also significantly increased. For example, in 2015, the models submitted for the image classification task achieved on average 94.5% accuracy [6]. In comparison, the submitted models for the image classification task in ILSVRC2012 achieved on average 71.3% accuracy [7].

A type of DL model, Convolutional Neural Network (CNN), demonstrates its value in applications such as image and speech processing, thanks to its high performance [8]. However, the drive for improving the accuracy of such models is to increase their size, which comes at the price of a large computational and memory cost (billions of operations and millions of parameters) [9, 10]. For example, ResNet [11] requires 25.6M parameters and 11.3B floating-point operations for  $224 \times 224$  input images.

To implement and train such computationally-intensive models, dedicated accelerators seem to be more suitable than a general Central Processing Unit (CPU) [12]. Graphical Processing Unit (GPU), Application-Specific Integrated Circuit (ASIC) and Field-Programmable Gate Array (FPGA) can be used to improve the throughput and latency of CNN. Today, CNNs are executed on clusters of GPU and CPU because they provide the best throughput [13]. However, they are highly demanding in terms of energy and therefore not adapted for embedded systems with constrained power resources.

For these resource-constrained embedded systems, FPGA and ASIC seem to be a promising solution because they are more energy-efficient. This was demonstrated by Qasaimeh et al. [14]. FPGA has a lower energy consumption compared to GPU by several orders of magnitude, as the vision application’s pipeline complexity grows (which is the case for CNN). We can observe in Table 1 the FPGA’s energy reduction ratios to GPU for various vision application’s pipelines. Moreover, because the FPGA has the property of reconfigurability comparing to the ASIC, developing accelerators on FPGA is time and cost-effective [15]. Therefore, FPGA should be the platform to run CNN on mobile devices.

| Pipeline                | Energy/frame (mJ/f) |
|-------------------------|---------------------|
| Background Subtraction  | $1.74 \times$       |
| Color Segmentation      | $1.86 \times$       |
| Harris Corners Tracking | $3.94 \times$       |
| Stereo Block Matching   | $8.83 \times$       |

Table 1: FPGA’s Reduction Ratios to GPU, from [14]

Furthermore, while GPU is currently the most dominant platform to perform CNNs, trends confirm that in the future, FPGA will be more suitable to accelerate them, according to Nurvitadhi et al. [16]. Two reasons can explain that. First, the advances in FPGA technology show that the performance disparity between FPGA and GPU has lessened. Second, recent CNN trends use pruning and extremely quantized data types to reduce the size of the CNN. It is in favor of the FPGA because those two approaches are easier to handle on it. However, FPGA is resource-constrained (limited memory, I/O bandwidth, and computing resources). As a result, the challenge is to find a mapping between the computational and the execution model [17].

According to Abdelouahab et al. [18], there are 2 phases when executing a CNN. The first phase is the **training**, where the model learns using the back-propagation algorithm [19]. The second phase is the **inference**, where the model predicts the output of new data samples, using the learned model and the feed-forward algorithm [20]. We can see the different processes in Figure 1. Usually, CNNs are trained once on GPU or FPGA and inference is executed every time CNN has to process new input. So, most efforts need to be focused on accelerating the inference phase [18]. The core of this work is therefore aimed at proposing a way to accelerate the inference of a model on FPGA.

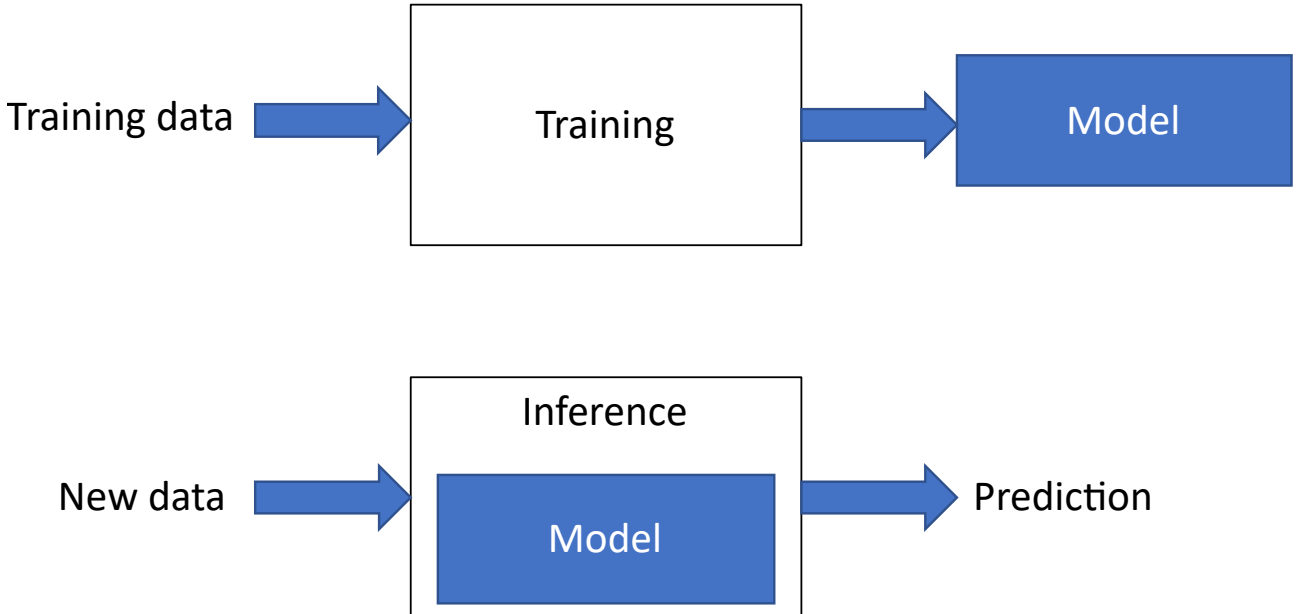


Figure 1: CNN execution phases, inspired by [16]

The inference can be accelerated using various optimizations. According to Abdelouahab et al. [18], these optimizations can be grouped into 3 main categories:

- **Algorithmic Optimizations:** the computational complexity of CNNs can be reduced by vectorizing the operations or using faster algorithms.
- **Datapath Optimization:** because of the limited resources on an FPGA, memory is often the bottleneck, and optimizing the memory management can increase the throughput.
- **CNN model Optimization:** an important issue of CNN is their computational complexity and hardware utilization. A solution is to trade accuracy for acceleration by using approximate computing.

**CNN model Optimization** is going to be the key point of this work because the CNN size and arithmetic complexity are the major issues when implementing a CNN on mobile devices [21]. One of these optimizations, named weight pruning, consists of removing all unnecessary weights of the network [18].

This work focus on weight pruning. The irregular data access and the custom data type of the non-pruned weights show it is suitable to implement pruning on FPGA. Therefore, this work concentrates on building an architecture using pruning on Depthwise Separable Convolution (DSC), an alternative way to perform convolution which uses fewer parameters and has a lesser computational complexity. The objective of this work is then to build an efficient architecture on FPGA implementing a sparse CNN network using DSC. To assess the benefits of the proposed pruning, we have chosen MobileNetV2 because it offers state of the art performance [22].

These work achievements are the following:

- A structured pruning scheme is proposed to reduce the number of parameters and the computational complexity of the DSC.
- To reduce memory usage, a compressed format is developed to store the non-pruned weights.
- An accelerator is designed to support the MobileNetV2 building block performing the sparse DSC.
- The obtained results show that the structured pruning scheme allows a reduction of the computational complexity and the number of parameters, and a faster inference by reducing the latency to execute the DSC.

## Structure of the thesis

This master thesis is composed of 3 chapters.

Chapter 1 details the background information about the two main topics of this thesis: CNN and FPGA. First, we explore the theory behind CNN and go deeper into its computational background in Section 1.1. Then, the concept and workflow of FPGA are explained in Section 1.2.

Using knowledge of Chapter 1, we explore in Chapter 2 the state of the art techniques on accelerating the inference on FPGA. We explore algorithmic and model optimizations to reduce

the cost of the convolution operation and the size of the models. The chapter also details datapath optimizations which reduce the inefficiency of the convolution on FPGA. We also review various FPGA architectures implementing DSC and pruning of standard convolution.

Chapter 3 uses the knowledge from Chapter 2 to design an efficient architecture using pruning and depthwise separable convolution. It also details the experimental setup to measure the performance of the developed architecture and discusses the obtained results.

Finally, the outcome of this work is shown in the Conclusion. A discussion on the methodology and future works is also presented.

# Chapter 1

## Background

In the frame of this thesis, knowledge about CNN and FPGA is required. The purpose of this chapter is then to detail all the background information.

First, Section 1.1 develops the theory behind CNN. We explore how such a model can be constructed, what are the state-of-the-art models and how can we train a CNN.

Finally, Section 1.2 explains what is a FPGA and what are its main components. We also detail the FPGA design flow.

### 1.1 Convolutional Neural Network

CNN is a type of NN specialized in analyzing visual imagery and natural language processing. CNNs are feedforward, sparsely connected NNs, and structured as a pipeline of layers [18]. It showed superior performances in different competitions related to Computer Vision and Image Processing [10]. Nowadays, CNNs are used in character and gesture recognition, video classification, face detection, etc [8]. This section aims at investigating the different aspects of a CNN.

First, the building blocks of a CNN are developed in Section 1.1.1 starting from its basic element, the *perceptron*. Then, we describe how to use multiple perceptrons to build layers that can perform complex functions. Finally, we detail the other types of layers required to improve the efficiency of the model, such as the pooling layer.

Then, Section 1.1.2 describes how to effectively combine these layers with a focus on state-of-the-art networks like AlexNet, VGG16, ResNet, and MobileNetV2.

Finally, Section 1.1.3 focuses on the training of CNN. We briefly explain the back-propagation algorithm, which consists of a forward-propagation and a back-propagation.

#### 1.1.1 Structure of a CNN

CNN is a pipeline of layers that can be stacked to form a network [18]. A layer is a high-level building block, which consists of a set of operations, weights, and non-linear functions. The output of one layer can be used as input of the next layer or as the final output of the network.

This section first introduces the simplest building element of a CNN: the perceptron. We describe

its different components and notably its activation function. Then, we can use this element to build more complex layers such as the fully connected and convolutional layer. Finally, we present the pooling layer which is used to reduce the spatial dimensions of the output of the previous layer.

## Perceptron

The perceptron is a computational model that was developed in 1957 by Rosenblatt et al. [23]. This computational model is inspired by the neurons which are the computational units of the human brain. A neuron receives information and when its threshold is reached, this information is released and transmitted to other neurons (the neuron “fires”) [23, 24].

The perceptron corresponds to the computational model of a neuron in a human brain. The perceptron is composed of  $N_{in}$  inputs  $\mathbf{x} = \{x_1, \dots, x_{N_{in}}\}$ ,  $N_{in}$  weights  $\mathbf{w}$ , and a bias  $b$ . Figure 1.1 illustrates the working principle of a perceptron. When it receives an input vector  $\mathbf{x}$ , a weighted sum of this vector is computed. If its result is higher than the threshold (controlled by  $b$ ), the perceptron is “activated” and its output becomes a non-zero value. Otherwise, the perceptron outputs a zero value. Equation (1.1) represents the mathematical formula of the perceptron where  $h$  is the activation function of the formula developed in Equation (1.2) [24].

$$h(\mathbf{x}|\mathbf{w}, b) = h\left(\sum_{i=1}^{N_{in}} x_i \cdot w_i + b\right) = h(\mathbf{w}^T \cdot \mathbf{x} + b) \quad (1.1)$$

$$h(\mathbf{w}^T \cdot \mathbf{x} + b) = \begin{cases} 1, & \text{if } \mathbf{w}^T \cdot \mathbf{x} + b > 0 \\ 0, & \text{Otherwise} \end{cases} \quad (1.2)$$

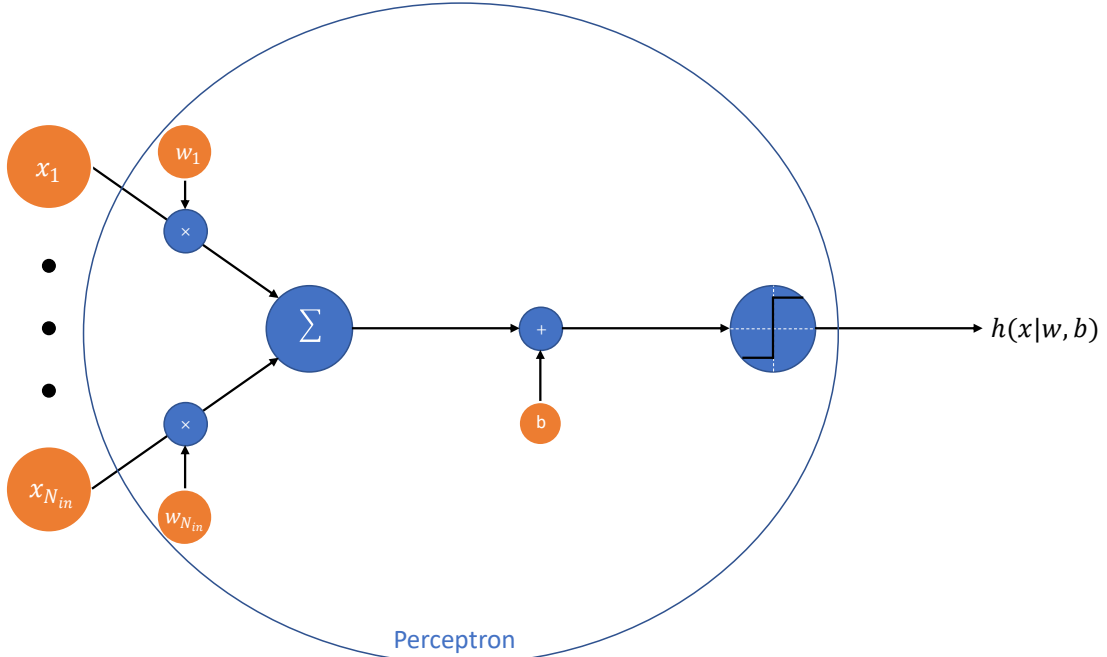


Figure 1.1: The Perceptron

As the perceptron computes a weighted sum, the perceptron can learn a task of interest by modifying those weights. However, this model is limited by the functions it can achieve. Later

section details how we can use multiple perceptrons to create a **fully-connected layer**, learning more complex functions.

## Activation functions

To decide if a perceptron is activated or not (if its threshold is reached), an activation function needs to be defined. To apply the back-propagation algorithm which makes CNN learn (will be further detailed in Section 1.1.3), this function needs to be differentiable [19]. Indeed, during the optimization of the network, the gradient of the activation function is computed. Therefore, Equation 1.2 cannot be used as an activation function because of its zero-value gradient. The learning does not converge and hence the CNN does not learn. Various activation functions were proposed with different properties, as illustrated in Figure 1.2. According to Khan et al. [10], the choice of an appropriate activation function can accelerate the learning phase and some activation functions have lower computational complexity than others [25].

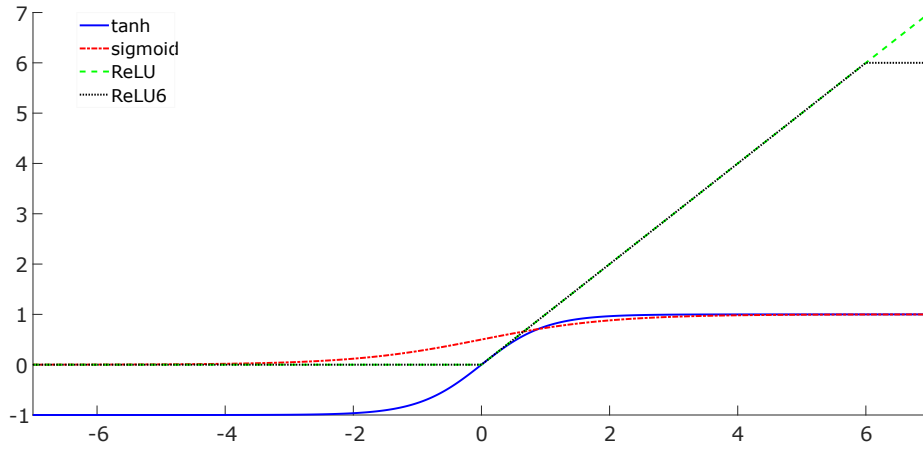


Figure 1.2: Activation functions

### Sigmoid and Hyperbolic Tangent (Tanh)

*Sigmoid* and *Tanh* are smooth activation functions that limit the  $\mathbb{R}$  domain into a narrower domain:  $[0, 1]$  for *Sigmoid*, and  $[-1, 1]$  for the *Tanh*, as shown in Figure 1.2 [24]. They can be described using Equations (1.3) and (1.4) [25].

$$h(x) = \frac{1}{1 + e^{-x}} \quad (1.3)$$

$$h(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (1.4)$$

However, they saturate at the asymptotes, which means that their gradient is close to 0 outside the input range  $[-4, 4]$  [26]. As the back-propagation algorithm (see Section 1.1.3) requires gradient multiplication, gradient far away from the output vanishes (close to 0) and deep models do not learn. It is the **vanishing gradient problem** [10, 24, 27, 28].

## ReLU and ReLU6

To overcome this vanishing gradient problem, the Rectified Linear Unit (ReLU) which is defined by Equation (1.5) was developed by Krizhevsky et al. [25].

$$h(x) = \max(0, x) \quad (1.5)$$

This function increases the learning and computational speed compared to the *Tanh* and *Sigmoid* functions. It also improves the efficiency of the propagation gradient (avoiding vanishing or exploding gradient) [18, 28]. However, some perceptrons can become inactive after learning. It means that their output is zero for all inputs. It is called the **dying neuron problem** which decreases the model capacity [24]. A solution would be to use a derivative of the ReLU activation function designed to solve this problem such as leaky ReLU for example [24, 28].

We can also use the dying neuron property to learn sparse features. It was done by Krizhevsky [29] by setting an upper bound to the ReLU output. For example, they chose to limit the output of the perceptron to 6, as described in Equation (1.6). This activation function is called ReLU6 because its output range is limited to  $[0, 6]$ .

$$h(x) = \max(0, x, 6) \quad (1.6)$$

ReLU6 has the advantage to be designed for fixed-point operations and quantization approaches (which will be described in Section 2.1.2), instead of floating-point operations. It means the number of bits for the integer part can be limited to 3 bits because the output  $\in [0, 6]$  (with 3 bit, we can represent numbers in the range  $[0, 2^3 - 1] = [0, 7]$ ). The accuracy of the model can thus be increased by assigning the other available bits to the decimal part. This property makes ReLU6 to be used in models that aim mobile and embedded platforms like MobileNet [30]. Indeed, fixed-point operations, especially on FPGA, are more efficient in terms of hardware utilization and power consumption [31]. In the frame of this thesis, the ReLU6 activation function is used.

## Fully Connected layer

The perceptron described in Section 1.1.1 can be considered as a linear classifier for which the decision boundary is a hyperplane defined in equation (1.7) [24].

$$b + w_1 \cdot x_1 + \dots + w_{n_{in}} \cdot x_{n_{in}} = 0 \quad (1.7)$$

The perceptron is therefore limited because it has only a linear decision boundary. For example, it means that the perceptron can learn the AND and OR Boolean functions but can not learn the XOR function [32]. To be able to build non-linear models, a topology of perceptrons needs to be used, as illustrated in Figure 1.3 [10]. Therefore, a fully connected layer used in CNN is a set of  $N_{in}$  inputs that are fed to  $N_{out}$  perceptrons. The  $N_{out}$  outputs are either fed to another fully connected layer or are used as the output of the CNN.

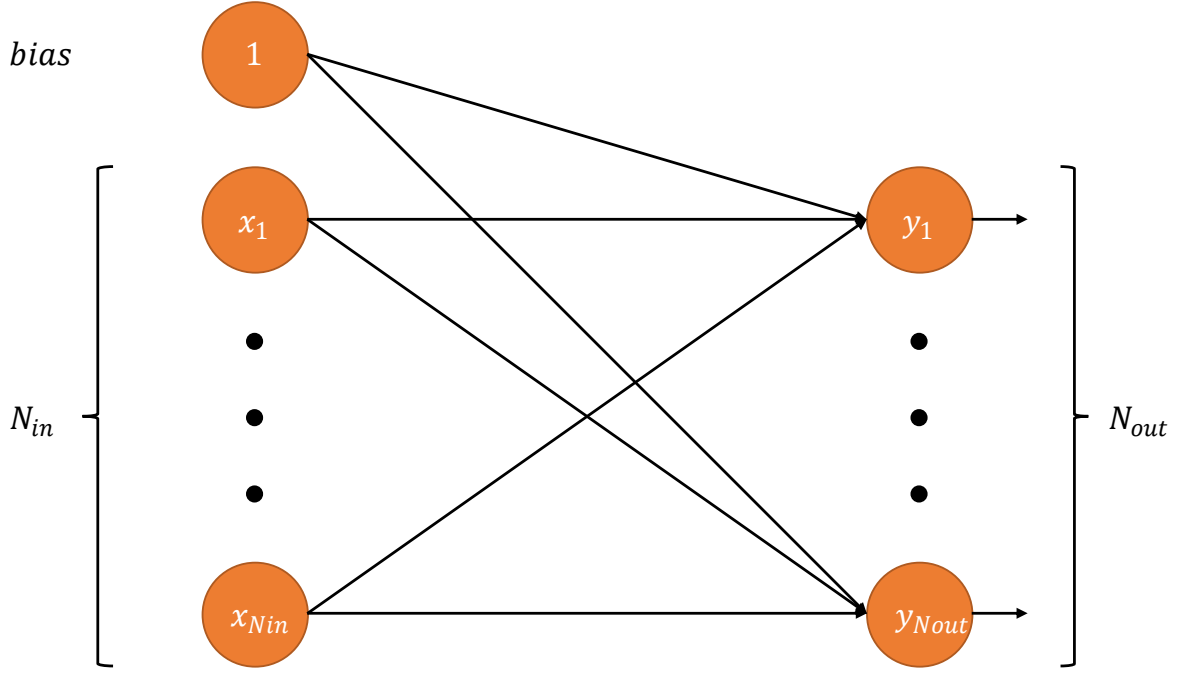


Figure 1.3: A fully connected layer

A fully connected layer is characterized by the number of neurons, activation function, and the value of weights. The output vector  $\mathbf{y}$  is defined by equation (1.8), where  $\mathbf{x}$  is the input vector ( $x_0 = 1$  corresponds to the biases),  $\mathbf{w}$  is the weights vector ( $w_*^i$  are the weights of the  $i$ th perceptron and  $w_0^*$  are the biases), and  $h$  is the activation function [18].

$$\mathbf{y} = h(\mathbf{w}^T \mathbf{x}) \quad \Leftrightarrow \quad \forall o \in \{1, \dots, N_{out}\} : y_o = h\left(\sum_{i=0}^{N_{in}} w_i^o \cdot x_i\right) \quad (1.8)$$

As we have seen that perceptrons can be used to construct a non-linear classifier, we see in the next section the main operation in the CNN: the **convolution**, which extracts the features from input images.

## Convolutional layer

As said previously, fully connected layers are non-linear classifiers. They take the extracted features of the input converted as a one-dimension vector [10]. As a result, the fully connected layers are usually placed at the end of CNNs. Therefore, the first layers extract features from the input to feed the fully connected layers [12]. Those extracting features layers are named **convolutional layers**.

The main operation in a CNN is the convolution which consists of extracting the features from the input image, also called input Feature Map (FM) [12, 33]. It is the main operation on a CNN, and it is the layer that gives the network its name. The first layers extract low-level features of the input FM while the deepest layers use these low-level features to build high-level ones [27].

The notation for describing the different entities of the layer is inspired by Ma et al. [34]. The input FM is characterized by 3 parameters:  $N_{ix}$  the width,  $N_{iy}$  the height, and  $N_{if}$  the depth.

As shown in Figure 1.4a this FM can be seen as successive layers of pixels forming a cuboid. Each layer is called a channel composed of  $N_{ix} \times N_{iy}$  pixels.

Then, the convolution layer is used to correlate this FM and a 4D weight filter to build an output FM, composed of high-level features [33]. This filter is composed of  $N_{of}$  kernels, where each kernel has size  $N_{kx} \times N_{ky} \times N_{if}$ , as can be seen in Figure 1.4b. The output FM is also characterized by three parameters: width  $N_{ox}$ , height  $N_{oy}$ , and depth  $N_{of}$ . Figure 1.4c illustrates the general layout of an output FM. The purpose of each kernel is to produce one channel of the output FM.

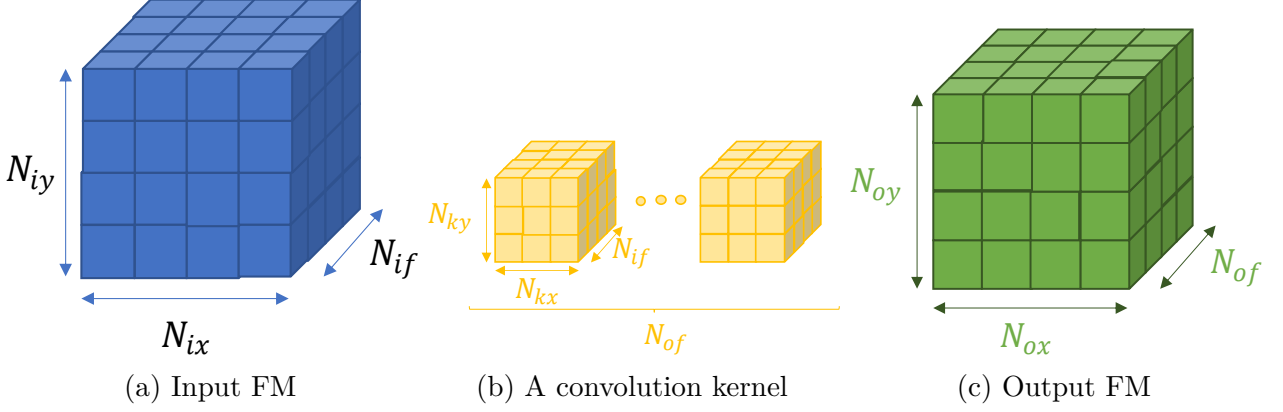


Figure 1.4: Volumes involved in the convolution operations

The convolution operation is illustrated in Figure 1.5. It can be considered as a specialized linear operation and is divided into five steps [24, 35]:

1. A window of the same size as a kernel extracts a chunk of pixels from the input FM.
2. The extracted chunk of pixels is element-wise multiplied with a kernel.
3. The output pixel is obtained by summing the products from the previous step.
4. The previous steps produce one output pixel of one channel. To compute one output channel, we simply slide the window on the input FM and repeat the first three steps. The output pixel at position  $(x, y)$  is obtained by the movement of the sliding window from the top left of the input FM.
5. We repeat the previous operations with each kernel of the weight filter to obtain the output FM.

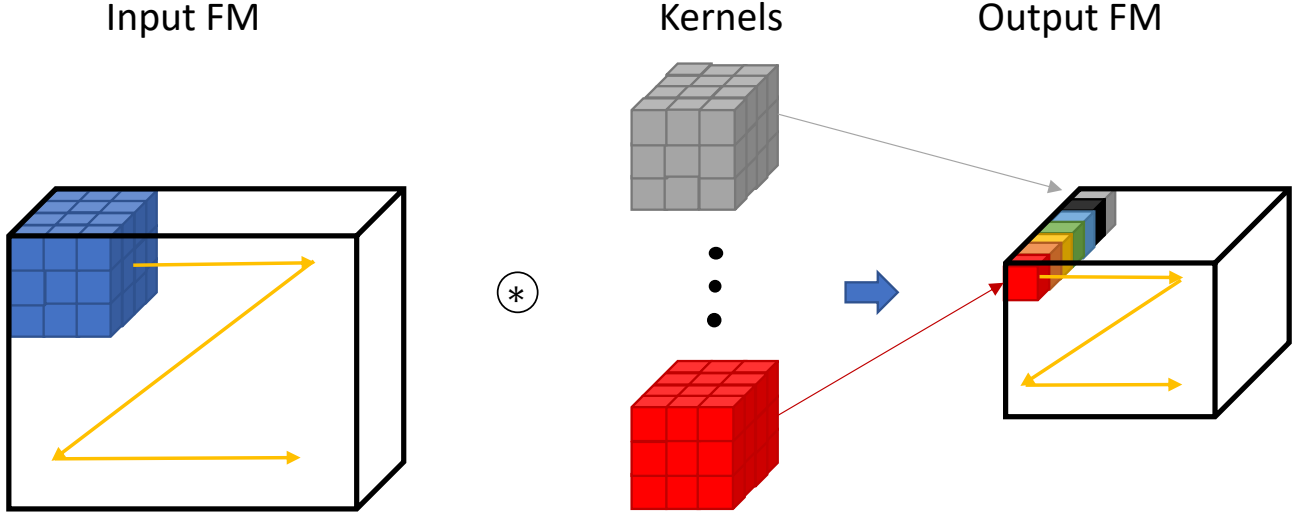


Figure 1.5: Convolution operation

Except for  $1 \times 1$  kernel, a spatial reduction between the input and output FM is observed, while the convolution can be used to increase the number of channels (if  $N_{if} < N_{of}$ ). It is explained by the fact that the sliding window movements can not go beyond the size of the input FM. Indeed, as the number of output pixels in one spatial axis is determined by the movement of the sliding window from the top left of the input FM, if the spatial dimensions of the two FM were equal, the sliding window extracts a chunk of data of spatial size  $N_{kx} \times N_{ky}$  centered around each pixel of the input FM. However, it would also mean that the sliding window can extract pixels that do not exist for the chunk of data centered on the edges of the input FM. Therefore, we can express the spatial reduction using equation (1.9) [34].

$$N_{o\{x,y\}} = N_{i\{x,y\}} - N_{k\{x,y\}} + 1 \quad (1.9)$$

*Padding* can be used to avoid this spatial reduction. It consists of increasing the spatial dimensions (width and height) of the input FM by inserting extra pixels (usually 0 value) around the edges of the input FM [12]. Using padding, the spatial dimension increase can be expressed using Equation (1.10), where  $N_{i'\{x,y\}}$  are the spatial dimensions of the input FM with padding applied. If we replace  $N_{i\{x,y\}}$  by  $N_{i'\{x,y\}}$  in Equation (1.9), there is no reduction in the dimensions.

$$N_{i'\{x,y\}} = N_{i\{x,y\}} + N_{k\{x,y\}} - 1 \quad (1.10)$$

The sliding movement of the window on the input FM is called the *stride* and is initially set to 1. The stride can be increased to decrease the spatial dimensions of the output FM [12]. For example, if padding and a stride of 2 are used:  $\frac{N_{ix}}{N_{ox}} = \frac{N_{iy}}{N_{oy}} = \frac{1}{2}$ , and hence the output FM has 4 times fewer pixels. Equation (1.11) represents the spatial reduction during convolution as a function of the stride [34]. There are also other ways to reduce the spatial dimensions of the output FM. One of them, named *pooling*, will be developed in the next section.

$$N_{o\{x,y\}} = \left\lfloor \frac{N_{i\{x,y\}}}{S} \right\rfloor - N_{k\{x,y\}} + 1 \quad (1.11)$$

In summary, the whole convolution operation can be mathematically defined by Equation (1.12)

[18].

$$\begin{aligned}
& \forall ox \in \{1, \dots, N_{ox}\}, oy \in \{1, \dots, N_{oy}\}, of \in \{1, \dots, N_{of}\} : \\
& FM_O[ox, oy, of] = \sum_{if=1}^{N_{if}} \sum_{kx=1}^{N_{kx}} \sum_{ky=1}^{N_{ky}} \\
& \quad FM_I \left[ ox \cdot S + kx - \left\lfloor \frac{N_{kx}}{2} \right\rfloor, oy \cdot S + ky - \left\lfloor \frac{N_{ky}}{2} \right\rfloor, if \right] \cdot W_{if}^{of}[kx, ky]
\end{aligned} \tag{1.12}$$

According to Goodfellow et al. [27], the convolutional layer shows several advantages compared to the fully connected layer. Indeed, the convolutional layer is sparsely connected, which means that there are fewer connections between the different layers. The convolutional layer is then easier to train and the convolution keeps spatial information.

Moreover, it has also parameter sharing: the same kernel is used to compute one channel of the output FM and fewer parameters are needed. To illustrate this weight reduction, in AlexNet [25], 94% of the weights are used in the fully connected layers. However, convolution is a computationally heavy operation. For example, in VGG-11, 98.2% of the operations are done in the convolution layers [36].

Furthermore, if the input FM is a vector ( $N_{ix} = N_{iy} = 1$ ), then it means that the convolution and the fully connected layer perform the same operation.

As the convolution has a huge arithmetic complexity, we see in the following section an alternate way to perform convolution while reducing its complexity.

## Depthwise Separable Convolution

Depthwise Separable Convolution (DSC) was first introduced by Sifre et al. [37]. According to Chollet [38], “A *depthwise separable convolution* consists in a **depthwise convolution**, i.e. a spatial convolution performed independently over each channel of an input, followed by a **pointwise convolution**, i.e. a  $1 \times 1$  convolution, projecting the channel’s output by the depthwise convolution onto a new channel space”.

It means that the DSC is composed of a depthwise convolution followed by a pointwise convolution, as illustrated in Figure 1.6. This alternative form of convolution was developed to reduce efficiently the number of weights and the arithmetic complexity, in exchange for a limited loss of accuracy [12]. As a result, the DSC has significantly fewer parameters and operations to the standard convolution, which was described previously. Equations (1.13) and (1.14) are used to express the reduction factors on weights and operations, where  $F_*$  are the reduction factors,  $W_{sc}$  and  $O_{sc}$  (resp.  $W_{dsc}$  and  $O_{dsc}$ ) are the weights and operations required for a standard convolution (resp. DSC) [12, 39].

$$F_w = \frac{W_{dsc}}{W_{sc}} = \frac{N_{kx} \times N_{ky} \times N_{if} + N_{if} \times N_{of}}{N_{kx} \times N_{ky} \times N_{if} \times N_{of}} = \frac{1}{N_{of}} + \frac{1}{N_{kx} \times N_{ky}} \tag{1.13}$$

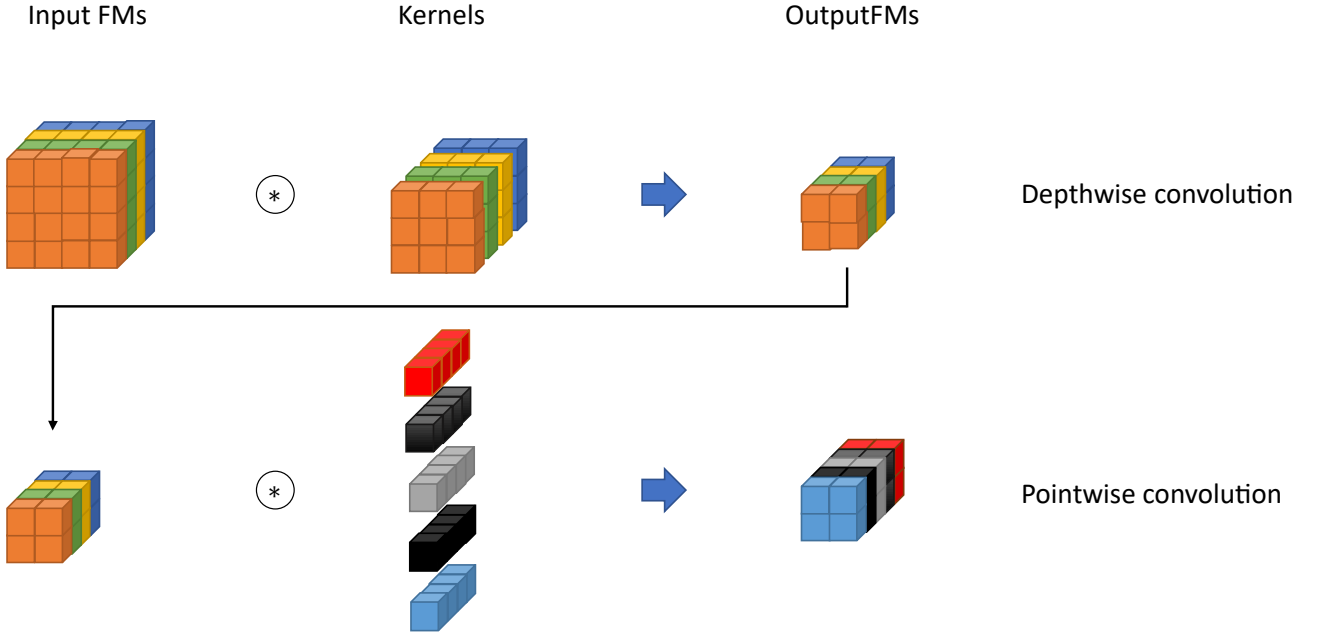


Figure 1.6: Depthwise Separable Convolution, inspired by [12, 39]

$$\begin{aligned}
 F_o &= \frac{O_{dsc}}{O_{sc}} = \frac{N_{kx} \times N_{ky} \times N_{if} \times N_{ox} \times N_{oy} + N_{if} \times N_{of} \times N_{ox} \times N_{oy}}{N_{kx} \times N_{ky} \times N_{if} \times N_{of} \times N_{ox} \times N_{oy}} \\
 &= \frac{1}{N_{of}} + \frac{1}{N_{kx} \times N_{ky}}
 \end{aligned} \tag{1.14}$$

Using Equation (1.13), Equation (1.14), and  $3 \times 3$  kernels, the reduction factors of computational complexity and weights are about 9 times (in comparison with the standard convolution) [40].

## Pooling

A Pooling layer is used to replace the output of the previous layer by a summary statistics of this output [27]. It is usually inserted between two successive convolutional layers to modify the output further.

Its goal is first to make the output approximately invariant to small translations. Second, it decreases the spatial size of each output FM. It means that we need less memory to store the parameters [27]. In summary, it reduces the number of parameters and computation of the network while increasing the receptive field [8, 24].

The pooling layer divides each FM into regions of size  $K \times K$  and outputs one pixel from each region. This way, the number of channels is kept constant while their spatial dimension is reduced by factor  $K$ . Various pooling functions exist, but the most common form uses filters of size  $2 \times 2$ . For example, the MAX pooling selects the highest value pixel from 4 samples, and the AVG pooling selects their average (meaning a 75% reduction of the pixels) [41]. Figure 1.7 illustrates an example of a pooling layer using MAX and AVG pooling.

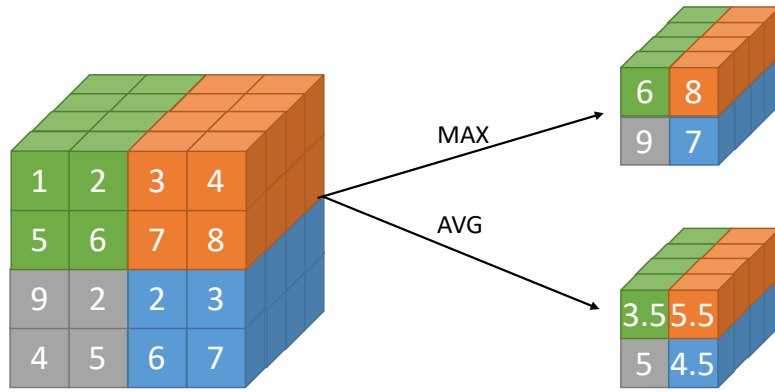


Figure 1.7: Examples of pooling layers

## Summary

The summary of this section is provided in Figure 1.8. First, an input image is fed to the CNN. Then, three different kinds of layers can be used:

1. **The convolutional layer**, which extracts the features from the input.
2. **The pooling layers**, which summarizes the output of the previous layer.
3. **The fully connected layer**, which is a non-linear classifier.

A typical CNN is composed of two parts, which are built from stacking the previously mentioned layers [24]:

1. **The feature extractor** part, composed of blocks made of convolutional, activation, and pooling layers.
2. **The classifier** part, composed only of fully connected layers.

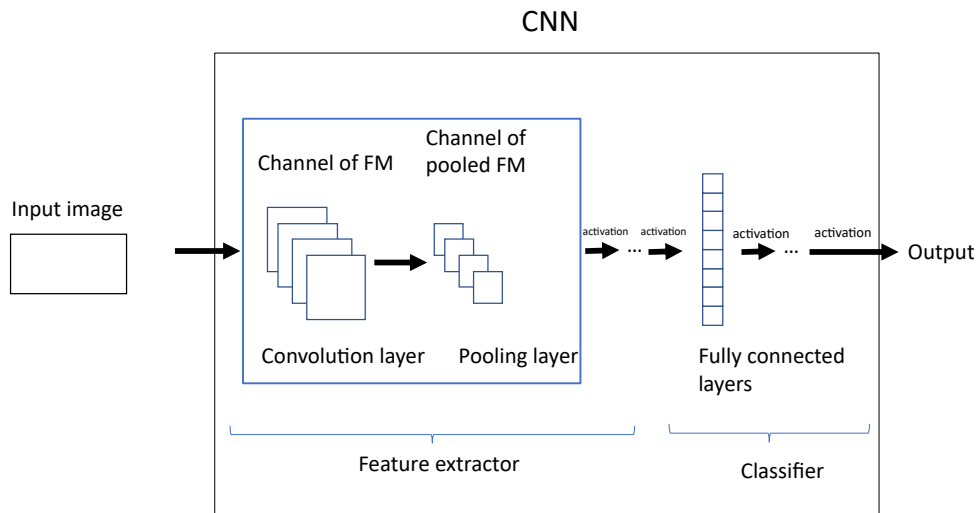


Figure 1.8: Working principle of a CNN and the different layers which composing it

### 1.1.2 CNN Models

After the analysis of the CNN structure in Section 1.1.1, this section details some of the well-known CNN networks. They are examples of how to construct an efficient CNN.

## AlexNet

AlexNet is a famous CNN which was developed in 2012 by Krizhevsky et al. [25]. It was a breakthrough in the deep CNN field and this model won the ImageNet competition. Krizhevsky et al. used some parameter optimizations and made the model deeper to considerably improve the learning ability of the CNN [10]. It is composed of 5 convolutional layers and 3 fully connected layers, as seen in Figure 1.9. Each convolutional layer has a ReLU activation function and it uses pooling.

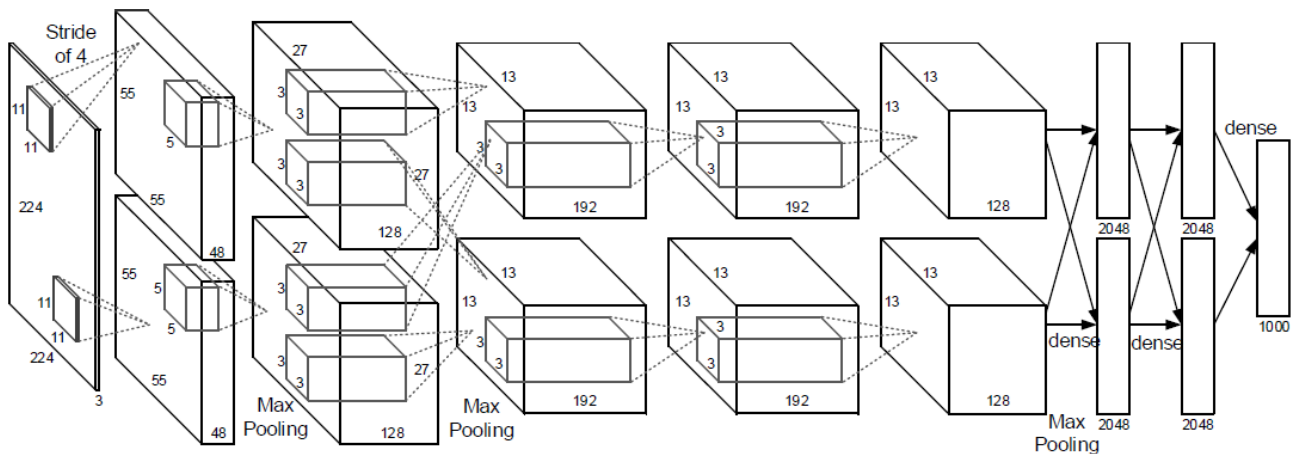


Figure 1.9: An illustration of the architecture of AlexNet, from [25]

## VGG

After the success of AlexNet in 2012, research was carried out to reduce the computational complexity while keeping accuracy. In 2014, VGG, a deeper variant of AlexNet was developed by Simonyan et al. [42]. It is composed of 5 groups of convolutional layers, where the number of layers depends on the version of VGG. It won the localization in the ImageNet challenge in 2014 [42]. An illustration is provided in Figure 1.10. This level of depth was possible thanks to the application of very small ( $3 \times 3$ ) convolution kernels. They allow a larger receptive field with fewer parameters and more non-linearities than a larger kernel [24]. However, it has a high memory request. 100MB per image needs to be stored in all FMs for the forward propagation [24].

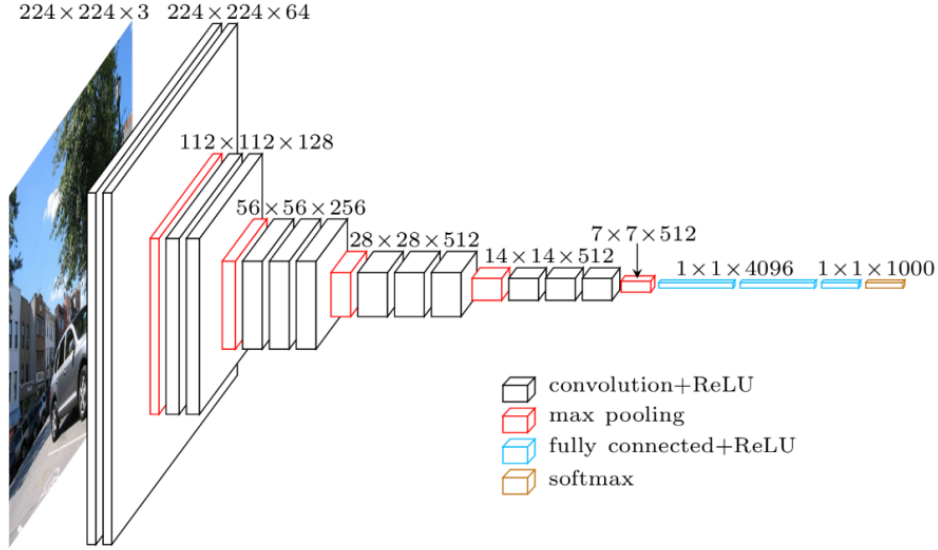


Figure 1.10: An illustration of the architecture of VGG16, from [42]

## ResNet

He et al. [11] developed ResNet in 2015. It is a very deep network which can contain from 50 to 1000 convolutional layers. It is composed of structures that are more complex and irregular than in the networks described previously. He et al. [11] showed that an increase in the depth of the network does not lead to an improvement in the performance. Indeed, beyond a certain amount of layers, a continuous increase of the depth leads to the degradation of the accuracy. This is not due to overfitting. It is because the deeper models are harder to optimize than shallower ones (vanishing gradient described in Section 1.1.3) [24].

However, deeper networks should at least have similar or better performance than shallower ones. Indeed, let's compare a deep and a shallow network. If the deeper network is composed of the same layers than the shallower one and that the other added layers are just identity mapping, then the network should have the same performance as the shallower one. Therefore, a deeper network should not have worse performance than shallower ones [24].

To overcome this issue, He et al. [11] introduced an *identity shortcut connection* which skips one or more layers and set weights to the identity, as can be seen in Figure 1.11. These weights associated with the shortcut connection can be used to learn a residual  $\mathcal{F}(x)$  to improve the solution. The performance of this very deep network allowed it to win the 2015 ILSVR for both localization and classification.

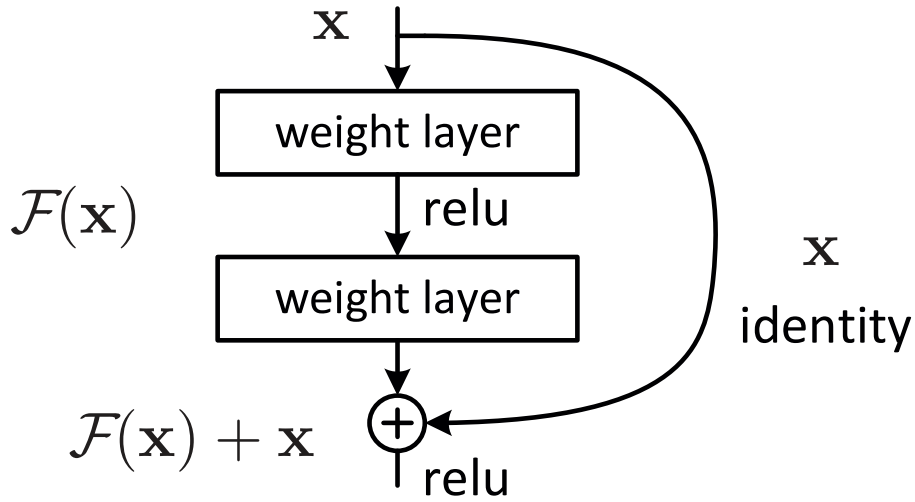


Figure 1.11: ResNet building block, from [11]

## MobileNetV2

According to Cheng et al. [21], the performance of CNN in recent years became outstanding. These improvements came at the cost of storage and computational complexity. It is not an issue for the CNN training phase, because the GPU and CPU also gained in computational units and memory. However, for the inference phase, the computational complexity and the storage requirements are way beyond the capabilities of most of the embedded applications and mobile devices such as FPGA [21].

- The enormous computational complexity of CNNs makes it difficult to deploy on real-time applications and it consumes battery power.
- The large number of parameters of CNNs consumes considerable storage and run-time memory.

To overcome this issue, Sandler et al. [22] introduced a network called MobileNetV2 in 2018. It is specifically developed for constrained environments. First, the size and number of operations are decreased thanks to DSC (see Section 1.1.1) and thanks to a new type of layer *Bottleneck residual block* that performs the bottleneck convolution, which can be observed in Figure 1.12 and Table 1.1. This layer is composed of a  $1 \times 1$  convolution to expand the number of the input FM channels by a factor  $t$  ( $N_{intf} = t \times N_{if}$ ). It is then followed by a DSC. The purpose of the first convolution layer which increases the number of channels is supposed to counterbalance the loss of information that occurred by the ReLU activation. They also added a skip connection to build a network of great depth, and the last convolution has a linear activation function [22]. Moreover, the bottleneck convolution uses padding and the output FM spatial dimensions are reduced using the stride  $S$ .

| Input                                                               | Operator                     | Output                                                              |
|---------------------------------------------------------------------|------------------------------|---------------------------------------------------------------------|
| $N_{ix} \times N_{iy} \times N_{if}$                                | $1 \times 1$ conv2d, ReLU6   | $N_{ix} \times N_{iy} \times (t \times N_{if})$                     |
| $N_{ix} \times N_{iy} \times (t \times N_{if})$                     | $3 \times 3$ dwse s=s, ReLU6 | $\frac{N_{ix}}{s} \times \frac{N_{iy}}{s} \times (t \times N_{if})$ |
| $\frac{N_{ix}}{s} \times \frac{N_{iy}}{s} \times (t \times N_{if})$ | $1 \times 1$ conv2d          | $\frac{N_{ix}}{s} \times \frac{N_{iy}}{s} \times N_{of}$            |

Table 1.1: Bottleneck convolution, from [22]

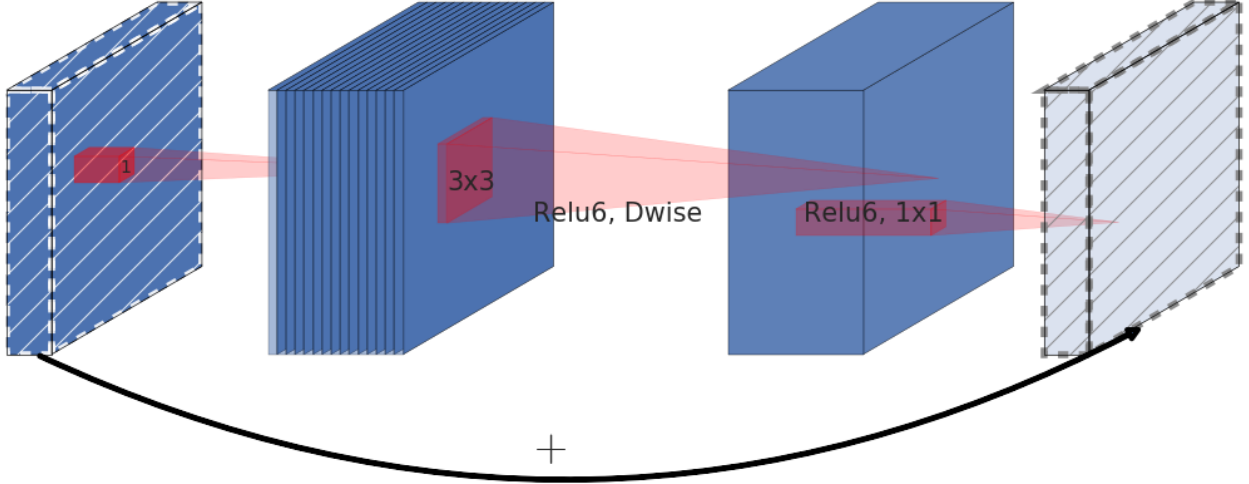


Figure 1.12: Bottleneck residual block, from [22]

MobileNetV2 is also composed of standard convolution, pooling, and  $1 \times 1$  convolution layers. More information about the structure of MobileNetV2 can be found in Appendix A. In the frame of this thesis, we design an FPGA-based accelerator that implements MobileNetV2.

### 1.1.3 Training

The previous section described the common building blocks present in CNNs and how to design a CNN. This section aims at detailing how a network improves automatically its performance for a specific task. This is called *learning*.

The learning phase starts after designing the network. However, before the model can learn, we have to initialize the weights. The most common initialization is a random Gaussian distribution [43]. However, the learning phase is affected by the initial values of the weights. If it is too small the network does not learn or if it is too large it might take a very long time to converge. Different initializations were suggested to improve the learning phase: Xavier initialization [26] and He initialization [43].

When values are assigned to the weights, the back-propagation algorithm can be performed to improve the efficiency of the network. The back-propagation algorithm is composed of two steps: the forward-propagation and the back-propagation. In the following sections, we briefly review each of them.

## Forward-propagation

The first step of the backpropagation algorithm is the forward propagation. During this step, the training data are used as input of the model, and each input is propagated through the network using the initialized weights. A vector of outputs is produced and is compared using a loss function (for example mean-square error defined by Equation (1.15)) with the label of the input (target  $\mathbf{t}$ ) [24]. The weights are then adjusted to minimize the loss function. This can be done using the algorithms found in the next section. When the model is trained, the inference consists only of the forward-propagation [18].

$$L(\mathbf{x}, \mathbf{w}) = \sum_{i=1}^N (t_i - p(x_i, \mathbf{w}))^2 \quad (1.15)$$

## Back-propagation

According to Ruder [44], gradient descent optimization algorithms are the most common ones used to perform optimizations on NN. These algorithms derive from the idea of gradient descent. As said in the previous section, gradient descent is a way to minimize the loss function to parametrize the weights of the model. Equation (1.16) defines the gradient descent algorithm, where  $\eta$  is the learning rate, a positive scalar determining the size of the step in the direction minimizing the gradient [27, 44]. If we update the weight in the opposite direction of the gradient, we can reach a local minimum.

$$\mathbf{w} = \mathbf{w} - \eta \frac{\partial L(\mathbf{x}, \mathbf{w})}{\partial \mathbf{w}} \quad (1.16)$$

**The original gradient descent** or **batch gradient descent** computes the gradient using the whole dataset (the batch) [24, 44]. Equation (1.17) is used to compute this batch gradient. However, this might be impossible to do in practice if the dataset is too large. Indeed, the gradient of the whole dataset has to be computed for each update. Therefore, variations of this algorithm have been proposed to make the gradient descent practical.

$$\frac{\partial L(\mathbf{x}, \mathbf{w})}{\partial \mathbf{w}} = \frac{1}{N_{in}} \sum_{i=0}^{N_{in}} \frac{\partial L(x_i, \mathbf{w})}{\partial \mathbf{w}} \quad (1.17)$$

**Stochastic gradient descent**, instead of using the entire dataset, performs the gradient descent algorithm on each training sample of the whole dataset [44, 24]. Equation (1.18) computes the stochastic gradient descent. It avoids the redundant computation of the batch gradient descent. It learns faster and can reach better local minima, however, it is complicated to find the global minimum.

$$\frac{\partial L(\mathbf{x}, \mathbf{w})}{\partial \mathbf{w}} = \frac{\partial L(x_i, \mathbf{w})}{\partial \mathbf{w}} \quad (1.18)$$

**Mini-batch gradient descent** is a trade-off between the two previous approaches [44, 24]. It performs an update of the weights for every mini-batch of the whole dataset. Equation (1.19) computes the mini-batch gradient descent, where  $N$  is the number of training samples. It shows

a better convergence than stochastic gradient descent by reducing the variance and has fewer computations than batch gradient descent.

$$\frac{\partial L(\mathbf{x}, \mathbf{w})}{\partial \mathbf{w}} = \frac{1}{N} \sum_{i=0}^{N-1} \frac{\partial L(x_i, \mathbf{w})}{\partial \mathbf{w}} \quad (1.19)$$

## 1.2 Field Programmable Gate Array (FPGA)

As developed in the Introduction, the focus on this study is on accelerating the inference stage on FPGA. According to Mittal et al. [45], FPGAs provide higher energy efficiency than both GPUs and CPU and higher performance than CPUs. Besides, FPGAs are very flexible hardware that can be tailored for each particular NN [46]. Furthermore, we can expect in the future that FPGAs will provide higher performance for performing CNNs than GPU [16].

According to Harris et al. [47], the definition of an FPGA is: “a *Field-Programmable Gate Array (FPGA)* is an array of reconfigurable gates”. The characteristic of this platform is that it can implement combinational and sequential logic but also multilevel functions. An FPGA also integrates built-in multipliers, high-speed I/Os, data converters, large RAM arrays, and processors.

An illustration of the general FPGA layout is provided in Figure 1.13. An FPGA consists of an array of configurable logic elements (LE), also called configurable logic blocks (CLB) [47]. Each LE can be configured to execute either combinational or sequential logic. Each LE is connected to other LEs to form an array. To interface with the outside world, the array is connected to a set of input/output elements (IOE).

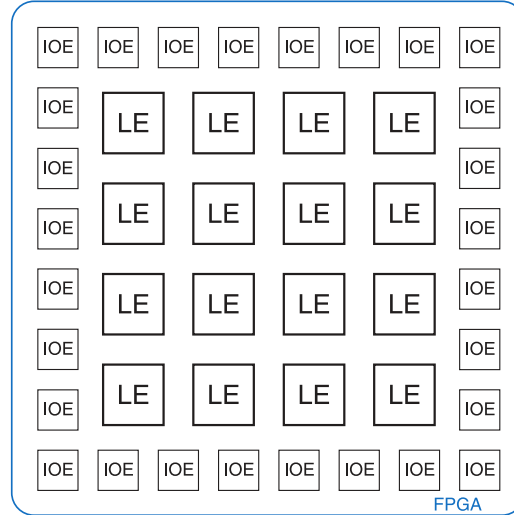


Figure 1.13: General FPGA layout, from [47]

Designs can be implemented on an FPGA using a software programming tool that can use either a Hardware Description Language (HDL) or a schematic. An HDL is a language that gives the specifications of digital design. The interest in using this language is the fast development cycle compared to a schematic, due to the higher abstraction level and the gate optimizations

[47]. The most popular HDL are *VHDL* and *SystemVerilog* which mostly differ on syntax. The architecture of this work is implemented in *SystemVerilog*.

The design flow of an FPGA can be described as follows [48]. First, the programmer creates the design of interest using either a schematic or HDL. Second, the design is synthesized, which means that the software determines how to configure the components of the FPGA to execute the specified function. Third, the software assesses if the design is correct using a functional simulation. If it is correct, the software fits the element into the FPGA and does the **timing analysis**. This step simulates the real delay to check if the timing requirements are met. Finally, the design is programmed into the FPGA when the timing analysis simulation is correct.

To perform a functional simulation of the proposed Device Under Test (DUT), we can use a *testbench* [47]. According to Harris et al. [47], “*The testbench contains statements to apply inputs to the DUT and ideally, to check that the correct outputs are produced*”. However, this practice is tedious and error-prone. Instead, we can use a *self-checking testbench*. In the frame of this thesis, we chose an alternative approach. Since we have to verify a huge number of outputs, the testbench produces instead an *output file*, which contains all outputs produced by the DUT. Therefore, we can compare the obtained results with the expected ones using a programming language (*Python* has been used).

#### Conclusions about the Background:

We have shown in this chapter what is a CNN, how to build, and train it. We have also seen that the most performing models have a huge computational complexity and memory utilization, which limits the implementation of such models on mobile platforms, such as FPGA. We have also seen what is an FPGA, what are its components, and how to construct a hardware design on it.

# Chapter 2

## State of the Art

Chapter Background detailed all the background information required to understand the theory behind CNN and FPGA. This chapter aims at reviewing the main optimization techniques known in the literature to implement an efficient CNN accelerator on FPGA, to achieve the objective of this thesis.

Section 2.1 details optimization techniques to modify the computational model of the CNN allowing an efficient inference on FPGA. Section 2.1.1 is focused on reducing the algorithmic complexity of the CNN by optimizing the convolution operation while Section 2.1.2 covers techniques to trade accuracy for a reduction of the number of parameters and algorithmic complexity.

Section 2.2 is focused on studying efficient ways to handle a CNN on FPGA. First, we review how a general CNN is implemented while reducing inefficiencies. Then, we look at architectures that successfully implement DSC in Section 2.2.2, and pruning in Section 2.2.3.

### 2.1 CNN optimizations for FPGA

Section 1.1.2 highlighted the issues experienced when implementing the inference phase on mobile devices such as FPGA. Indeed, the computational complexity and the storage utilization required by the most performing models are beyond the FPGA resource capability. Therefore, this section explores the state-of-the-art approaches to reduce the arithmetic complexity and hardware utilization to accelerate the inference stage on FPGA. Section 2.1.1 details how to efficiently optimize the convolution operation on FPGA. Section 2.1.2 covers techniques to reduce the size of the model and hence the number of parameters and operations required.

#### 2.1.1 Algorithmic Optimizations

According to Shawahna et al. [8], 90% of computation time in CNN is consumed by the convolution operation. Therefore, decreasing the number of operations in a convolution will reduce the computational complexity of CNN. This section investigates algorithms that accelerate the convolution operation.

## GEMM

General Matrix Multiplication (GEMM) is commonly used to process CNN on CPU and GPU [18]. The GEMM converts the convolution operation as a matrix-vector multiplication. In other words, a mini-batch of input FM and kernels are transformed into two 2D matrices, as illustrated in Figure 2.1. It is done in such a way that the multiplication between those two matrices gives the same result as a standard convolution.

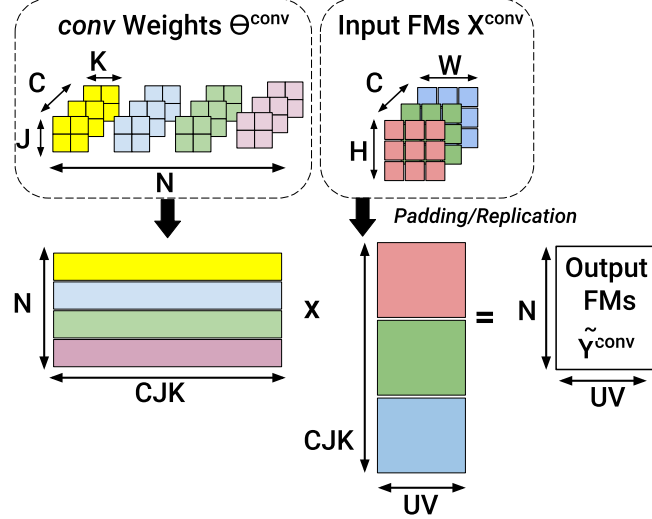


Figure 2.1: GEMM-based processing of a standard convolution layer, from [18]

The advantage of this method becomes more significant as the size of the mini-batch grows [18]. However, this approach is not appropriate for FPGA. Indeed, Zhu et al. [35] and Sze et al. [49] pointed out that the FMs, when converted to a vector, must be copied multiple times. It leads to huge memory storage requirements, memory footprint, and complex memory management access patterns. Furthermore, it does not reduce the arithmetic complexity of the convolution [50].

## Fast algorithms for convolution

As mentioned previously, the GEMM can accelerate the convolution but it does not reduce the number of operations. Therefore, we can investigate algorithms that aim at computing the convolution operation with fewer operations, named **fast convolution algorithms**. According to Liang et al. [50], there exist two fast convolution algorithms that perform efficiently the 2D convolution: Winograd minimal filter algorithm and Fast Fourier Transform (FFT).

The idea behind these fast convolution algorithms is to transform a kernel and a tile of input into a target domain, where the convolution corresponds to an element-wise multiplication in that domain [18]. In the end, we can obtain the corresponding tile of output by applying the inverse transformation on the product tile. As a result, the fast algorithm produces a tile of output (rather than a single pixel) and reduces the total number of operations by computing an element-wise multiplication (rather than a multiplication followed by an addition).

Therefore, both fast convolutions can be described by a common formula, as expressed in

Equation (2.1), where  $T_o$  (resp.  $T_I$ ) is the output (resp. the input) tile,  $K$  is the kernel,  $\mathcal{H}$  (resp.  $\mathcal{H}^{-1}$ ) is the domain transformation (resp. the inverse transformation) required to perform the algorithms and  $\odot$  is the element-wise multiplication [50].

$$T_o = \mathcal{H}^{-1} [ \mathcal{H}(T_I) \odot \mathcal{H}(K) ] \quad (2.1)$$

### Winograd-based convolution

The Winograd minimal filter algorithm was first introduced by Winograd [51]. According to Equation (2.1) and Abdelouahab et al. [18], the algorithm can be defined using the following steps:

- An input FM tile  $T_I$  of size  $(T_{ix} \times T_{iy})$  is pre-processed:

$$\mathcal{H}(T_I) = \mathbf{G}^T T_I \mathbf{G}$$

- In the same way, the kernel  $K$ , of size  $(K_x \times K_y)$  is also transformed:

$$\mathcal{H}(K) = \mathbf{B}^T K \mathbf{B}$$

- The inverse transformation function is then applied to obtain the output tile:

$$\mathcal{H}^{-1}(E) = \mathbf{A}^T E \mathbf{G}$$

Where  $E$  is the result of the element-wise multiplication of the two transformed tiles.

$$E = \mathcal{H}(T_I) \odot \mathcal{H}(K)$$

As a result, the output tile  $T_o$ , of the Winograd Filtering algorithm denoted  $F(T_{ix} \times T_{iy}, K_x \times K_y)$ , is computed using Equation (2.2) [51]. The advantage of using the Winograd-based convolution is that the transformation matrices  $\mathbf{G}$ ,  $\mathbf{B}$ ,  $\mathbf{A}$  can be computed off-line once  $(T_{ix}, T_{iy}, K_x, K_y)$  are fixed, using the Winograd Algorithm [51]. Therefore, these domain transform operations become multiplications with a constant and can be optimized on FPGA [50].

$$T_o = \mathbf{A}^T [ \mathbf{G}^T T_I \mathbf{G} \odot \mathbf{B}^T K \mathbf{B} ] \mathbf{A} \quad (2.2)$$

According to Winograd [51], the Winograd convolution reduces the number of multiplications at the cost of an increase in the number of additions. The gain obtained is defined in Equation (2.3) [51]. For example, if  $T_{ix} = T_{iy} = 2$  and  $N_{kx} = N_{ky} = 3$ , the complexity reduction factor  $F_{multiplication}$  equals 2.25 [52].

$$F_{multiplication} = \frac{T_{ix} \times T_{iy} \times N_{kx} \times N_{ky}}{(T_{ix} + N_{kx} - 1) \times (T_{iy} + N_{ky} - 1)} \quad (2.3)$$

### FFT-based convolution

The FFT, introduced by Cooley et al. [53], is an algorithm used to compute the discrete Fourier transform. As presented earlier, once the tile and kernel are in the frequency-domain, the operation corresponding to the convolution is the element-wise multiplication. Therefore, we can use the FFT to perform the convolution operation described in Equation (2.1), where [50]:

- $\mathcal{H}(\ast) = FFT(\ast)$
- $\mathcal{H}^{-1}(\ast) = IFFT(\ast)$

Using FFT, the computational complexity factor can be reduced from  $O(N_{ix} \times N_{iy} \times N_{kx} \times N_{ky})$  to  $O(N_{i\{x,y\}} \log_2(N_{k\{x,y\}}))$  [54].

## Comparison between the two fast convolution algorithms

Winograd-based convolution seems to be a preferred way to perform fast convolution. Indeed, Lavin et al. [52] demonstrated that Winograd convolution is more efficient when the kernel and the stride are small ( $K_{\ast} \leq 3$  for the kernel) while FFT-based convolution is more adequate when the kernel is large [55, 56]. However, current CNN trends use small kernels [22, 50] and so, the Winograd convolution.

Moreover, there are examples of successful FPGA implementations of Winograd algorithm: Liang et al. [50] and Aydonat et al. [57] used Winograd transform and reduced their computational complexity by around 50%. However, the Winograd algorithm also increased bandwidth utilization [58].

Yet, Liang et al. [50], Chitsaz et al. [56], and Zeng et al. [59] worked on the optimization of the frequency-domain convolution on FPGA with small filter by using Overlap-and-Add (OaA) technique [54] to lower the number of operations and increase the data parallelism. However, according to Liang et al. [50] and Podili et al. [60], FFT-based convolution still requires more memory, bandwidth, and logic resources (additions and multiplications) than Winograd. Furthermore, Zhang et al. [61] pointed out that performing FFT convolution with  $1 \times 1$  kernel is inefficient, and they performed it in time-domain only.

### 2.1.2 Model Optimizations

As mentioned previously, the major issues, when implementing a CNN on an FPGA, are the CNN size and its computational complexity. Research was done to develop techniques tackling those two issues by directly modifying the CNN architecture. Nurvitadhi et al. [16] believe that sparsity exploitation and extremely compact data types will become the norm in next-generation CNNs.

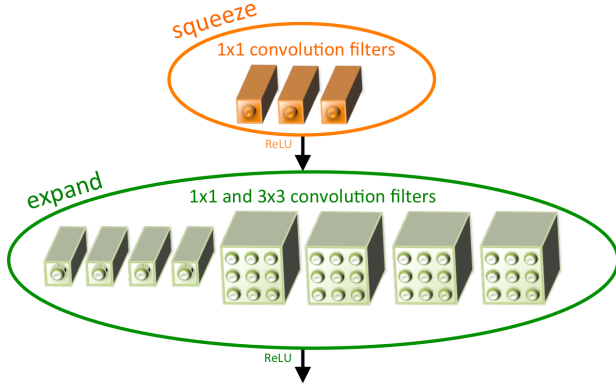
## Efficient Model Design

Section 1.1.2 presented several state-of-the-art models. However, those models (except MobileNetV2) were designed to provide the highest performance possible but did not consider the implementation of such models on mobile and embedded devices [62]. Therefore, several other models were designed to run on such constrained platforms trading a reduction of the number of parameters and operations in exchange for a drop of accuracy. Indeed, if we observe Figure 2.3, the lightweight models do not match the high-performance ones in terms of accuracy only.

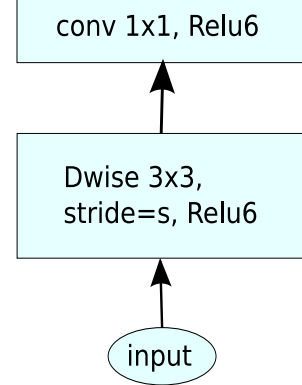
A clever choice of design decreases the number of parameters and computations of the model while reducing the drop of accuracy. As many approaches were proposed to reduce the size of a model, this study will focus on architectures that target the embedded space. This section describes five of these architectures:

- **SqueezeNet** [62] was focused on reducing the number of parameters of AlexNet (see Section 1.1.2) by introducing a new building block: **Fire Module**. Therefore, their architectures are very similar. SqueezeNet replaces all layers (except the first and last one) by **Fire Modules**. The **Fire Module**, observed in Figure 2.2a, is composed of two convolutional layers. The first one called *squeeze block* only performs  $1 \times 1$  convolutions to squeeze the number of input channels. The reduction of the number of channels decreases the computational complexity and number of parameters of the next convolutional layer. Moreover, they also chose  $1 \times 1$  convolution because it requires fewer parameters than  $3 \times 3$  convolution. The second convolutional layer called *expand block* is composed of  $1 \times 1$  and  $3 \times 3$  convolutions. With this architecture, the size of AlexNet is decreased from 240MB to 4.8MB [62]. It can even be reduced to 0.47MB without a drop in accuracy method by applying Deep Compression [63]. However, it has a big memory footprint, is slower in runtime, and consumes more energy than AlexNet [49].
- **MobileNet** [30] uses DSC, described in Section 1.1.1, to build small and low latency models that can fulfill the design requirements, as can be seen in Figure 2.2b. Two hyper-parameters are used to set the model size and throughput:
  - The width multiplier  $\alpha \in [1; 0[$ , which reduces the number of input and output channels at each layer,
  - The resolution multiplier  $\rho \in [1; 0[$ , which reduces spatially the input and output FMs at each layer.
- **ShuffleNet** was developed by Zhang et al. [64], in 2018. It is designed to be a computation efficient architecture, especially for mobile devices with very limited computing power. Indeed, it reduces the computation cost while maintaining the accuracy by using **pointwise group convolution**, decreasing the computation complexity of  $1 \times 1$  convolutions. It also uses **channel shuffle** operation on the channels such that **group convolutions** obtain information from different groups. Then more powerful structures can be built with multiple group convolutional layers. However, the group convolutions and the bottleneck structures add Multiply-Accumulate (MAC) which is a non-negligible cost [65]. The group convolution contributes to network fragmentation and reduces parallelism. Moreover, the “Add” operation, as seen in Figure 2.2c, is quite significant.
- **NasNet** was developed by Zoph et al. [66], in 2018. The idea was to use a search method called Neural Architectural Search (NAS), to find good convolutional architectures on a specific dataset. For that purpose, NasNet uses a Recurrent Neural Network (RNN) to generate efficient architectures. The RNN generates sample child networks with different architectures, which are trained to convergence. The accuracy of the child networks is used to train the RNN, which will generate better architectures over time. A convolution layer can be seen in Figure 2.2d. The learned architecture is flexible as it may be scaled in terms of computational cost. The network provides a higher accuracy with comparable parameters and MAC than MobileNet and ShuffleNet (described previously) [66]. However, the resulting network ends up very complex [22].
- **MobileNetV2** was developed by Sandler et al. [22], in 2018. It is an improvement of MobileNet (described previously) in terms of accuracy and does not require special

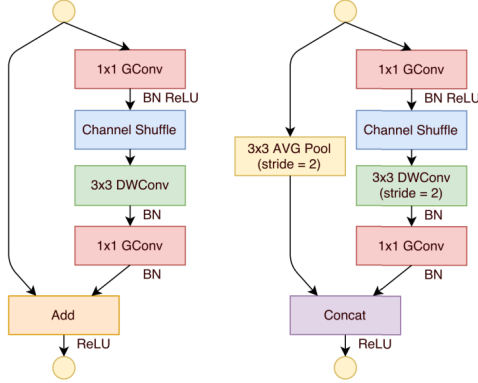
operators. It has also a smaller memory footprint. Furthermore, MobileNetV2 has faster inference and fewer parameters than MobileNet. MobileNetV2 has already been explained in Section 1.1.2.



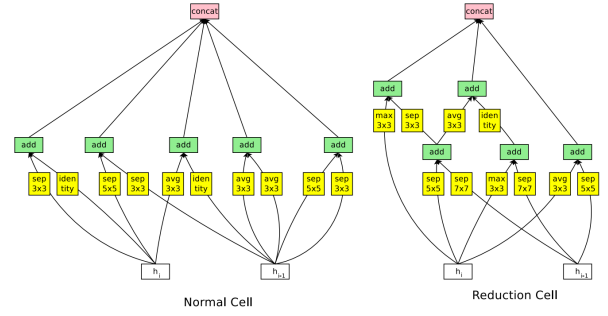
(a) Squeezenet Fire Module[62]



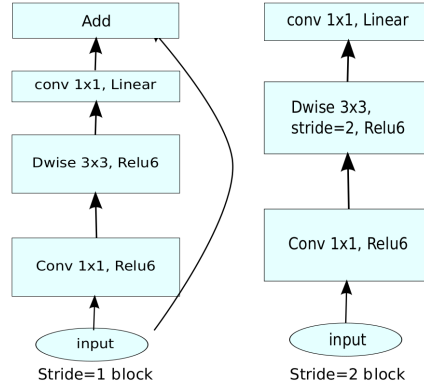
(b) MobileNet convolutional block [30]



(c) ShuffleNet convolutional block [64]



(d) NasNet convolutional blocks [66]



(e) MobileNetv2 convolutional blocks [22]

Figure 2.2: Convolutional block from different architectures

The architecture used in this work was developed to implement MobileNetV2 because of its simplicity and its state-of-the-art performance (see Table 2.1). Moreover, MobileNetV2 requires fewer parameters while providing state-of-the-art accuracy when comparing to the different architectures, as shown in Figure 2.3.

| Network           | Top 1       | Params      | MAdds       | CPU          |
|-------------------|-------------|-------------|-------------|--------------|
| MobileNetV1       | 70.6        | 4.2M        | 575M        | 113ms        |
| ShuffleNet (1.5)  | 71.5        | <b>3.4M</b> | 292M        | -            |
| ShuffleNet (x2)   | 73.7        | 5.4M        | 524M        | -            |
| NasNet-A          | 74.0        | 5.3M        | 564M        | 183ms        |
| MobileNetV2       | <b>72.0</b> | <b>3.4M</b> | <b>300M</b> | <b>75ms</b>  |
| MobileNetV2 (1.4) | <b>74.7</b> | 6.9M        | 585M        | <b>143ms</b> |

Table 2.1: Performance on ImageNet, comparison for different networks [22]

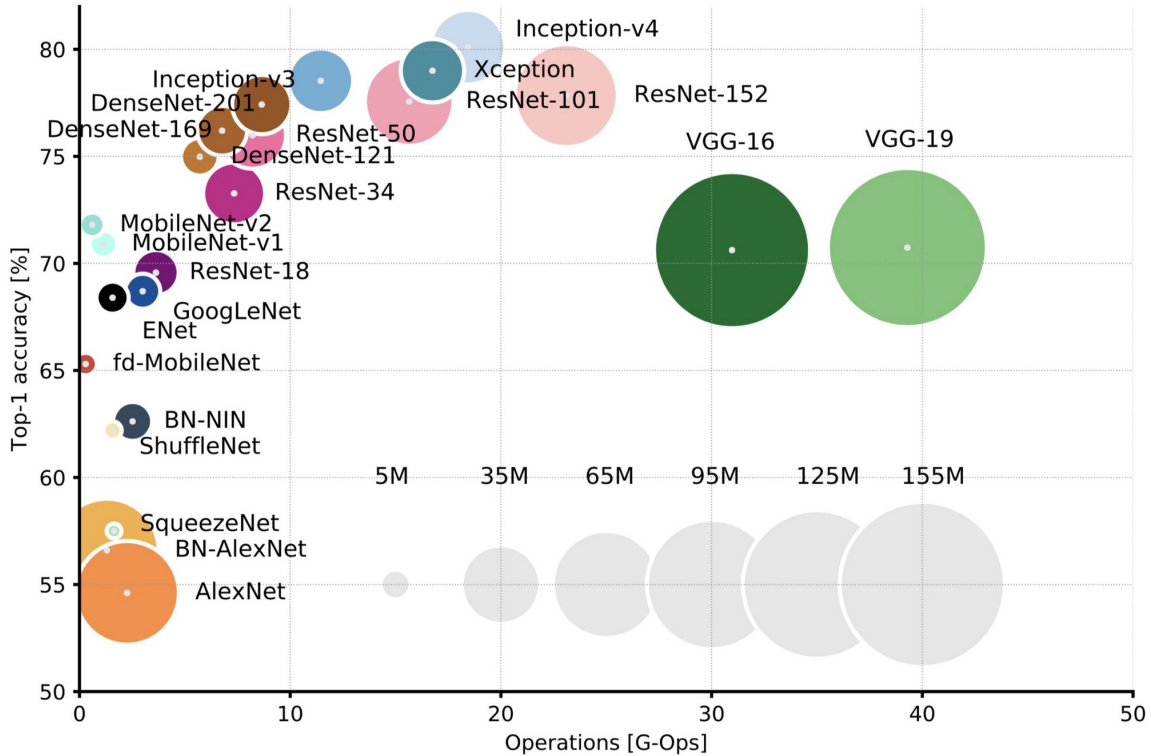


Figure 2.3: Ball chart reporting the Top-1 accuracy of various architectures vs. their computational complexity [67]

## Pruning

To improve the inference phase of a CNN, pruning can be used, especially for platforms with limited computational resources [68]. According to Liu et al. [68] and Denton et al. [69], the huge number of parameters in a network might create a problem of **over-parametrization**. Over-parametrization means that there are redundancies in NN parameters and that the same performance could be achieved with only a subset of them. In other words, a lot of parameters are unimportant or unnecessary [21]. Pruning is defined as removing the parameters considered as not important. For example, Baoyuan Liu et al. [70] achieve more than 90% sparsity of parameters in convolutional layers in AlexNet with less than 1% accuracy loss.

We can explain why pruning works by **The Lottery Ticket Hypothesis** [71, 72]: “A randomly initialized, dense neural network contains a subnetwork that is initialized such that—when trained in isolation—it can match the test accuracy of the original network after training for at most

*the same number of iterations.*” From this postulate, those unimportant weights can be set to zero (prune) because they do not improve the accuracy of the model.

According to Cheng et al. [21], pruning has two major benefits for the inference phase. First, less storage is required. Indeed, the non-pruned weights are sparsely distributed among the kernels. Thus, they can be stored in a compressed format reducing memory utilization. Second, pruning reduces the arithmetic complexity of the network. As convolutions perform a weighted sum with the input FM, each Multiply-Accumulate (MAC) operation with a pruned weight can be discarded. Moreover, Han et al. [73], Mao et al. [74], and Kang [75] pointed out that some pruning ratios can also improve the accuracy of the network, which can be explained by a form of regularization.

Various pruning schemes are focused on increasing the sparsity of the network without a drop of accuracy [63, 73]. We call this pruning scheme where all unimportant parameters are pruned without extra constraint **unstructured pruning** [21]. However, it is challenging to exploit the performance and the high parallelism of FPGA with this kind of pruned network. Indeed, this kind of pruning scheme creates irregularities in the data access pattern [35]. It means that the number of pruned weights is different in each kernel, and we should adapt the circuitry to the worst case. As a consequence, all filters conduct wasteful operations except the worst case [76]. Furthermore, Anwar et al. [77] pointed out that unstructured pruning requires overhead for computing addresses of the sparse non-pruned elements. Therefore, we should find pruning patterns that would be more hardware-friendly.

In contrast to the unstructured pruning, we have **structured pruning** schemes [75]. They combine a structure regularization for accuracy and locality optimization for computation efficiency. According to Anwar et al. [77], “*Structured pruning has no or little extra costs*”. We can categorize the various schemes into different groups [21, 75, 77, 78]:

- **Depth-wise:** all the weights of a layer are pruned. The layer is then removed.
- **Kernel-wise:** instead of pruning all the weights, we keep a ratio of kernels, which means a reduction of the number of output channels. This pruning scheme is provided in Figure 2.4a.
- **Channel-wise:** it is one of the most popular methods because it still can fit in the convolutional deep learning frameworks [68]. A layer of the input FM is pruned, which means that the layer is also pruned in all the kernels, as can be seen in Figure 2.4b.
- **Shape-wise:** we prune the same weights in each kernel or group of kernels. For example, this pruning scheme was used in Zhu et al. [35]. It is illustrated in Figure 2.4c

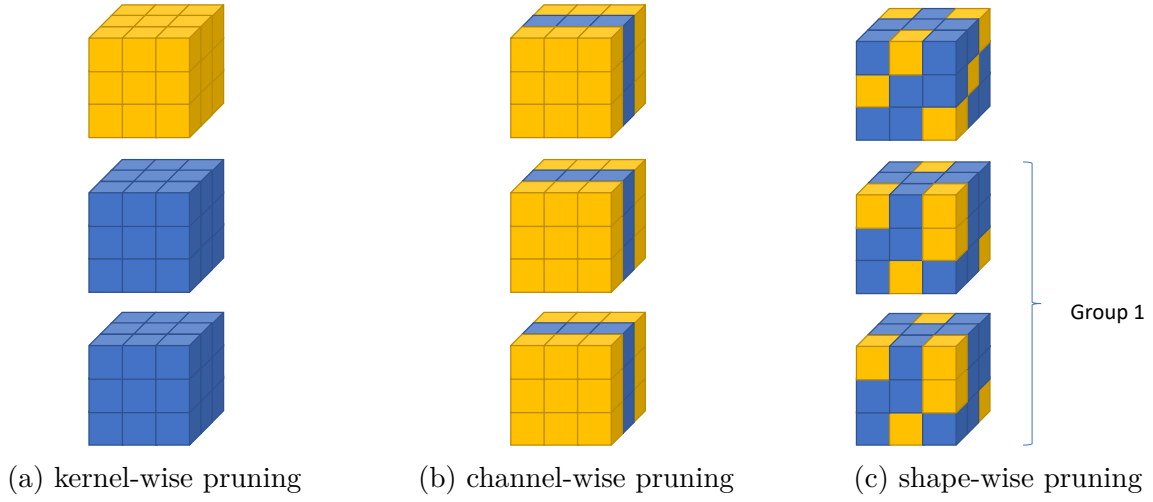


Figure 2.4: Structured pruning schemes, where the yellow weights are the pruned ones, inspired by [21]

The previously cited pruning schemes are ordered from very coarse-grained to fine-grained sparsity [74]. As explained previously, coarse-grained sparsity (channel-wise and filter-wise pruning) provides a higher acceleration and can be used when implementing CNN on GPU or CPU [21, 74]. However, finer-grained ones provide higher accuracy and as the sparsity increases, the accuracy is less affected, as can be seen in Figure 2.5. Therefore, this work focuses on developing customized hardware that can exploit a more fine-grained sparsity [74] to provide a higher pruning while limiting the drop of accuracy.

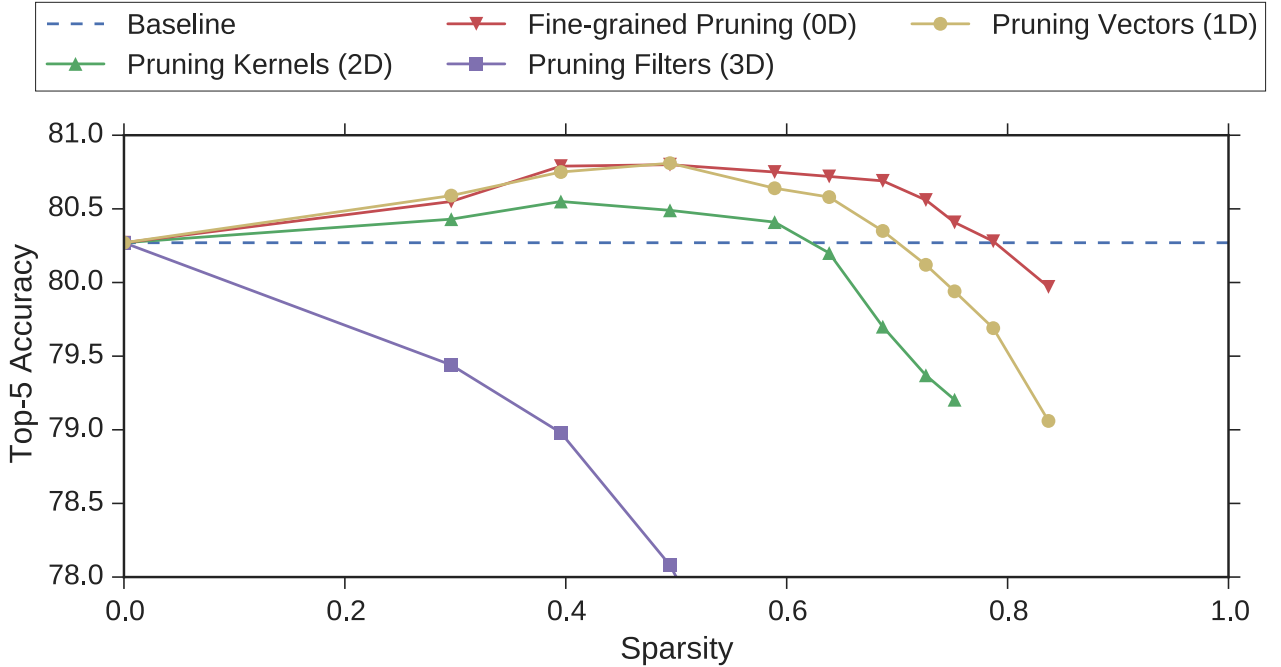


Figure 2.5: Accuracy-Sparsity Curve of AlexNet obtained by pruning [74]

Some studies focused on the acceleration of the inference step of lightweight models thanks to pruning. Zhang et al. [40] and Tu et al. [79] applied pruning on DSC kernels. They both chose **Channel-wise** pruning because it does not create sparse connections and it efficiently improves

the speed of the inference. It also reduces the computational cost of the  $1 \times 1$  (pointwise) convolutions, which has the biggest number of parameters and computational complexity. In MobileNet, it is about 95%. By discarding one channel, the associated depthwise convolution is also avoided. Moreover, the pointwise kernel producing that channel in the previous block can also be pruned. We can see the process in Figure 2.6.

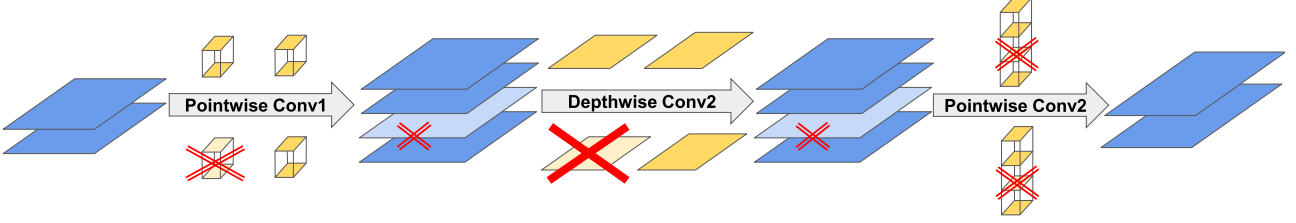


Figure 2.6: Pruning a depthwise separable convolution [79]

In this work, we focus on a structured pruning scheme for depthwise separable convolution. More precisely, we develop an architecture on FPGA that combines both advantages of pruning and depthwise separable convolution.

## Quantization

Quantization is an approach to trade accuracy for a decrease of the storage requirements of a network [63]. Indeed, we can define quantization as the reduction of the number of bits representing a weight or pixel. Moreover, instead of using a floating-point number, we can use a **fixed-point number** [21]. As said previously, fixed-point numbers are known to be more efficient on hardware such as FPGA because we can use integer arithmetic [31]. Quantization to fixed-point numbers can then reduce the memory requirement and the latency of the inference stage.

The format to encode the fixed-point representation of a real number is the *Q-format* [80]. A  $N$ -bit number, noted  $Q_{m,n}$ , is divided into two parts separated by an implied binary point. The  $m$  bits are used to represent the integer part of the number (including the sign bit), and the  $n$  bits are used to represent the fractional part, as can be seen in Figure 2.7. The bitwidth associated with each part can be either the same, either fine-tuned for each layer after analysis. Indeed, Qiu et al. [81] and Yin et al. [82] choose a different range for each layer, but doing it for every weight is not memory-efficient.

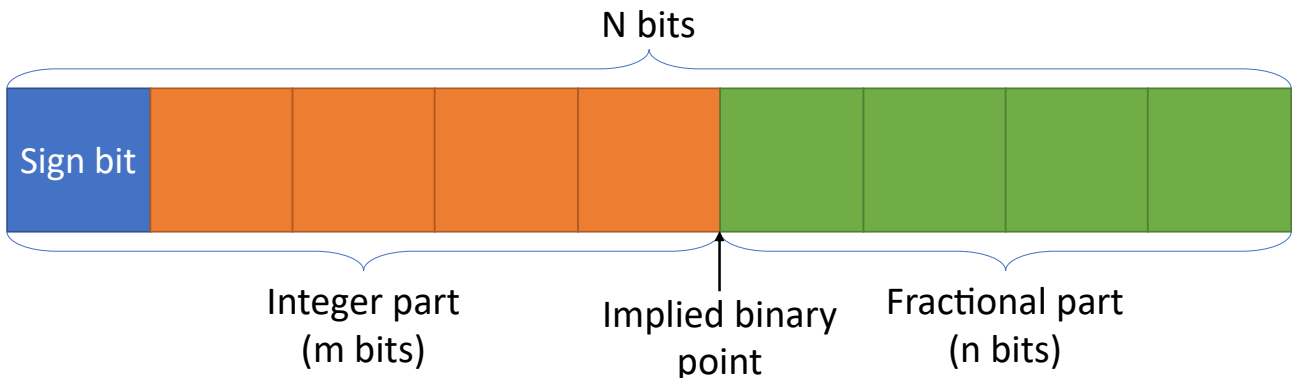


Figure 2.7: Illustration of the Q-format

As pointed out by Han et al. [63], quantization and pruning techniques are orthogonal and can be combined to compress further the network. Unfortunately, not all existing networks are friendly for quantization, like MobileNet. MobileNet with quantized pixels and weights has a large drop of accuracy compared with its Non-quantized version (70.50% using floating-point model vs 1.80% using an 8-bit pipeline) [83]. However, the work of Sheng et al. [83] showed that the source of the accuracy drop was the design of the separable convolution core layer. They proposed therefore a new quantization-friendly separable convolution core layer. Works on MobileNetV2 should then be done to verify the fixed-point inference accuracy. Still, an 8-bit pipeline might not be optimal for MobileNetV2 as increasing the bitwidth to 16-bit could boost accuracy [21]. This bitwidth is also widely used [84, 39].

**Therefore, 16-bit fixed-point Q-format is adopted for input data, weights, and intermediate data in the frame of this thesis. Moreover, as said previously in Section 1.1.1, we can limit the integer part to 3 bits (one bit is added to express the sign of the weights), and we can use  $Q_{4.12}$ .**

## 2.2 CNN optimizations on FPGA

Section 1.2 provided a general layout of what is an FPGA and how it works. This section is focused on how to implement a given model on an FPGA, and what hardware optimizations can be made to accelerate the inference.

Section 2.2.1 reviews de various approaches to perform an efficient inference of a general CNN on FPGA.

Section 2.2.2 gives an overview of how to perform DSC on FPGA and Section 2.2.3 is focused on successful FPGA-based architecture that implemented pruning.

### 2.2.1 General CNN inference on FPGA

As described in Section 1.2, FPGA is an array of programmable gates that can interface with the outside world. As a result, to perform the convolution operation, the FPGA has to fetch weights and input FM from the outside world (for example an external memory) and compute the convolution with them. Once the result is obtained, the FPGA can output the results through its IOE.

According to Chen et al. [85], the data movement can often be more energy-consuming than actual computation. Therefore, smart memory management is required to implement CNN on low power devices such as FPGA. We discover in this section various techniques for handling the Datapath on the FPGA.

## Systolic Arrays

Systolic arrays were implemented on the early FPGA-based accelerators [18, 86, 87]. A static systolic array is a static array of Processing Element (PE), controlled by the CPU. The PE is the basic computing unit for convolution. Therefore, each PE is involved in a part of the computation and can communicate with adjacent PEs. An illustration of this principle can be found in Figure 2.8. The configuration can only support convolution with a kernel size  $K_*$  lower than a maximal kernel size, such that  $K_* \leq K_m$  (because the number of PE is fixed). The systolic array allows spatial data reuse between rows and columns, high clock frequency, and temporal data reuse using FIFO queues [88, 89].

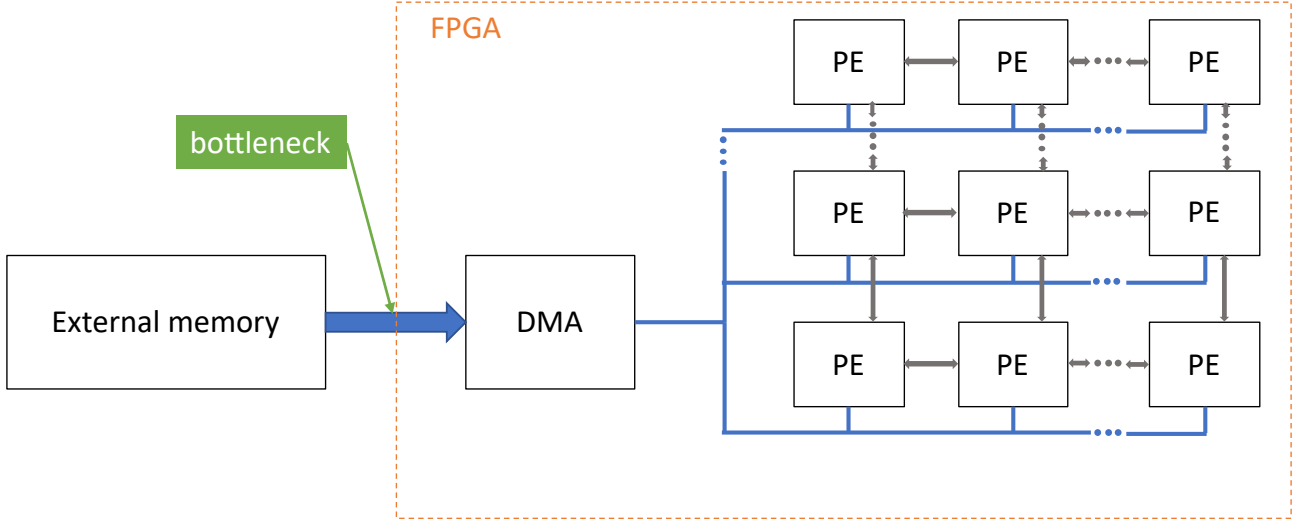


Figure 2.8: Static Systolic Arrays, inspired by [18]

However, Systolic Arrays have inefficiency problems. First, when the kernel size is much smaller than the maximal kernel size ( $K_* \ll K_m$ ), there is an underutilization of the resources. For example, [87] notes that for  $3 \times 3$  kernels, only 9% of the Digital Signal Processing (DSP) blocks are used. Second, data caching is not implemented. It means that it has always to fetch input from the external memory and it increases latency [18, 90]. The device performance is depending then on the memory bandwidth and it becomes memory-bounded. Furthermore, it is not energy efficient. According to Horowitz [91], the DRAM accesses severely impact both throughput and energy efficiency. External memory accesses are more expensive by several orders of magnitude in terms of energy and latency.

## Data-flow MoC for CNNs (2D mapping)

Data-flow Model of Computation (MoC) can be used to accelerate CNNs on FPGA. This approach is motivated by the feed-forward aspect of the inference stage of CNN, which is purely data-driven [18]. Data-flow Model of Computation (MoC) was firstly investigated by Li et al. [92]. CNN can be represented as a graph where:

- **Nodes** are processing units called *actors*. Each actor follows a data-driven execution where the execution is triggered by the availability of input, which is the case for a CNN.

- **Edges** are communication FIFO channels. Actors exchange data called *tokens* through those FIFO channels.

A representation of such a network can be found in Figure 2.9, where  $FM_*^i$  is the  $i$ th layer of the FM.

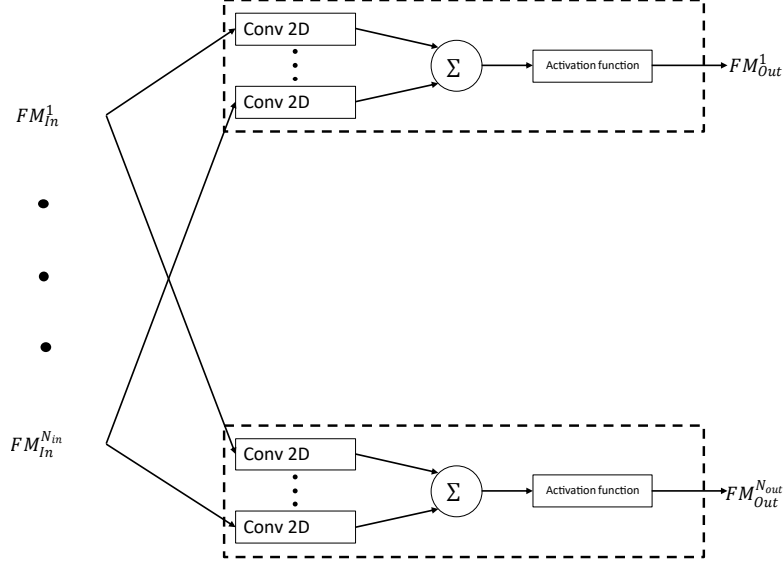


Figure 2.9: Graph representation of a convolution layer, inspired by [18]

As the number of tokens produced and consumed by an actor can be specified in a CNN, we can apply a static data-flow [93]. Therefore the CNN can be modeled as a topology matrix and we only need to optimize those matrix components to minimize latency or energy consumption [94]. Those parameters are then used to derive PE and buffer configurations. However, as pointed by Abdelouahab et al. [95], we need to have direct hardware mapping between the graph and the FPGA to be efficient. It means that all the computations must be unrolled. However, we are then bounded by the hardware resources and the size of CNN, preventing implementing this approach for deep models [18].

## Loop Optimizations

To overcome the inefficiencies of Systolic arrays and because Data-flow MoC can only be applied to small networks, data must first be cached into the on-chip memory of the FPGA before processing. However, according to Ma et al. [34], the on-chip memory of FPGA is not large enough to store all the data (requiring gigabytes). As a result, we need *tiling* to fit a smaller portion of data into memory on-chip [20]. We can summarize the dataflow as follow: the FPGA fetches a tile of data from external memory to on-chip buffers. Then the data is fed to the PEs to compute the convolution. The results of the computation are transferred back to on-chip buffers and afterward, to the external memory. When the results are written back to memory, we can fetch a new tile of data. We can see the process in Figure 2.10

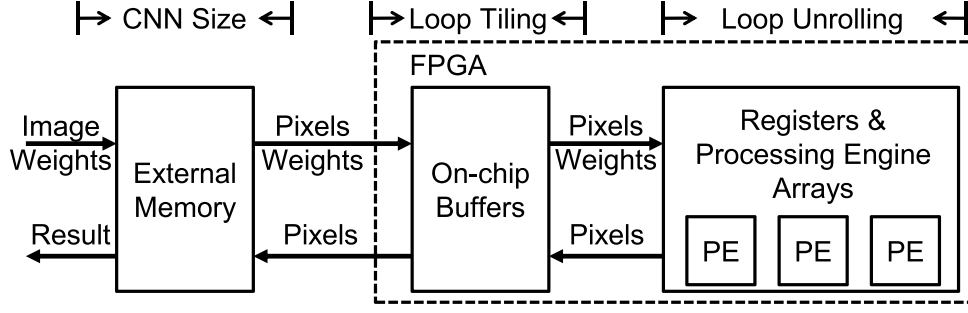


Figure 2.10: FPGA memory hierarchy, from [34]

Consequently, we can define a storage hierarchy with different energy costs. We can define the memory levels as [49, 91]:

1. **External Memory:** it stores all the CNN data. It has a large memory capacity ( $\sim$ GB) but accesses are costly in terms of energy and latency.
2. **On-chip memory:** it stores the data fetched from external memory to feed the registers and PEs. Accesses are less expensive to the external memory but it has less storage ( $\sim$ hundreds of KB).
3. **Register:** is associated with the PEs. The storage capacity is less than a few KB but it is faster and more energy-efficient. Accesses are 1 or 2 orders of magnitude lower energy than from External memory.

Therefore, after tiling, the convolution operation can be decomposed in multiple loops, as observed in Figure 2.11. Nevertheless, an improper loop tiling may degrade the performance of the FPGA because we must minimize access to expensive memory levels. The major challenge when implementing a CNN on FPGA is then to maximize data reuse in small cost memories. To address those problems, loop optimization techniques are applied to the nested loops in Figure 2.11. There are three techniques:

1. **Loop tiling:** as said above, the on-chip memory of the FPGA is not large enough to store all the weights and FMs of all CNN layers [18]. The FPGA stores a tile of the data, accessed from the external memory, and the lower bound is set on the required on-chip buffer size. The tiling parameters represent the tile size. They are defined as  $T_*$ , where  $*$  is the name of the dimension. For example, the size of an input pixel buffer (resp. output) is  $T_{ix} \times T_{iy} \times T_{if} \times \text{pixel\_datawidth}$  ( $T_{ox} \times T_{oy} \times T_{of} \times \text{pixel\_datawidth}$ ). Furthermore, we have the following constraint:  $T_* \leq N_*$ .
2. **Loop interchange:** it defines the order of sequential computation of the convolution loops. There are 2 types of loop interchange: *intratile*, patterns of data movements from on-chip memory to registers (loop one to four in Figure 2.11); *intertile*, patterns of data movements from external memory to on-chip buffers (the other loops).
3. **Loop unrolling:** it determines the number of parallel computations. Four types of unrolling configuration can be done (depending on the loop to unroll). The different configurations can be seen in Figure 2.12. As the FPGA has constrained resources, it is not always possible to fully unroll a loop. The unrolling parameters are defined as:  $P_*$ , where  $*$  is the name of the dimension. We also have the following constraint:  $P_* \leq T_* \leq N_*$ .

```

1  for (int of=0; of<N; of+=Tof){
2    for (int iy=0; iy<Niy, iy+=Tiy){
3      for (int ix=0; ix<Nix, ix+=Tix){
4        for (int if=0; if<N; if+=Tif){
5          // DRAM: Load in on chip buffers the tiles
6          // X[l,if:if+Tif,iy:iy+Ty,ix:ix+Tix]
7          // Theta [l,n:n+Tof,if:if+Tif,j,k]
8          for (int tof=0; tof<Tof; tof++){ -> Loop-4
9            for (int tiy=0; tiy<Tiy, tiy++){ -> Loop-3
10             for (int tix=0; tix<Tix, tix++){ -> Loop-3
11              for (int tif=0; tif<Tif; tif++){ -> Loop-2
12               for (int ky=0; ky<Nky, ky++){ -> Loop-1
13                for (int kx=0; kx<Nkx, kx++){ -> Loop-1
14                  Y[tof,tiy,tix] += X[tif,tiy+ky,tix+kx] *
15                    W[tof,tif,ky,kx];
16                }}}}
17              // DRAM: Store output tile
18            }}}}
19          }

```

Figure 2.11: Loop Tiling, from [18]

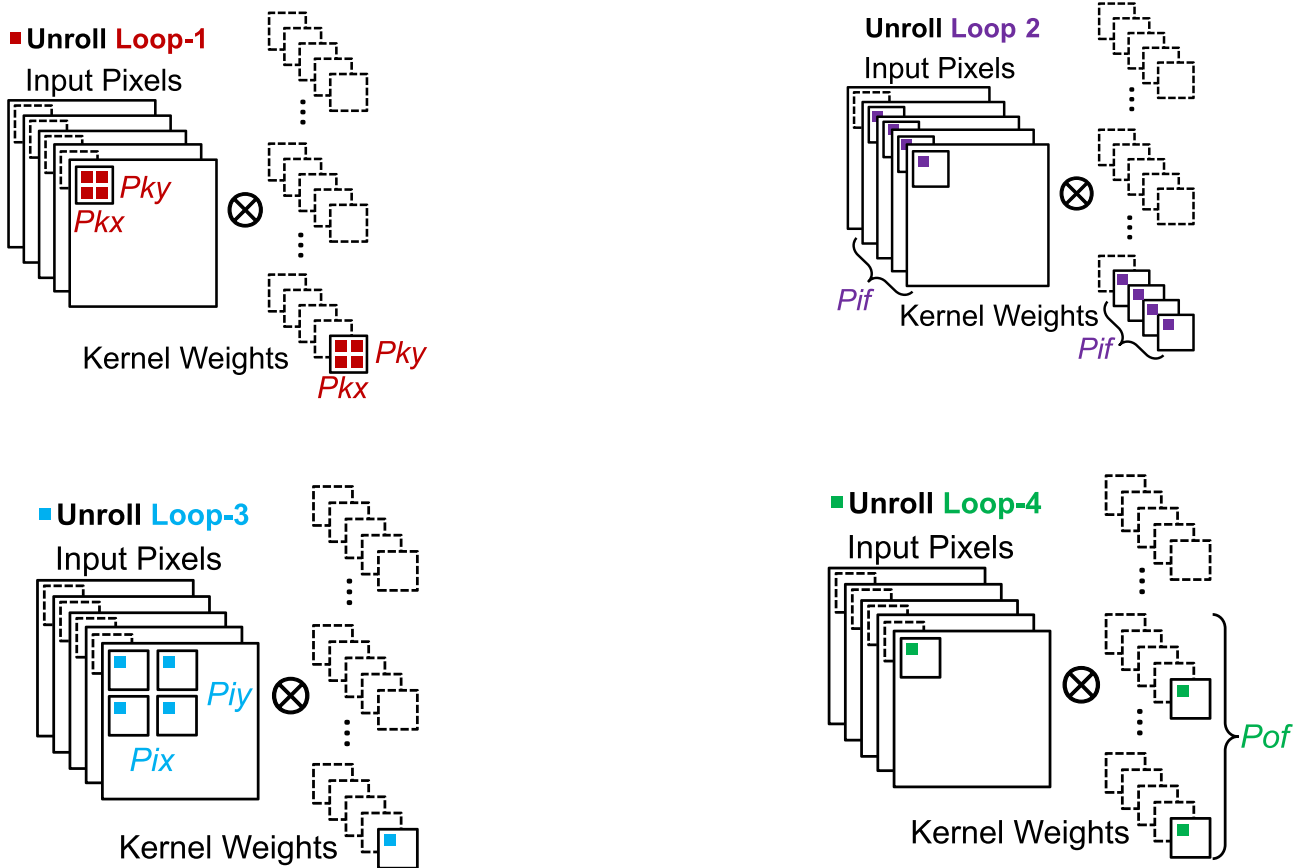


Figure 2.12: Loop unrolling configurations, from [34]

The loop optimization design variables determine the number of PEs (unrolling parameters), the buffer size, and the number of external memory access (tiling parameters). We are going to review different studies of the design space exploration of unrolling parameters, tiling parameters, and loop interchange, to design the most efficient accelerator on the target platform.

The work of Zhang et al. [20] proposes an analytical solution using the roofline model [96] to identify the CNN design with the best performance and lowest FPGA resource requirement. They chose a loop unrolling of  $(T_{if} \times T_{of})$  to avoid complex connection topologies for all arrays. It allows spatial reuse of pixels for  $T_{of}$  unrolls, and the temporal reuse of weight for  $(T_{ox} \times T_{oy})$ . However, the disadvantage of this method is the need for a large local memory to store the partial sums  $(T_{of} \times T_{ox} \times T_{oy})$ .

Once the loop ordering and unrolling parameters have been set, the roofline model can be used to find the optimal tiling parameters. Mittal [89] defines this design space exploration method as: “*The roofline model relates performance to computational performance and off-chip memory traffic*”. As an implementation can either be computation-bounded or memory-bounded, it uses those two limits to find the best trade-off between memory bandwidth and computational speed, where  $CTC$  is the communication-to-computation ratio.

1. **Computational roof:**  $= \frac{\# \text{ of operations}}{\# \text{ of execution cycles}}$ .
2. **CTC ratio:**  $= \frac{\# \text{ of operations}}{\# \text{ of external data access}}$

Therefore, the achievable performance can be computed using equation 2.4.

$$Att \text{ perf} = \min (Computational \text{ roof}; CTC \text{ ratio} \times bandwidth) \quad (2.4)$$

The roofline method is shown in Figure 2.13. We can conclude that the best solution to pick is solution C since it offers the best performance while requiring the least bandwidth [20].

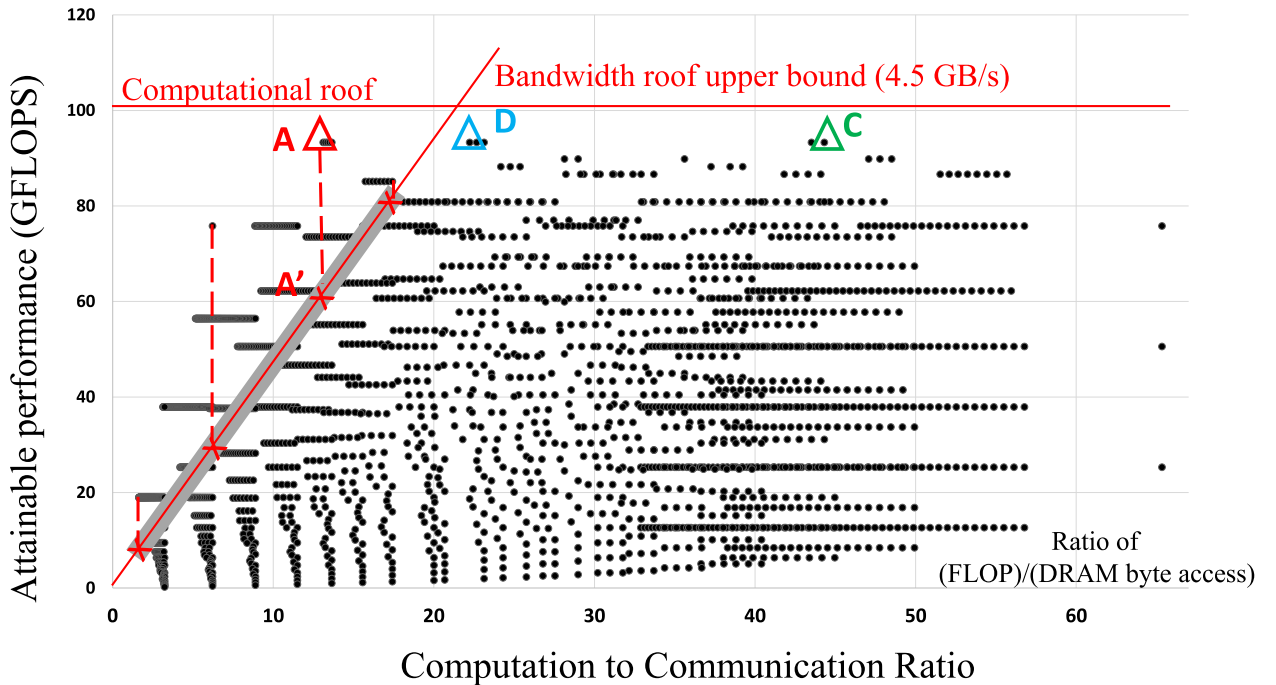


Figure 2.13: Design space of platform-supported designs, from [20]

Motamedi et al. [15] chose the best parameters according to the resources and the best data reuse scheme of the target platform. They did not tile on the kernel on the spatial axis because

recent CNNs have small filters and it would mean that a pixel can be loaded multiple times. As the tile goes over the input and output FM channel, there is a choice between maximizing the reuse of the input FM (MIR, each FM is loaded once but some intermediate results have to be written back to memory) or output FM (MOR, to avoid writing back intermediate results to memory but some input FM have to be loaded more than once). Then they performed an exhaustive search on the tiling and PE parameters to find the optimal design. It can be layer-specific (but the FPGA needs dynamic reconfiguration and Zhang et al. [20] proved that it is inefficient) or values are fixed for all layers.

A different approach, studied by Ma et al. [34], proposes a design space exploration regarding various design objectives:

- **Latency:**  $P_*$  should be common factors of  $T_*$  for all convolution layers to fully utilize PEs and  $T_*$  should be the common factors of  $N_*$  to make full use of external memory transactions.
- **Partial sum storage:** to minimize the concurrent storage of partial sums in local memory and allowing to keep them into the PE, we should promote the computation of an output pixel to evacuate it to the external memory.
- **On-chip memory access:** on-chip memory access can be minimized by reusing at most pixels or weights in the PEs.
- **External memory access:** to minimize external access, a sufficient buffer size should be assigned to pixels and weights.

They proposed two flowcharts in Figure 2.14 to determine the number of partial sums and external accesses.

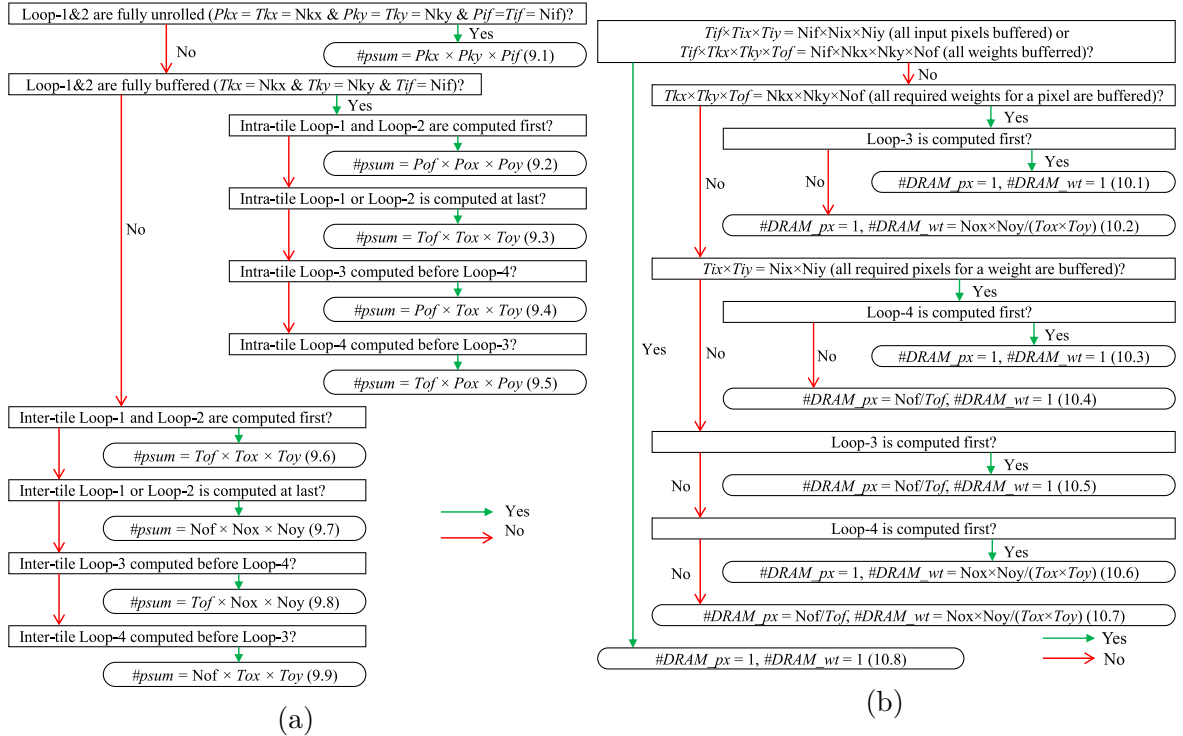


Figure 2.14: Design space exploration of: (a) total number of partial sums that need to be stored in memory; (b) the number of external memory accesses, from [34]

### 2.2.2 DSC

Two studies of the implementation of the DSC appeared to be relevant for this work: Bai et al. [39] and Liu et al. [12]. They present an FPGA accelerator that implements both standard convolution and DSC. Networks like MobileNetV2 use both convolutions, the first one for keeping as much information as possible, the second one to speed up the inference.

They both use a commonly used DL architecture that is called a heterogeneous system: a CPU which controls the memory accesses and an accelerator that computes the convolution [12]. The accelerator is designed in such a way that it can perform each layer of the network.

The accelerator in each work has the same structure. The accelerator uses weights, input, and output buffers. To use the same accelerator for both convolutions, they first compute an element-wise multiplication followed by an adder tree. The pattern of additions varies between the studies.

On one side, Bai et al. [39] divide depthwise and pointwise convolution. As the element-wise multiplication produces a cuboid, we only have to sum either in the spatial or channel axis, or all the cuboid pixels to perform the desired convolution.

On the other side, Liu et al. [12] perform both convolutions using the same adder tree. The dataflow to feed the adder tree depends then on the convolution. However, to use the same structure for both convolutions, it must fill some registers with zero-value (the DSC has fewer arithmetic operations, if we use the same structure we adapt it to the operation with the biggest arithmetic complexity).

To improve the throughput of the network, they use a ping-pong buffer for the weights. Instead of using one buffer that stores the data for convolution, they use two buffers. Alternatively, one fetches weights for the next tile while the other is used for convolution. An illustration is found in Figure 2.15.

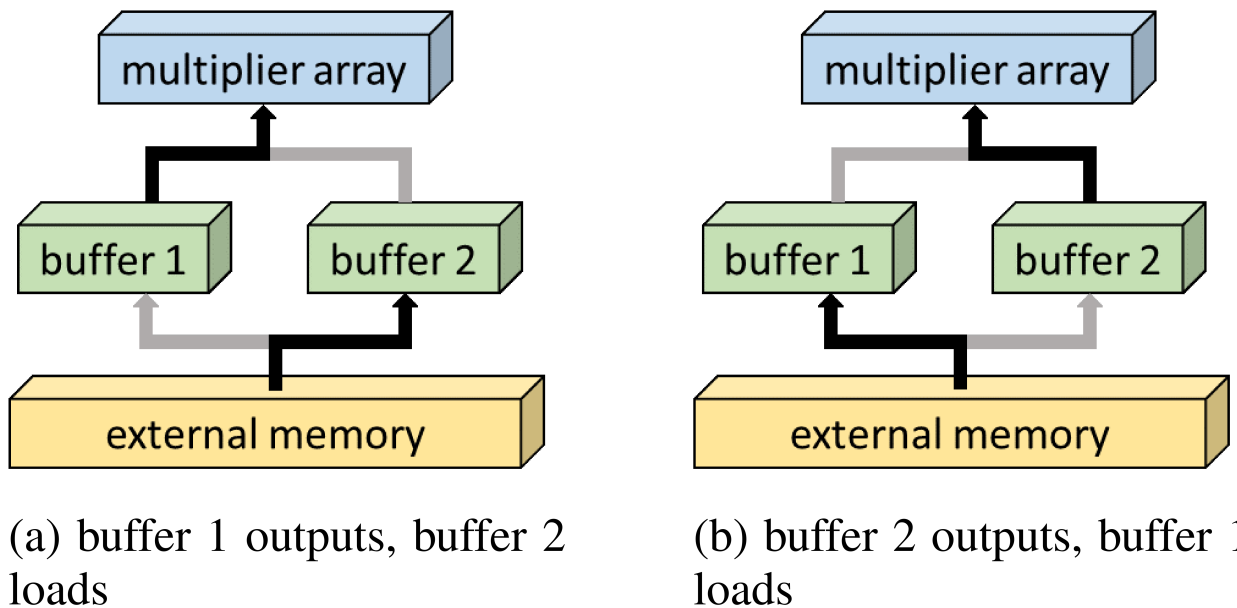


Figure 2.15: Weight buffer in ping-pong structure, from [39]

Finally, Liu et al. [12] used the roofline model as a method for design space exploration.

### 2.2.3 Pruning

When implementing a pruned CNN on an FPGA, we must address the problem of sparsity to keep the parallelism and the performance of the FPGA [35]. As a regular data access pattern is assured by applying a structured pruning scheme, there are some sources of inefficiency that an implementation must handle.

First, the PEs must avoid performing computation involving 0 weights. Kang [75] fetches  $N_{par}$  data in the channel axis and the  $N_{non-zero}$  weights corresponding to the fetched group. A multiplexer is used to choose the pixels associated with non-zero weights. An improvement can be made by avoiding 0 value pixels. Zhu et al. [35] clock gated the computation involving pixels with 0 value to save energy.

Second, to reduce the storage utilization, the zero-weights in a kernel must be discarded. The weights must be encoded in such a way that we keep both values and index of the weights to reduce the overhead of computing the output address. Usually, kernels are encoded in a format derived from the Compressed Sparse Row (CSR) [74]. For example, [35] modified the format to compress further and accelerate the inference.

Finally, the problem of **load-imbalance** can arise if the number of non-pruned weights is different in each PE [97]. Therefore some PE can finish before another one and it can lead to inefficiency. Zhu et al. [35] and Kang [75] solved this by setting a uniform number of weights in each kernel or group of fetching weights.

#### Conclusion about the State of the Art:

We knew from the previous chapter that the best performing models have a huge computational complexity and memory utilization, which limit the implementation of such models on mobile platforms, such as FPGA. Therefore, we reviewed the literature to find optimizations that would allow an efficient implementation of CNNs on FPGA. We have explored algorithmic optimizations, but they only reduce the computational complexity while increasing hardware utilization. Model optimizations seem to be more relevant because they aim at reducing both the number of parameters and operation in exchange for a drop of accuracy. Therefore, we can investigate the combination of different optimizations to further compress models, like pruning and quantization.

As a result, this work focuses on developing a pruning scheme on DSC to assess if their gain can be combined. To study the performance of the proposed pruning scheme, we use it on a state-of-the-art model that contains DSC: MobileNetV2. Moreover, as quantizing is an orthogonal approach to pruning, we also use fixed-point formats to accelerate further the inference of the architecture. We chose 16-bit because the loss of accuracy can be controlled.

# Chapter 3

## Accelerating Convolutional Neural Networks for FPGA using Depthwise Separable Convolution and Pruning

In the previous chapter, we introduced pruning, which reduces both size and computational complexity of CNNs. We also showed that there exist various pruning schemes. Each of them can be categorized from coarse-grained to fine-grained. Coarse-grained pruning schemes have the advantage that they can be easily implemented on GPU and CPU in exchange for a drop of accuracy. On the other hand, fine-grained pruning schemes can limit this reduction of accuracy but can create irregular access pattern that makes them difficult to implement on such platforms.

To have an efficient FPGA-based accelerator, we should focus on implementing the most fine-grained pruning scheme possible to apply a high sparsity while limiting the loss of accuracy. This shows the advantage of using FPGA accelerator because, as said previously, this type of pruning scheme is not easily implementable on CPU and GPU

In this chapter, we define an FPGA-based accelerator architecture that integrates both pruning and DSC (see Section 1.1.1) to reduce the number of weights and operations. To show its applicability, this architecture implements a state-of-the-art network that includes DSC and targets an embedded platform: MobileNetV2 (Sections 1.1.2 and 2.1.2). Since the pruning can cause degradation of the performance, we have to determine the design objectives of the architecture and its associated pruning scheme, which are the following:

1. The pruning scheme is as fine-grained as possible.
2. The pruning scheme reduces the computational complexity.
3. The pruning scheme allows for a reduction of the memory required to store the weights.
4. The proposed architecture provides a logically correct output.
5. An increase in the sparsity improves the performance of the architecture.

Moreover, the FPGA-based accelerator architecture is implemented on a Cyclone V FPGA.

Section 3.1 aims at developing the pruning scheme and the algorithm to handle it. We also define in this section the weight and operation reduction factors obtained when applying the proposed pruning scheme. We also define a weight compressed format to reduce weight memory use. We then perform a loop analysis on the proposed algorithm to determine the optimal hardware design variables and the structure of the architecture.

Section 3.2 describes the implementation of the FPGA-based accelerator architecture and its dataflow. We start by explaining the overall architecture and then we go in more detail into each of its components.

Section 3.3 shows the results obtained and discusses if the proposed design objectives are met.

### 3.1 Design of an FPGA-based accelerator integrating structured sparse DSC

As previously said, this section aims at developing a pruning scheme for the DSC that fulfills the proposed design objectives and an algorithm able to support it. First, we detail the pruning scheme used and the weight and operation reduction factors obtained when applying the pruning scheme. Second, we analyze the sparse kernels to find a compressed format that reduces memory usage and overhead of determining the address of the non-pruned weights. Third, we describe the algorithm proposed to perform the sparse DSC using the proposed pruning scheme. Finally, we analyze the convolution loops to determine the optimal design of the PE that performs the convolution.

#### 3.1.1 Pruning scheme

DSC is composed of two types of convolution: depthwise and pointwise convolutions (see Section 1.1.1). We decided to apply pruning on the pointwise filters for the following reasons:

- In MobileNet and MobileNetV2, most of the operations are done in the pointwise convolutions [40, 79]. So it is the operation where the reduction of weights and computational complexity would be the most relevant (each operation requires one weight).
- As each kernel of a pointwise filter is a vector of size  $1 \times 1 \times N_{if}$ , we, therefore, have to prune in the channel-axis. This scheme was proven to be successful by Kang [75].
- The pruning scheme can be applied to each  $1 \times 1$  convolution of the network.

As mentioned above, we decided to develop a pruning scheme inspired by the methodology of Kang [75], which prunes weights in the channel-axis.

Ideally, without pruning, each PE performing a pointwise convolution fetches  $N_{if}$  weights and input pixels corresponding to spatial coordinates  $(x, y)$  in each input channel (pointwise convolution only convolves pixels in the channel-axis). Then each weight is multiplied with its corresponding pixel and the products are summed to obtain one output pixel of one channel. If we apply an unstructured pruning on the pointwise filter, we can use the same algorithm as above but only each non-pruned weight is multiplied with its corresponding pixel (multiplication implying zero-value weights can be safely discarded).

However, as the resources of the FPGA are limited, the PE can only fetch  $N_{par} \leq N_{if}$  weights and pixels at each time (which corresponds to a **fetching group**; each input FM and kernel

are composed of  $N_{gr}$  fetching groups). Therefore, we would do the convolution between each fetching group and accumulate the different partial sums belonging to the same output pixel until an output pixel is produced. If an unstructured pruning is applied, this can cause several issues, as pointed out by [75].

First, it can cause misalignment between the weight and pixel fetching groups. Indeed if we prune the first  $N_{par}$  weights in the kernel, the first pixel fetching group is useless. Second, there can be a load-imbalance problem (see Section 2.2.3) that can occur between two PEs if the number of pruned weights between two kernels is different.

To solve these problems, the solution proposed by Kang [75] is to align pixel and weight fetching groups. Each pixel fetching group has a corresponding weight fetching group containing at least one non-pruned weight. Kang [75] also constrains the number of non-pruned weights in each fetching group to be the same number, called  $N_{np}$ . Therefore, it solves also the load-imbalance problem. As a result, the proposed sparse DSC can be illustrated in Figure 3.1.

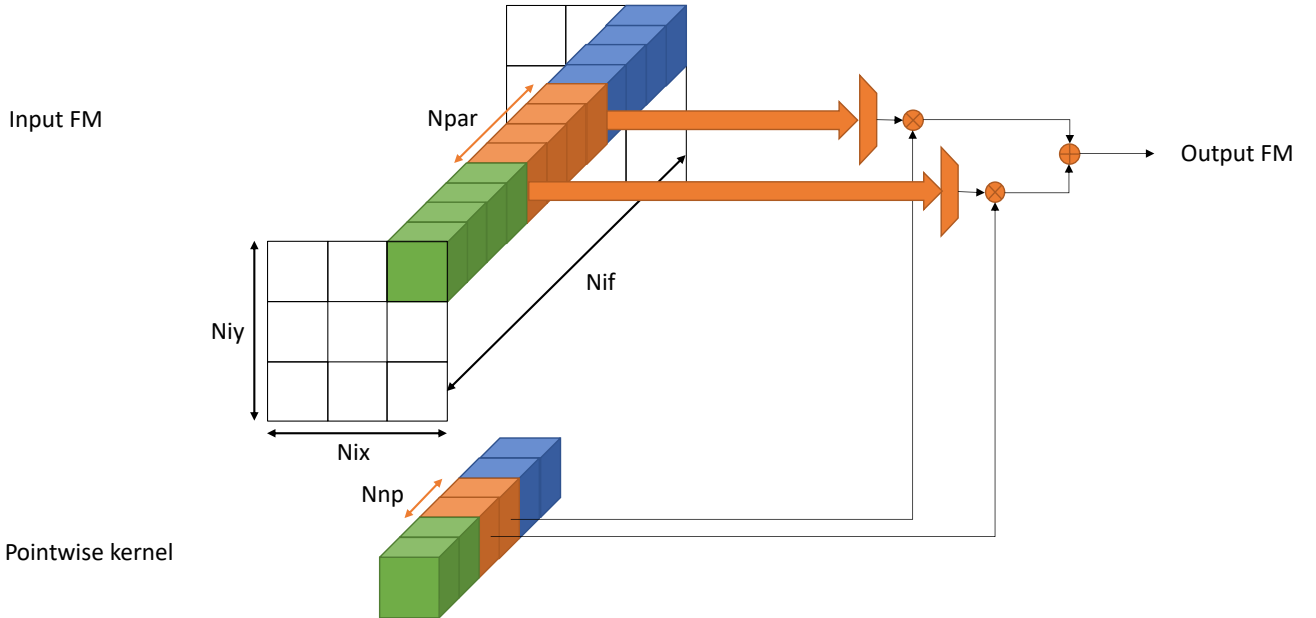


Figure 3.1: Process of a pointwise convolution with a sparse pointwise kernel, where  $N_{par} = 4$  and  $N_{np} = 2$ , inspired by [75]

To summarize, each pointwise kernel is composed of  $N_{gr}$  fetching groups of  $N_{np}$  weights. Each group corresponds to a pixel fetching group of size  $N_{par}$ . We define the ratio between the number of non-pruned weights and the size of the fetching group without pruning as the **pruning ratio**  $\alpha$ , in other words, the ratio of weights remaining after pruning. The ratio of pruned weights is therefore equal to  $1 - \alpha$ . The pruning parameters are defined in Table 3.1. Moreover, the proposed pruning scheme can also be applied to each  $1 \times 1$  kernel.

|           |                                           |                                                              |
|-----------|-------------------------------------------|--------------------------------------------------------------|
| $N_{par}$ | The input pixel fetching group size.      | $N_{par} \leq N_{if}$                                        |
| $N_{np}$  | The pointwise weight fetching group size. | $N_{np} \leq N_{par}$                                        |
| $N_{gr}$  | The number of fetching groups             | $N_{gr} = \left\lfloor \frac{N_{if}}{N_{par}} \right\rfloor$ |
| $\alpha$  | The pruning ratio                         | $\alpha = \frac{N_{np}}{N_{par}}$                            |

Table 3.1: Pruning parameters

We should add that this pruning scheme is not the same as *channel pruning* since we do not constrain the same weights to be pruned in each group. Therefore, I did not consider the pruning of the depthwise kernel associated with the pruning of weight in each kernel, as discussed in Section 2.1.2. We proposed then a pruning-scheme that is not coarse-grained (channel pruning), satisfying first design objective: **“The pruning scheme is as fine-grained as possible”**.

## Reduction factors

We can now evaluate the reduction factors of the sparse DSC compared with the DSC without pruning, and with the standard convolution. We consider that the size of the input feature map is  $N_{ix} \times N_{iy} \times N_{if}$ , the size of the depthwise filter is  $N_{kx} \times N_{ky} \times N_{if}$ , the size of the non-pruned pointwise convolution  $N_{if} \times N_{of}$ , the fetching group size is  $N_{par}$ , the number of pruned weights is  $N_{np}$ , and the size of the output FM is  $N_{ox} \times N_{oy} \times N_{of}$ .

The number of weights  $W_{DSC}$  and operations  $O_{DSC}$  of the DSC can be determined using Equation (3.1) and (3.2) [12, 39].

$$W_{DSC} = N_{kx} \times N_{ky} \times N_{if} + N_{if} \times N_{of} \quad (3.1)$$

$$O_{DSC} = N_{ix} \times N_{iy} \times N_{kx} \times N_{ky} \times N_{if} + N_{ix} \times N_{iy} \times N_{if} \times N_{of} \quad (3.2)$$

In the same manner, the amount of weights  $W_{PR\_DSC}$  and operations  $O_{PR\_DSC}$  of the sparse DSC can be determined using Equation (3.3) and (3.4).

$$W_{PR\_DSC} = N_{kx} \times N_{ky} \times N_{if} + N_{np} \times N_{gr} \times N_{of} \quad (3.3)$$

$$O_{PR\_DSC} = N_{ix} \times N_{iy} \times N_{kx} \times N_{ky} \times N_{if} + N_{ix} \times N_{iy} \times N_{gr} \times N_{np} \times N_{of} \quad (3.4)$$

Thus, the weights  $F_{Wg}$  and operations  $F_{Op}$  reduction factors can be expressed according to Equation (3.5) and (3.6). The demonstration allowing to find these relations can be found in Appendix B.

$$F_{Wg} = \frac{W_{DSC}}{W_{PR\_DSC}} = \frac{1 + \frac{N_{kx} \times N_{ky}}{N_{of}}}{\alpha + \frac{N_{kx} \times N_{ky}}{N_{of}}} \quad (3.5)$$

$$F_{Op} = \frac{O_{DSC}}{O_{PR\_DSC}} = \frac{1 + \frac{N_{kx} \times N_{ky}}{N_{of}}}{\alpha + \frac{N_{kx} \times N_{ky}}{N_{of}}} \quad (3.6)$$

As the reduction factors depend on the pruning ratio  $\alpha$  and the number of output channels  $N_{of}$ , we can illustrate the evolution of the reduction factors depending on  $\alpha$  and the different  $N_{of}$  of MobileNetV2, as shown in Figure 3.2.

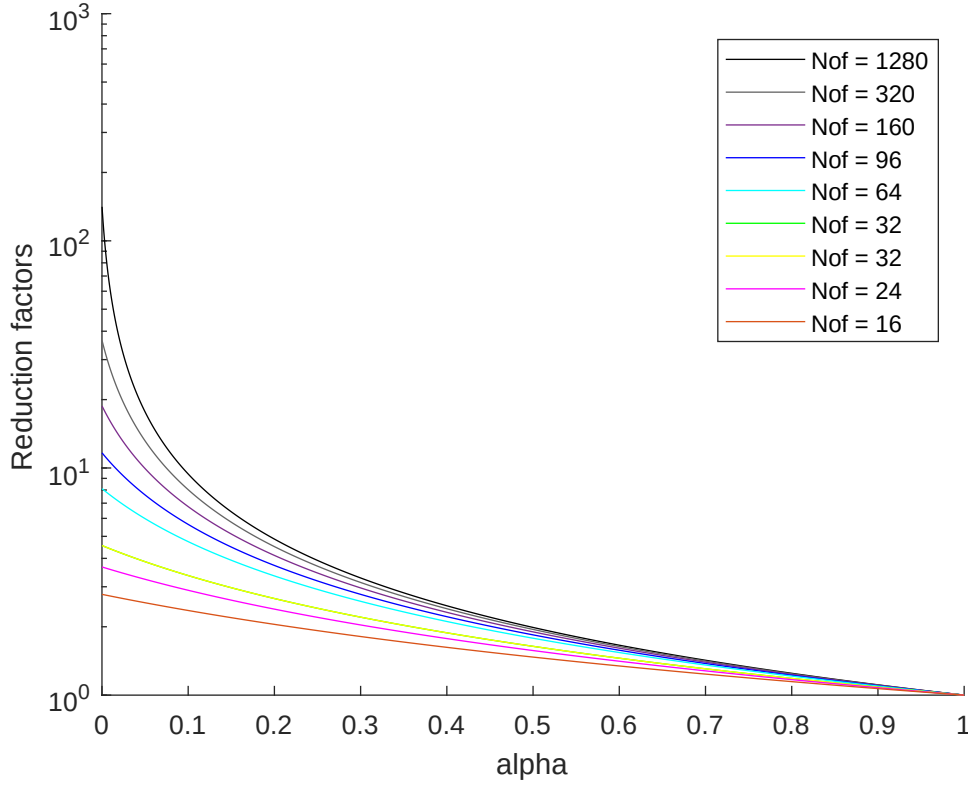


Figure 3.2: Evolution of the reduction factors depending on the pruning ratio and the number of output channels

For example, if we apply a pruning ratio of 25% (meaning keeping 25% of the weights in each fetching group), the reduction factors compared to the DSC without pruning are between two and four times. If we consider the reduction factor between DSC and standard convolution is about 9 times [40], the reduction factors between sparse DSC and standard convolution is between 18 and 36 times (if  $\alpha = 25\%$ ). We, therefore, validated the second design objective: **“The pruning scheme reduces the computational complexity”**.

### 3.1.2 Compressed format

After applying the pruning scheme on the pointwise kernels, these kernels become sparse. This means that they contain zero-value weights, as illustrated in Figure 3.3. Therefore, to reduce memory usage, we should only store each non-pruned weight in a compressed format that allows finding its address with no extra-overhead.

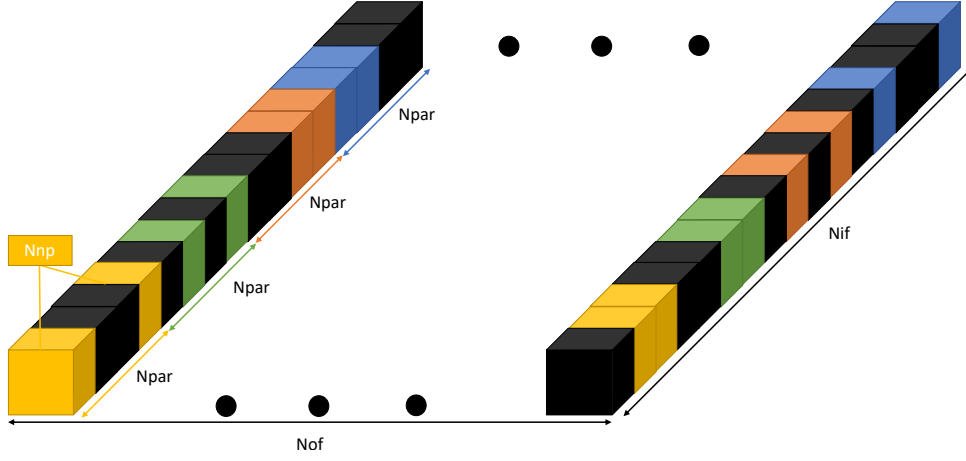


Figure 3.3: A pointwise kernel after pruning, where black cubes are pruned weights

A conventional storage format for a sparse matrix is the Compressed Sparse Row (CSR) [98]. According to Buluç et al. [98], “*The compressed sparse row (CSR) format stores the nonzeros (and ideally only the nonzeros) of each matrix row in consecutive memory locations, and it stores an index to the first stored element of each row*”. In other words, we can represent a sparse matrix using three vectors:

- **Vector val**: containing all the values of the non-pruned weights.
- **Vector col\_ind**: containing the column indices of the elements in **val**.
- **Vector row\_ptr**: containing the index of each row in **val**.

However, as for [35], the CSR format can be compressed further thanks to the pruning scheme applied. Indeed, since the kernels to compress are vectors, we can reduce the storage usage by removing the **row\_ptr** vector (the kernel has only one row).

Moreover, we can have a better compression if we reduce the number of bits allocated to represent the column index of a weight. Indeed, as each weight corresponds to an input fetching group of size  $N_{par}$ , it is sufficient to store the position of the weight in that group, instead of its absolute position in the kernel. As a result, we can reduce the number of bits from  $\log_2(N_{if})$  (absolute position) to  $\log_2(N_{par})$  (relative position), where  $N_{par} \leq N_{if}$ . Moreover, it also reduces the overhead to find the corresponding pixel in a fetching group, for each weight.

In brief, each sparse kernel is converted into 2 vectors of size  $N_{np}$ : vector **val**, which contains the value of the non-pruned weights, and vector **position**, which contains the position of the corresponding pixel in a fetching group. An example is illustrated in figure 3.4.

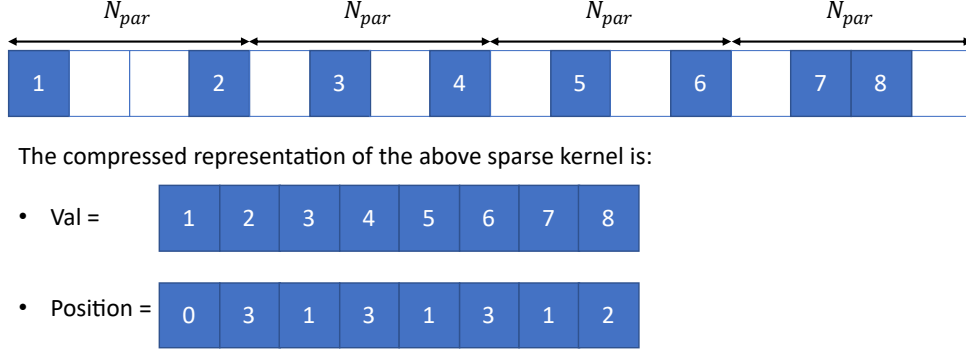


Figure 3.4: Illustration of the compressed format, where  $N_{par} = 4$  and  $N_{np} = 2$

Since this compressed format requires storing the value and the position of each weight (and hence more bits allocated per weight), we computed the maximal pruning ratio, or in the same way, the minimal ratio of pruned weights, required to have a reduction of memory usage. The relation between maximal pruning ratio and memory reduction can be found in Equation (3.7) where  $BW_{weight}$  is the bitwidth required to represent the value of a weight. The demonstration to find the maximal pruning ratio  $\alpha$  to reduce memory usage can be found in Appendix C.

$$\alpha < \frac{BW_{weight}}{BW_{weight} + \log_2(N_{par})} \quad (3.7)$$

The curve representing the minimal ratio of pruned weight, equals to  $1 - \alpha$ , depending on  $N_{par}$  when  $BW_{weight} = 16$  is illustrated in Figure 3.5. We can conclude from the curve that we need to have a ratio of pruned weights to be greater or equal than 40% to save memory. We can also conclude that we can prune fewer weights if we reduce the value of  $N_{par}$ .

We, therefore, validated the third design objective, which is: **“The pruning scheme allows a reduction of the memory required to store the weights”**.

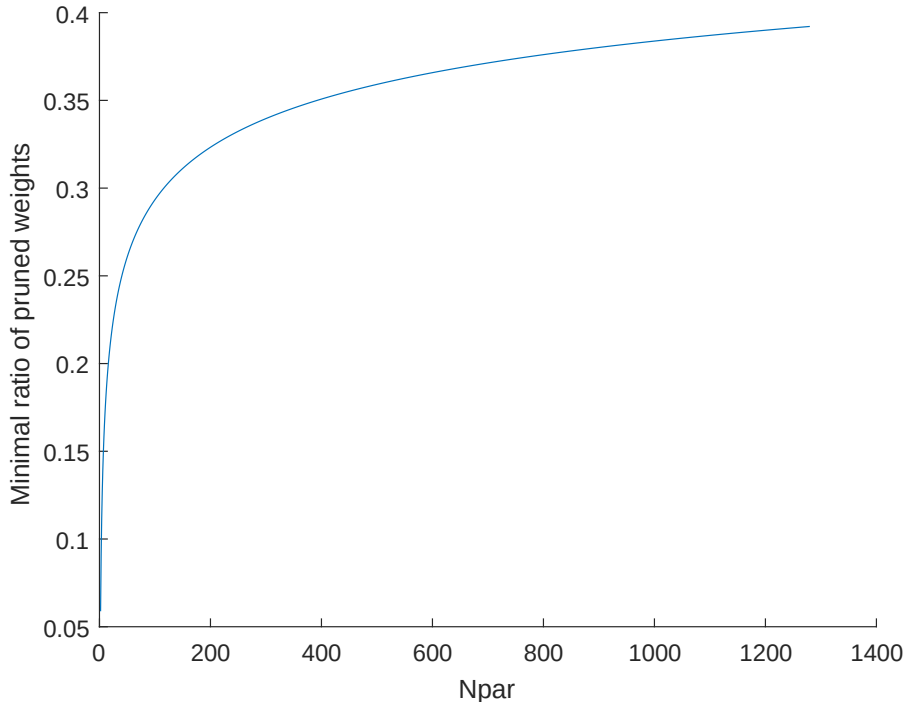


Figure 3.5: Minimal ratio of pruned weights if  $BW = 16$

### 3.1.3 Adding the pruning scheme to MobileNetV2

As we have seen how the weights of a  $1 \times 1$  kernel are pruned and explained the compressed format of the kernels, we present in this section how to integrate the proposed pruning scheme into MobileNetV2. In MobileNetV2, the DSC layer is included in a larger building block, the *Bottleneck residual block* which performs the bottleneck convolution (see Section 1.1.2). In the bottleneck convolution, the DSC follows a  $1 \times 1$  convolution layer that expands the number of input channels by a factor  $t$ . As a result, we have to compute two sparse  $1 \times 1$  convolutions. We note that the FM produced by the  $1 \times 1$  convolution is referenced as **intermediate FM** in the rest of this thesis. The intermediate FM is composed of  $N_{gr_{int}} = t \times N_{gr}$  fetching groups, which are fed to the DSC.

As explained in Section 3.1.1, each  $1 \times 1$  and pointwise convolutions have to fetch  $N_{par}$  input pixels in the channel-axis, which corresponds to  $N_{par}$  channels. Consequently, if the  $1 \times 1$  convolution produces  $N_{par}$  channels, the DSC can fetch those intermediate products to produce partial results of the output FM. Indeed, we can perform the depthwise convolution with each intermediate channel and then perform the pointwise convolution with the corresponding weight in each pointwise kernel, as illustrated in Figure 3.6. The output FM is computed when the DSC has performed these steps for each intermediate group.

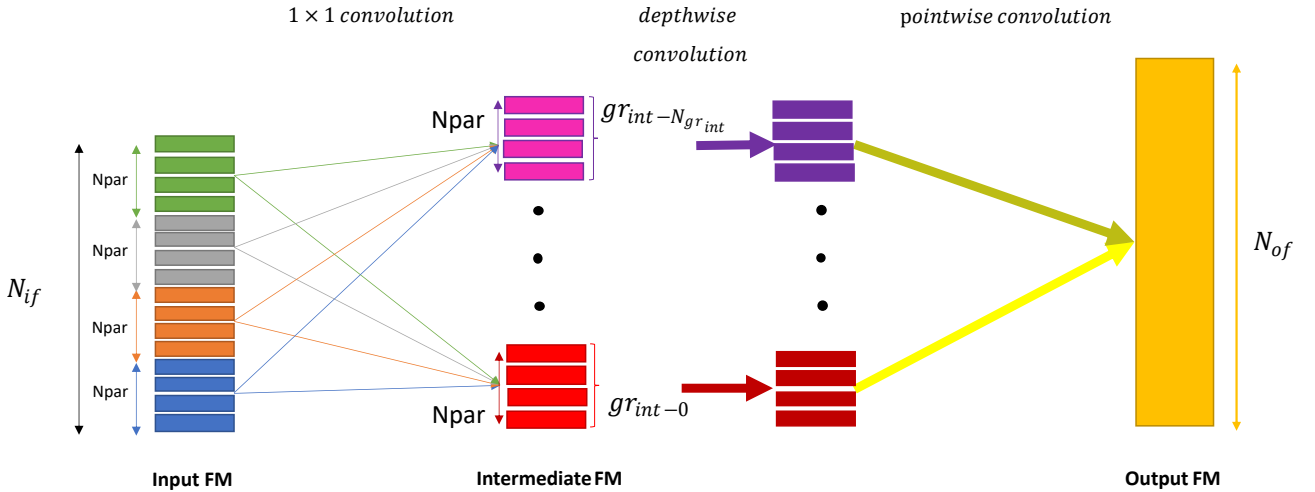


Figure 3.6: Illustration of the algorithm used to perform the convolutions

This approach was chosen, instead of producing directly each final output result, to reuse at most the products of the  $1 \times 1$  convolution. Indeed, if we aim at directly producing a final output, the  $1 \times 1$  convolution has to produce  $N_{kx} \times N_{ky} \times N_{ky} \times N_{par}$  intermediate results. When the DSC has computed the partial sum from this fetching group (for one kernel since we prioritize the computation of one final pixel), the old intermediate pixels will be overwritten by the next fetching group intermediate pixels, which could have been reused for another pointwise kernel.

We can translate this algorithm into pseudocode, observed in Algorithm 1, where *group* (respect. *group<sub>int</sub>*) is the index of the fetching group in the input FM (resp. intermediate FM) and  $N_{gr_{int}} = \left\lceil \frac{N_{if} \times t}{N_{par}} \right\rceil$  is the number of intermediate fetching groups. The detailed algorithm to perform each convolution can be described as follow:

---

**Algorithm 1** Pseudocode of the algorithm

---

```
for  $group_{int} := 0; group_{int} < N_{gr_{int}}; group_{int}++$  do
   $1 \times 1$  convolution ( $group_{int}$ )
  DSC( $group_{int}$ );
end for
```

---

1.  $1 \times 1$  **convolution**: We must fetch the  $N_{par}$  kernels corresponding to the DSC fetching group  $group_{int}$ . Indeed, the DSC needs to fetch  $N_{par}$  channels of the intermediate FM (each kernel produces one channel). Then we can perform the convolution with the input FM. As described in Section 1.1.1, the kernel acts like a sliding window on the input FM. For each pixel at position  $(ix, iy)$ , the convolution loads iteratively each weight and pixel fetching group in the channel-axis. For each fetching group  $group \leq N_{group}$ , the convolution is performed by multiplying each non-pruned weight with its corresponding pixel and accumulates the multiplication with the result of the previous group. The process is finished when the  $N_{par}$  intermediate FM channels have been produced (of size  $N_{ix} \times N_{iy}$ ). The corresponding pseudocode is found in Algorithm 2 and the process is shown in Figure 3.7.

---

**Algorithm 2** Sparse  $1 \times 1$  convolution pseudocode

---

```
for  $int_f := 0; int_f < N_{par}; int_f++$  do ▷ Loop 3
  for  $int_x := 0; int_x < N_{ix}; int_x++$  do ▷ Loop 2
    for  $int_y := 0; int_y < N_{iy}; int_y++$  do ▷ Loop 2
      for  $group := 0; group < N_{gr}; group++$  do ▷ Loop 1
        for  $i_f := 0; i_f < N_{np}; i_f++$  do ▷ Unrolled
           $wgt = filter_{1 \times 1}[group_{int} \times N_{par} + int_f][group \times N_{np} + i_f]$ 
           $acti = FMI_I[wgt[pos] + group \times N_{par}][int_y][int_x]$ 
           $FM_{int}[group_{int} \times N_{par} + int_f][int_y][int_x] += acti \times wgt[val]$ 
        end for
      end for
    end for
  end for
end for
```

---

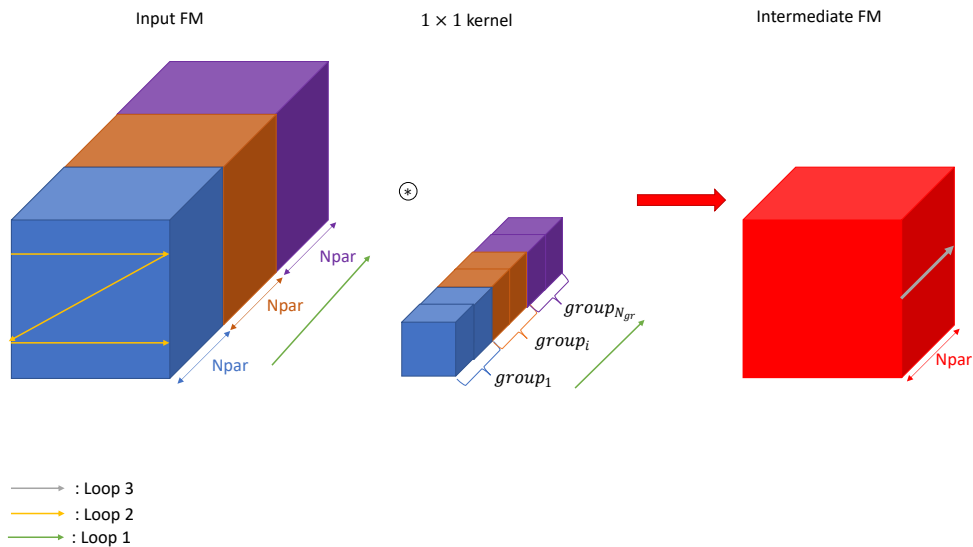


Figure 3.7: Representation of the sparse  $1 \times 1$  convolution

2. **DSC**: Once the  $1 \times 1$  convolution has produced the next  $N_{par}$  intermediate FM channels, we can do the depthwise convolution with these channels. The depthwise convolution first computes the 2D convolution of each of the intermediate FM channels. Afterward, we can load the corresponding weight fetching group in each of the pointwise filters. We can compute the partial convolution for each of the output FM, and each partial result is summed with its corresponding previous partial result. When we have performed the previous step for every intermediate fetching group, the output FM is computed. The corresponding pseudocode is found in Algorithm 3 and the process is shown in Figure 3.8. We note that we have to add padding pixels to the intermediate FM since spatial dimensions are only reduced using stride  $S$  (see Section 1.1.1).

---

**Algorithm 3** Sparse DSC convolution pseudocode

---

```

for  $o_y := 0; o_y < N_{oy}; o_y ++$  do                                ▷ Loop 4
  for  $o_x := 0; o_x < N_{ox}; o_x ++$  do                                ▷ Loop 4
    % Depthwise convolution
    for  $int_f := 0; int_f < N_{par}; int_f ++$  do                        ▷ Loop 3
      for  $k_y := 0; k_y < N_{ky}; k_y ++$  do                            ▷ Loop 2
        for  $k_x := 0; k_x < N_{kx}; k_x ++$  do                            ▷ Loop 2
           $FM_{Dw}[group_{int} \times N_{par} + int_f][o_y][o_x] +=$ 
             $FM_{int}[group_{int} \times N_{par} + int_f][o_y \times S + k_y][o_x \times S + k_x] \times$ 
             $kernel_{dw}[group_{int} \times N_{par} + int_f][k_y][k_x]$ 
        end for
      end for
    end for
    % Pointwise convolution
    for  $o_f := 0; o_f < N_{of}; o_f ++$  do                                ▷ Loop 1
      for  $i_f := 0; i_f < N_{np}; i_f ++$  do                            ▷ Unrolled
         $wgt = filter_{Pw}[o_f][group_{int} \times N_{np} + i_f]$ 
         $acti = FMI_{Dw}[wgt[pos] + group_{int} \times N_{par}][o_y][o_x]$ 
         $FM_o[o_f][o_y][o_x] += acti \times wgt[val]$ 
      end for
    end for
  end for
end for

```

---

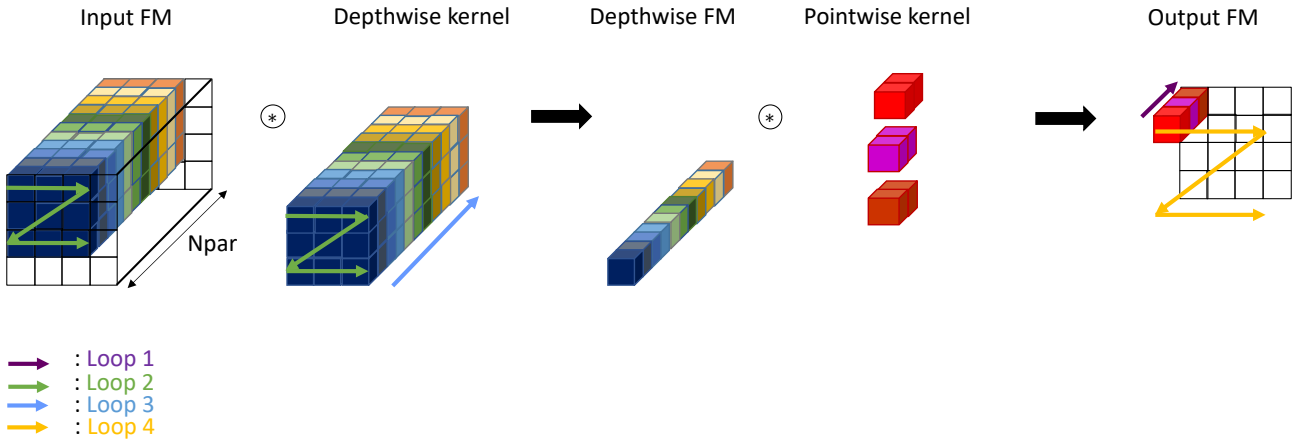


Figure 3.8: Representation of the sparse DSC convolution

### 3.1.4 Loop analysis

We have determined how the bottleneck convolution is going to be performed. Now we define how it will be implemented on FPGA. As explained in Section 2.2, the CNN inference is executed in three steps:

1. The FPGA loads a tile of the data from the external memory into the on-chip memory.
2. The PEs of the FPGA fetch data from the on-chip memory, use them to perform convolution, and store the computed result into the on-chip memory.
3. Once the tile of output results is produced, it is stored into the external memory. Either the FPGA loads the next tile and restarts the previous steps, or it stops if the last tile of data has been convolved.

If we look at Algorithms 2 and 3, each convolution is implemented with various levels of loops, named **convolution loops**. To optimize these operations, three main loop optimization techniques can be used, as presented in Section 2.2: loop tiling, loop unrolling, and loop interchange. Those loop optimization techniques include hardware design parameters that define the acceleration factor and hardware footprint [34]. Therefore, we have to analyze the convolution loops to find the optimal tiling, unrolling, and loop interchange parameters. We applied the same methodology as Ma et al. [34]. However, as they studied the standard convolution, in this work we adapted it to the proposed sparse DSC.[34]

Ma et al. [34] identified design objectives to optimize the convolution operation. These following design objectives should be minimized:

- Partial sum storage.
- On-chip memory accesses.
- Off-chip memory accesses.

We will try to minimize them for both  $1 \times 1$  and depthwise separable convolution.

#### Hardware design parameters

Before analyzing the design objectives, we must first define what are the different tiling and unrolling parameters for each convolution. Tables 3.2 and 3.3 define the different dimensions of the volumes and the corresponding hardware design parameters.

**$1 \times 1$  convolution:** the  $1 \times 1$  convolution is composed of three convolution loops (the fourth loop is fully-unrolled according to the algorithm). During Loop-1, we get each fetching group and we do the convolution between them. Loop-1 is determined by the number of input and weight fetching groups (they are the same since otherwise there would be misalignment). The purpose of Loop-1 is to compute the final output result of the convolution. Loop-2 slides spatially the kernel over the input FM to produce one channel of the intermediate FM. Since the spatial size

of each kernel is  $1 \times 1$ , there is no spatial reduction between the input and intermediate FM. The purpose of Loop-3 is to choose which of the required  $N_{par}$  intermediate FM channels is produced.

|                        | # of weight<br>and input<br>fetching group | Input FM &<br>intermediate FM<br>Spatial Axis | intermediate FM<br>channels |
|------------------------|--------------------------------------------|-----------------------------------------------|-----------------------------|
| Convolution Loops      | Loop-1                                     | Loop-2                                        | Loop-3                      |
| Convolution Dimensions | $N_{gr}$                                   | $N_{ix}, N_{iy}$                              | $N_{par}$                   |
| Loop Tiling            | $T_{gr}$                                   | $T_{ix}, T_{iy}$                              | $T_{intf}$                  |
| Loop Unrolling         | $P_{gr}$                                   | $P_{ix}, P_{iy}$                              | $P_{intf}$                  |

Table 3.2:  $1 \times 1$  convolution loop dimensions and hardware design variables, inspired by [34]

**DSC:** the DSC is composed of four convolution loops. Two loops are specific to the depthwise convolution, one to the pointwise convolution, and one loop is shared between the two convolutions. As presented in Section 3.1.3, the pointwise convolution uses the results of the depthwise convolution to produce  $N_{of}$  partial results to avoid reperforming the same  $1 \times 1$  and depthwise convolutions. Loop-1, therefore, aims at looping over the output channels. The depthwise convolution is composed of two loops. Loop-2 performs the 2D convolution between the input channel and the corresponding kernel and Loop-3 loops over the input channel. Finally, Loop-4 is the one that selects which output pixels to compute and fetches the associated input pixels chunk of data from the on-chip memory.

|                           | Output FM<br>channels | Pointwise<br>kernel size | Intermediate<br>FM channels | Intermediate FM<br>Spatial Axis | Output FM<br>Spatial Axis |
|---------------------------|-----------------------|--------------------------|-----------------------------|---------------------------------|---------------------------|
| Convolution<br>Loops      | Loop-1                | Loop-2                   | Loop-3                      | Loop-4                          | Loop-4                    |
| Convolution<br>Dimensions | $N_{of}$              | $N_{kx}, N_{ky}$         | $N_{par}$                   | $N_{ix}, N_{iy}$                | $N_{ox}, N_{oy}$          |
| Loop<br>Tiling            | $T_{of}$              | $T_{kx}, T_{ky}$         | $T_{intf-dw}$               | $T_{ix}, T_{iy}$                | $T_{ox}, T_{oy}$          |
| Loop<br>Unrolling         | $P_{of}$              | $P_{kx}, P_{ky}$         | $P_{intf-dw}$               | $P_{ix}, P_{iy}$                | $P_{ox}, P_{oy}$          |

Table 3.3: DSC loop dimensions and hardware design variables, inspired by [34]

## Partial Sum Storage

**$1 \times 1$  convolution:** to minimize the partial sum storage, we have to keep all partial results in the registers. If we store those partial results into the on-chip or off-chip memory, memory accesses will be less efficient. The optimal solution would be to fully unroll Loop-1. However, two problems arise. First, since the FPGA has constrained resources, it could not be feasible on the target platform. Second, the number of fetching groups vary across the layers since it depends on the number of input channels ( $N_{gr} = \frac{N_{if}}{N_{par}}$ ). We should adjust the unrolling parameter  $P_{gr}$  to the worst case and there would be inefficiency problems. Indeed, if  $N_{gr} < P_{gr}$ , some PE would not be used. A better solution would be to fully buffer each fetching group and

keep the partial results in the registers. Therefore, we should compute Loop-1 first (otherwise some results would be stored in the on-chip memory). In summary, the number of partial sums is determined by the number of parallel  $1 \times 1$  convolution PEs, as shown in Equation (3.8). Each  $1 \times 1$  convolution PE can operate in parallel  $P_{gr}$  fetching groups. To minimize the storage of partial sum, we should add the following constraint  $T_{gr} = N_{gr}$ . Moreover, the ratio of  $P_{gr}$  to each  $N_{gr}$  of the network should be an integer to avoid these inefficiency problems.

$$\#psum_{1 \times 1} = P_{ix} \times P_{iy} \times P_{intf} \quad (3.8)$$

**DSC:** we can find two kinds of partial sum: the partial depthwise results and the partial output results. According to the proposed algorithm, we can not keep the partial output results in the registers since we produce from each intermediate fetching group all possible partial output results ( $T_{of} = N_{of}$ ). It means that we must keep the tile of output FM in the on-chip memory as partial results, as illustrated in Equation (3.9). Since the kernel size is small and constant across the network, we can fully unroll Loop-2 and set  $P_{kx} = T_{kx} = N_{kx}$ ,  $P_{ky} = T_{ky} = N_{ky}$ . This approach was also chosen by Motamedi et al. [15]. We can also fully unroll Loop-3  $P_{intf-dw} = T_{intf-dw} = N_{par}$ , since the fetching group size  $N_{par}$  is up to the programmer. As the depthwise convolution is fully unrolled, we can determine the number of depthwise partial sums using equation (3.10).

$$\#psum_{DSC_{PW}} = T_{ox} \times T_{oy} \times T_{of} \quad (3.9)$$

$$\#psum_{DSC_{DW}} = N_{kx} \times N_{ky} \times N_{par} \quad (3.10)$$

## On-chip memory accesses

According to Ma et al. [34], the number of on-chip accesses is determined by Equation (3.11), where  $\#read_{px}$  (resp.  $\#read_{wg}$ ) is the number of reads accesses in the on-chip memory for input pixel (resp. weight),  $\#read\_write\_psum$  is the number of write and read accesses if we store the partial sums in the on-chip memory, and  $\#write\_px$  is the number of accesses to write the output results (which is equal to the number of output pixels). Therefore, to reduce on-chip memory accesses, we must limit partial sums stored in on-chip memory and limit the number of reads. The number of reads can be reduced by reusing the data in the registers and can be expressed using Equation 3.12 and 3.13 [34], where  $N_{op}$  is the total number of operations in a convolution. We can analyze for each convolution  $Data\_Reuse_{px}$  and  $Data\_Reuse_{wg}$ .

$$\#On\_chip\_accesses = \#read_{px} + \#read_{wg} + \#read\_write\_psum + \#write\_px \quad (3.11)$$

$$\#read_{px} = \frac{N_{op}}{Data\_Reuse_{px}} \quad (3.12)$$

$$\#read_{wg} = \frac{N_{op}}{Data\_Reuse_{wg}} \quad (3.13)$$

**$1 \times 1$  convolution:** data reuse should be maximized to limit the number of on-chip memory accesses which are less efficient than register accesses. Ma et al. [34] define two kinds of data reuse: spatial reuse (how many data are reused in one cycle) and temporal reuse (if a data can be reused in more than one cycle). Temporal data reuse is only possible if we compute Loop-2 first (we keep the same kernel for the pixels of a channel). However, since we compute Loop-1

first, this is not possible. To increase spatial data reuse, we must increase the parallelization and so the unrolling parameters, as illustrated in Equation (3.14) and (3.15), where  $Data\_Reuse_{px}$  (resp.  $Data\_Reuse_{wg}$ ) is the data reuse for input pixels (resp. weight).

$$Data\_Reuse_{px-1 \times 1} = P_{intf} \quad (3.14)$$

$$Data\_Reuse_{wg-1 \times 1} = P_{ix} \times P_{iy} \quad (3.15)$$

**DSC:** since Loop-3 and Loop-2 are fully unrolled, we have full temporal data reuse for depthwise weights. The data reuse for pixel and weight for the depthwise convolution can be found in Equation (3.16) [34] and (3.17). For the pointwise convolution, we can avoid reading input pixels from on-chip memory if we keep the  $N_{par}$  results of the depthwise convolution in registers. Using the same methodology as for  $1 \times 1$  **convolution**, the weight data reuse can be expressed using Equation (3.18)

$$Data\_Reuse_{px-dsc} = \frac{N_{kx} \times N_{ky} \times N_{par} \times P_{intx} \times P_{inty}}{((P_{intx} - 1)S + N_{kx}) \times ((P_{inty} - 1)S + N_{ky})} \quad (3.16)$$

$$Data\_Reuse_{wg-dw} = N_{kx} \times N_{ky} \times N_{par} \quad (3.17)$$

$$Data\_Reuse_{wg-pw} = P_{ox} \times P_{oy} \quad (3.18)$$

### Off-chip memory accesses

In the model presented in Section 2.2.1, as the on-chip memory is not large enough to store the whole CNN size, this one is stored in external memory. However, accesses to external memory are more expensive in terms of latency and energy. Therefore, we should limit the number of off-chip memory accesses per pixel and weight. Instead, we tile the weights and input FM that we store in the on-chip memory. Once the tile has been fully used, we can load the next tile until the convolution is done.

Since we store every pixel fetching group in the on-chip memory, each pixel is loaded only once from the external memory. However, thanks to padding and the 2D convolution, the input FM tile can be expressed as Equation (3.19) [34]. If the input FM tile does not cover the spatial size of the input FM, some pixels are included in multiple tiles due to the depthwise convolution, as illustrated in Figure 3.9. However, it does not concern every pixel. Therefore, we can express the number of external memory accesses per pixel as Equation (3.20).

$$T_{ix/iy} = (T_{ox/oy} - 1)S + P_{kx/ky} \quad (3.19)$$

$$\#DRAM_{px} = 1 \quad (3.20)$$

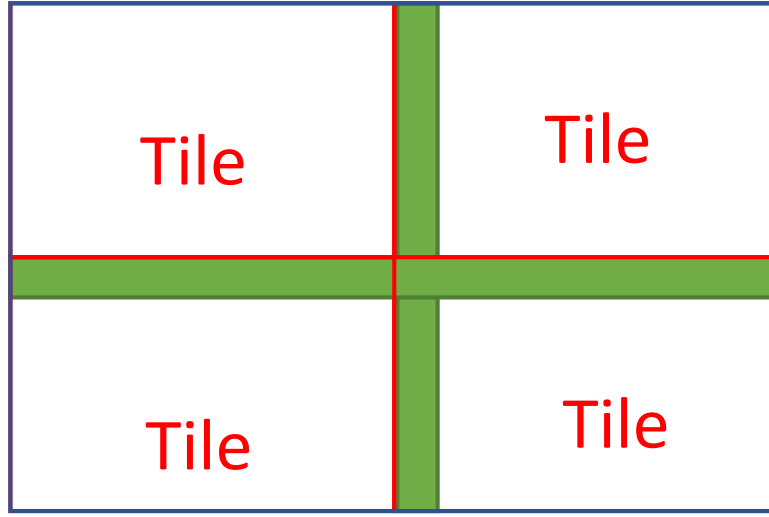


Figure 3.9: Tiling process on an input FM, where the green area is the input pixels that would be loaded more than once due to 2D convolution

If all the weights are not buffered into the on-chip buffer, we have to load from external memory the weight corresponding to the intermediate FM fetching group. It means that we must fetch each weight every time a new output tile is produced. Therefore, the number of weight external memory access is expressed as Equation (3.21).

$$\#DRAM\_wg = \frac{N_{ox} \times N_{oy}}{T_{ox} \times T_{oy}} \quad (3.21)$$

## Summary

After having analyzed the convolution loop, we determined the unrolling and tiling hardware design parameters:

- Loop interchange: for both convolutions, we do the loops in the given order (first Loop-1, ..., and then Loop-n).
- Loop Tiling: as explained previously, the following tiling parameters are set:
  - $T_{gr} = N_{gr}$
  - $T_{intf} = N_{par}$
  - $T_{intf-dw} = N_{par}$
  - $T_{of} = N_{of}$
  - $T_{kx} = N_{kx}$
  - $T_{ky} = N_{ky}$

The other tiling parameters should be set as close as  $N_*$  (depending on the target platform). Moreover, as pointed out by Ma et al. [34], the following constraint should be added to avoid inefficiency: “ $T_*$  should be a divisor of  $N_*$ ”. Indeed, the PEs are configured to execute a full tile. If the constraint is not satisfied, some PEs would not be used.

- Loop Unrolling: as explained previously, the following unrolling parameters are set:

- $P_{kx} = N_{kx}$
- $P_{ky} = N_{ky}$
- $P_{intf-dw} = N_{par}$

The other unrolling parameters should be set as close as  $T_*$  (depending on the target platform). Moreover, as pointed out by Ma et al. [34], the following constraint should be added to avoid inefficiency: “ $P_*$  should be a divisor of  $T_*$ ”.

## 3.2 Implementing sparse DSC on FPGA

In the previous section, we detailed how to prune  $1 \times 1$  kernels, how the bottleneck convolution is computed, and we analyzed the design hardware variables. In this section, we explore how we can implement the chosen design into an FPGA. The hardware design was implemented using SystemVerilog and the corresponding code can be found at this address:

<https://github.com/ggheysen/UCL-EPL-Master-Thesis-2019-2020>.

According to the scope of this thesis, the purpose of the implementation is to verify the fourth and fifth design objectives, namely:

- The proposed architecture provides a logically correct output.
- An increase of the sparsity improves the performance of the architecture.

Those design objectives are related to the implementation of the Bottleneck residual block with sparse  $1 \times 1$  convolution. Therefore, instead of implementing each type of layer of MobileNetV2, we focus only on this one. As those two objectives are independent of the degree of parallelization, it was chosen to set all unconstrained unrolling parameters to 1. We did not implement the skip connection since it is independent of the pruning scheme. However, as demonstrated by Bai et al. [39] and Liu et al. [12], we can extend the DSC PEs to support all kinds of layers of MobileNetV2.

First, we start by describing the overall architecture of the accelerator. Then, we detail more precisely each component defined previously. The performance results of the architecture will be discussed in the next section.

The architecture is inspired by three works: Zhu et al. [35], Bai et al. [39], and Kang [75].

### 3.2.1 Overall architecture

The overall architecture has been inspired by Zhu et al. [35]. The mains components that compose our FPGA-based architecture are the following:

- **External Memory:** it contains the CNN data and it is where the output FM will be stored.
- **Main controller:** it synchronizes the different components of the architecture by sending them control signals.
- **Direct Memory Access (DMA):** it handles the read and write requests between the FPGA and the external memory.
- **PE:** each PE performs the convolution with weights and pixels fetched from external memory into the on-chip memory. In the architecture, we have two kinds of PE:  $1 \times 1$  convolution PE and DSC PE. Since we have defined that each data is represented using a fixed-point number format (see Section 2.1.2), the convolution between weights and pixels is done with integer arithmetic.
- **Buffer:** it composes the on-chip memory and contains a tile of data. There are two categories of buffers: **data buffer** which contains convolution data (pixels and weights) loaded from the external memory, and **result buffer** which contains partial or final results of the  $1 \times 1$  or DSC convolutions. Our architecture is composed of four data buffers ( $FM_I$  buffer,  $Conv_{1 \times 1}$  buffer,  $Conv_{DW}$  buffer,  $Conv_{PW}$  buffer), and two result buffers ( $FM_{int}$  buffer,  $FM_O$  buffer).

An illustration of the overall architecture can be found in Figure 3.10. We should add that each component is synchronous with the clock signal  $clk$  and is reset when the  $rst$  signal is enabled. For the sake of clarity, this was not included in the drawing.

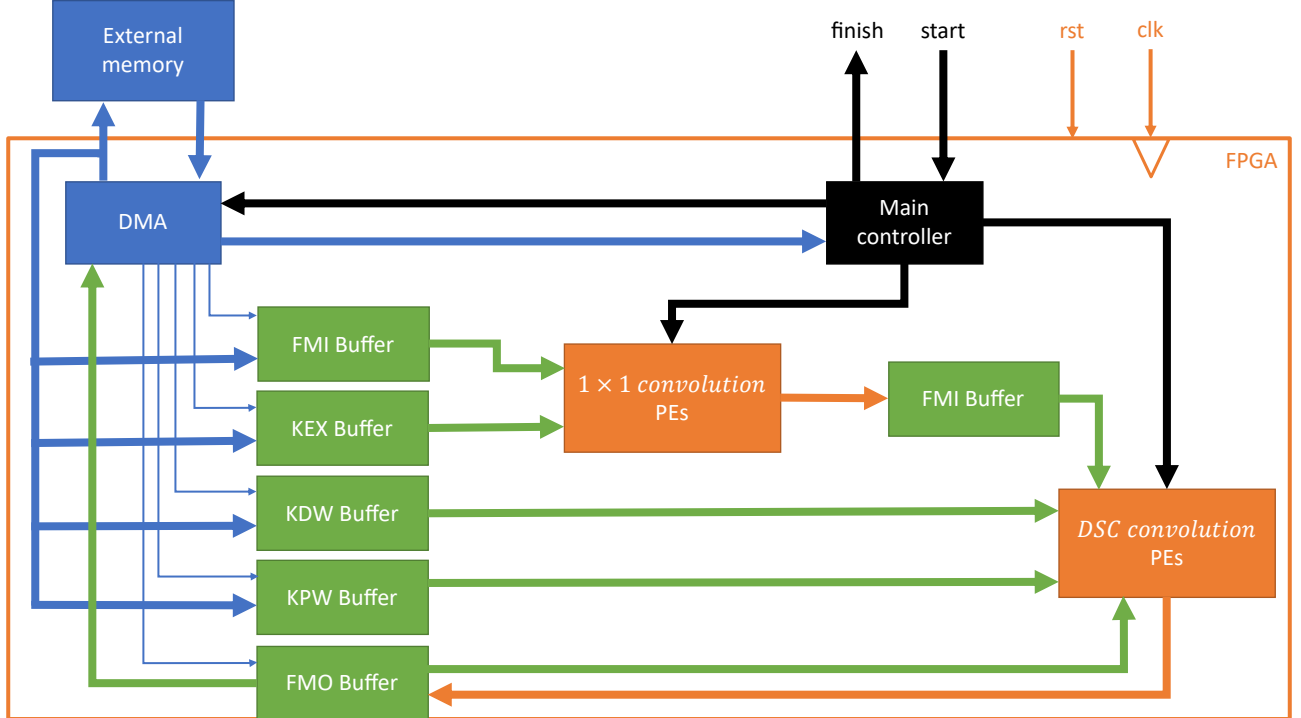


Figure 3.10: Overall architecture of the accelerator

We explain more precisely the behavior of each component in the following sections. An exhaustive list of their different input and output signals can be found in Appendix D.

### 3.2.2 External memory

The external memory is external component (of the FPGA) that contains all the data required to feed the network (MobileNetV2). Each data stored in memory can be identified by its address (either external memory or buffer). We can identify 6 types of data in MobileNetV2:

1. **Input FM:** where the bitwidth required to represent the value of one pixel is equal to  $BW_{pixel}$ . Channels are stored one by one, and the content of one channel is stored line by line. For example, the memory address of the pixel at position  $(ix, iy, if)$  can be expressed using Equation (3.22).

$$address_{FM_I}(ix, iy, if) = ix + iy \times N_{ix} + if \times N_{ix} \times N_{iy} \quad (3.22)$$

2. **Output FM:** the output FM is stored in the same manner as the input FM. We should add that the output FM of one layer becomes the input FM of the next layer. Therefore, the addresses where the output FM pixels are stored become the addresses of the input FM of the next layer and inversely.
3.  **$1 \times 1$  kernel:** the weights of each  $1 \times 1$  convolution (bottleneck layers and  $1 \times 1$  convolution layers), where  $BW_{weight}$  is the number of bits to represent the value of weight and  $\log_2(N_{par})$  the number of bits to represent its position. Each  $1 \times 1$  filter is stored kernel by kernel, and each kernel is stored weight by weight. For example, the memory address of the weight  $x$  in fetching group “group” of kernel  $f$  can be expressed using Equation (3.23).

$$address_{K_{EX}}(kx, group, kf) = kx + group \times N_{np} + kf \times N_{np} \times N_{gr} \quad (3.23)$$

4. **Depthwise kernel:** where the bitwidth representing one weight is equal to  $BW_{weight}$ . Channels are stored one by one, and the content of one channel is stored line by line. For example, the memory address of the weight at position  $(kx, ky, kf)$  can be expressed using Equation (3.24).

$$address_{K_{DW}}(kx, ky, kf) = kx + ky \times N_{kx} + kf \times N_{kx} \times N_{ky} \quad (3.24)$$

5. **Pointwise kernel:** the kernels of the pointwise convolution. The weights are stored in the same manner as for the  $1 \times 1$  filter.
6. **Standard convolution kernel:** since the first layer of MobileNetV2 is a standard convolution [22], the weights must be also stored in the external memory, in the same manner as the depthwise filter.

An offset for each type of data is added to the computation of the address to avoid that one address is shared between multiple data. As a result, before executing one layer, the main controller must have the following information, labeled as **Layer information**:

1.  $N_{ix}$  and  $N_{iy}$ : the spatial dimensions of the input FM. Since  $N_{ix}, N_{iy} \leq 224$  [22], we can use 8 bits to represent them.
2.  $N_{if}$  and  $N_{of}$ : the number of channels of the input and output FM. Since  $N_{if}, N_{of} \leq 1280$  [22], we can use 11 bits to represent them.

3.  **$t$** : the expansion factor of the  $1 \times 1$  convolution. Since  $t \leq 6$  [22], we can use 3 bits to represent it.
4.  **$S$** : the value of the stride. Since  $S$  can be either 1 or 0 [22], we can use 1 bit to represent it.
5.  **$N_{gr}$  and  $N_{gr-int}$** : the number of fetching groups for the  $1 \times 1$  convolution and the DSC. Since the largest value possible for both parameters is the maximum  $N_{if}$  ( $N_{par} = 1$ ), we can use 11 bits to represent them.
6. **Layer**: to tell the main controller which layer is going to be executed. Since there are 4 kinds of layers in MobileNetV2 [22], we use 2 bits to represent it. However, this was not included in the design because only the bottleneck convolution was implemented.
7. **Offset**: the offset of the different data types must be transferred to the DMA to correctly compute the address of the required data. The number of bits required to represent one offset is the bitwidth of the address of the external memory.

As a consequence, those layer information are also stored in the external memory. The overall structure of the external memory can be found in Figure 3.11.

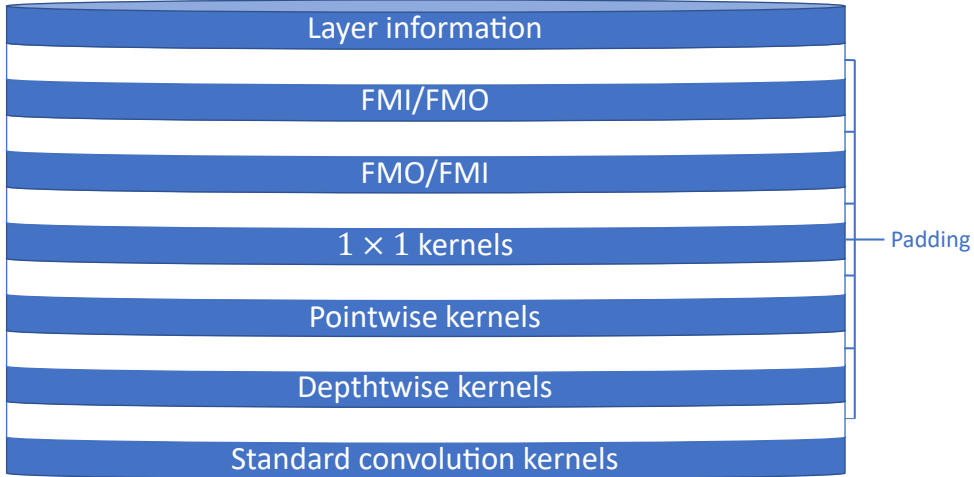


Figure 3.11: overall structure of the external memory

We can now evaluate the total memory size required to hold the CNN in external memory. Since this depends on the pruning parameters, we compute the upper-bound of memory required ( $\alpha = 1, N_{par} = N_{if-max}$ ). For the same reason, we also set the memory required to store each filter or FM as the memory usage of the largest element of the same type across all layers of the network. Moreover, according to [99], the maximum external memory size for a Cyclone V FPGA supported is  $4GB$ , and hence the maximum bitwidth of the offset is  $32\ bits$ . Similarly, the maximum  $BW_{pixel}$  and  $BW_{weight}$  are equal to  $32\ bits$ .

As a result, the memory usage of the network depending on the bitwidth used to represent each pixel and weight is expressed in Equation (3.25). The curve representing the maximal external memory required can be found in Figure 3.12. We can conclude that we need external memory size of at least  $70\ MB$ , which is smaller than the maximal external memory of Cyclone V FPGA

(4GB).

$$\begin{aligned}
Max\_Mem = & \left( \#layers_{tot} \times BW_{layer\_information} \right) \\
& + (Max\_FMI \times BW_{pixel}) \\
& + (Max\_FMO \times BW_{pixel}) \\
& + (\#layers_{standard-conv} \times Max\_KSC\_filter \times BW_{weight}) \\
& + (\#layers_{bottleneck-conv} \times Max\_KDW\_filter \times BW_{weight}) \\
& + (\#layers_{bottleneck-conv} \times Max\_KPW\_filter \times \alpha \times (BW_{weight} + \log_2(N_{par}))) \\
& + ((\#layers_{bottleneck-conv} + \#layers_{1-1-conv}) \times \\
& \quad \alpha \times Max\_K11\_filter \times (BW_{weight} + \log_2(N_{par})))
\end{aligned} \tag{3.25}$$

$$\begin{aligned}
\#layers_{tot} &= 21 \\
\#layers_{standard-conv} &= 1 \\
\#layers_{bottleneck-conv} &= 17 \\
\#layers_{1-1-conv} &= 2 \\
BW_{layer\_information} &= 258 \text{ bits} \\
Max\_FMI &= 112 \times 112 \times 32 = 401408 \\
Max\_FMO &= Max\_FMI \\
Max\_K11\_filter &= 320 \times 1280 = 409600 \\
Max\_KDW\_filter &= 3 \times 3 \times 6 \times 160 = 8640 \\
Max\_KPW\_filter &= 6 \times 160 \times 320 = 307200 \\
Max\_KSC\_filter &= 3 \times 3 \times 32 = 288
\end{aligned}$$

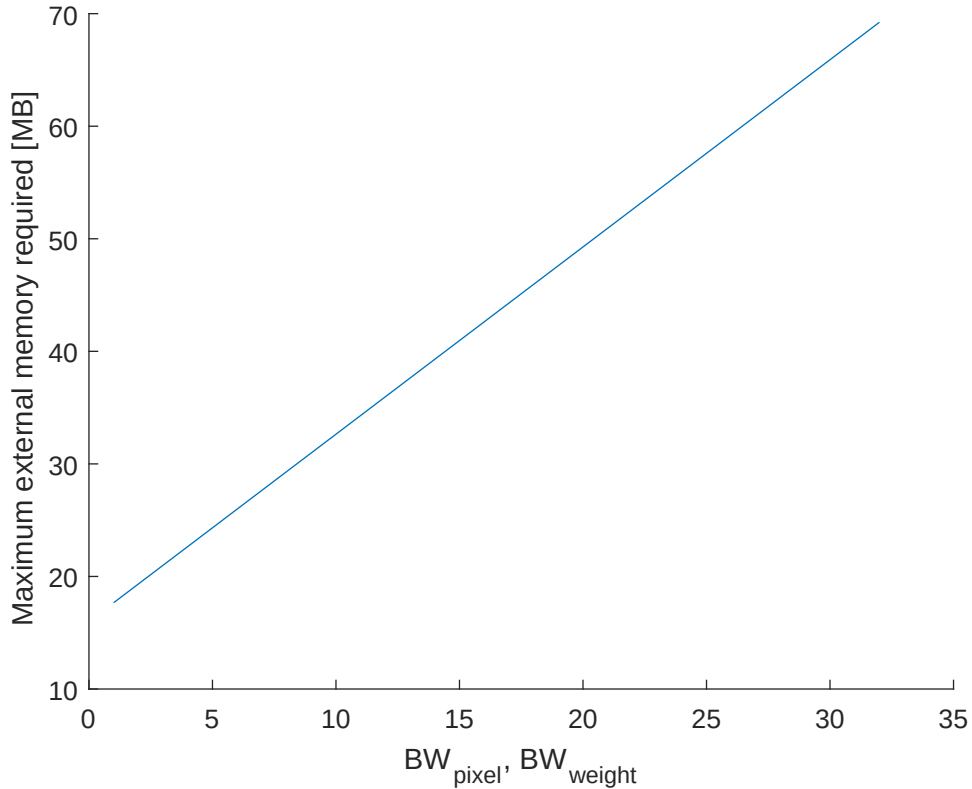


Figure 3.12: Maximum external memory required in *MB* in relation with the bitwidth of each data  $BW_{pixel}$  and  $BW_{weight}$

The maximum memory required can be reduced with pruning. For example, by modifying Equation (3.25), with  $N_{par} = 8$ ,  $N_{np} = 4$ , and  $BW_{weight} = BW_{pixel} = 16$ , the total memory required is equal to 16.54 MB.

### 3.2.3 Main controller

The role of the main controller is to synchronize the different components of the architecture (except the memory components). Therefore, each component is a “slave” that stays in an idle state until the main controller wakes up one of them to perform a particular operation.

To wake up a component, the main controller sends it a starting signal  $s_*$  and optionally additional information about the operation to perform. Afterward, the main controller remains idle, waiting for the component to terminate its operation. Once the component has completed its task, it sends a finishing signal  $f_*$  to the main controller. The main controller can then wake up a component to perform the next task.

The main controller remains in an idle state until it receives a starting signal indicating that the external memory contains all data to perform a bottleneck convolution. I have implemented the main controller to compute one bottleneck convolution, but its architecture could be extended to iterate through all the layers of the network.

We can describe the behavior of the main controller using a Final State Machine (FSM), as illustrated in Figure 3.13. For the sake of clarity, additional information transmitted to the components are not shown in the Figure. The **IDLE** state is the state where the main controller is waiting to perform a bottleneck convolution. Once the main controller has received a starting signal, it performs the following operations:

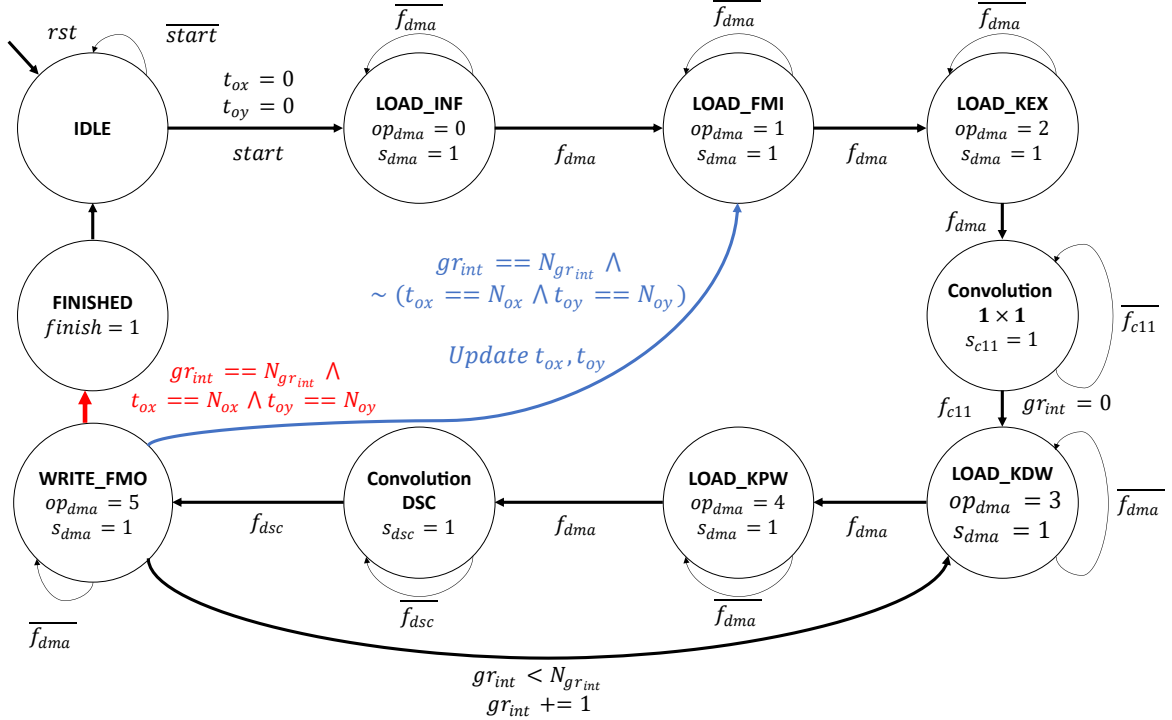
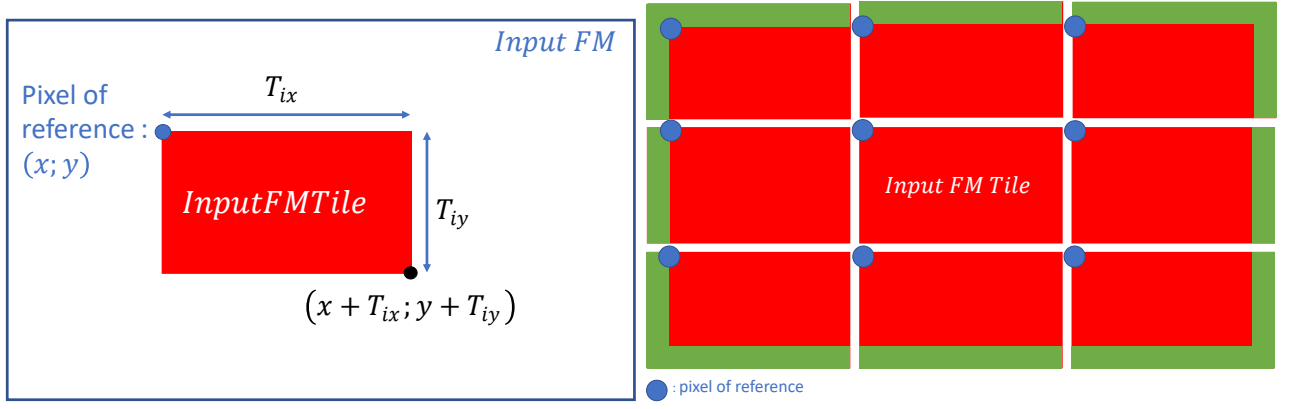


Figure 3.13: FSM of the main controller

1. **LOAD\_INF**: to perform the bottleneck convolution, the main controller first needs to fetch the layer information stored in the external memory. Therefore it tells the DMA to fetch them ( $op_{dma} = 0$ ).
2. **LOAD\_FMI**: the bottleneck convolution starts by fetching a tile of the input FM from the main memory. As all input pixels in the channel-axis are buffered (corresponding to all the fetching groups), an input FM tile can be determined only by the spatial coordinates of the tile first element - pixel of reference (with the smallest  $(x, y)$ ). Indeed, the DMA can fetch, for each channel, all pixels in the range  $[(x, y); (x + T_{ix}, y + T_{iy})]$ , corresponding to the input FM tile, as illustrated in Figure 3.14a.  
  
As a result, when the main controller asks the DMA to load a tile of input FM into the on-chip memory ( $op_{dma} = 1$ ), it also transfers the spatial coordinates and the memory address of its pixel of reference. Moreover, since we have to add padding to the tile (spatial dimensions are not reduced by the convolution operations), zero-value pixels are stored in the on-chip memory at corresponding positions when the pixel to fetch is outside the input FM dimension, as shown in Figure 3.14b.
3. **LOAD\_KEX**: after fetching the input FM tile, the main controller asks the DMA to load the  $N_{par} \ 1 \times 1$  kernels into the on-chip memory ( $op_{dma} = 2$ ), corresponding to the intermediate fetching group  $group_{int}$ . Since the filter is stored kernel by kernel, we only have to transfer the external memory address of the first weight of the kernel producing the first intermediate channel in the intermediate fetching group.
4. **CONV\_11**: after loading the weights and pixels, the  $1 \times 1$  convolution can be performed. Once the  $N_{par}$  intermediate channels corresponding to the intermediate fetching group  $group_{int}$  have been produced, the DSC can be executed.
5. **LOAD\_KDW** and **LOAD\_KPW**: before doing the DSC, the pointwise and depthwise kernels corresponding to the intermediate fetching group should be loaded by the DMA ( $op_{dma} = 3$  and  $op_{dma} = 4$ ). Depthwise kernels are fetched in a similar way as fetching the  $1 \times 1$  kernels. Since we have to load from the external memory the same fetching group from each pointwise kernel, then we simply have to transfer the address of the first weight in the corresponding fetching group.
6. **CONV\_DSC**: all output FM tile partial products can be computed by performing the DSC on the  $N_{par}$  intermediate channels produced by the  $1 \times 1$  convolution. When all intermediate fetching groups have been processed, final results are computed and can be written into the external memory. Otherwise, the next  $N_{par}$  intermediate channels should be computed. Moreover, since output FM partial results will be kept in on-chip memory, the main controller indicates to the DSC PEs whether the values read from the *FMO Buffer* should be considered as valid. If it is not the case, the value should be set to 0.
7. **WRITE\_FMO**: this state indicates that there are only final results in the *FMO Buffer*. The main controller tells the DMA that its content can be written to the external memory ( $op_{dma} = 5$ ). As for the input FM tiles, the output FM tiles are determined by their pixel of reference, which is also transmitted to the DMA. If all tiles have been processed, the bottleneck convolution is completed and the main controller state turns to **FINISHED**. Otherwise, a new tile has to be processed and the main controller returns to **LOAD\_FMI** state.
8. **FINISHED**: The bottleneck convolution is done and the main controller enables the finish signal.



(a) Example of the pixel of reference for an input FM tile (b) Padding configuration of input FM tiles, where red area represents pixels fetched from external memory and green area represent padding pixels

Figure 3.14: Illustration of the concept of the pixel of reference and padding

### 3.2.4 DMA

The purpose of the DMA is to fill the **data buffer** with its corresponding tile fetched from external memory and write the output pixels into the external memory. Therefore, it has to manage six types of operation, referenced by its operation number  $op_{dma}$ :

- $op_{dma} = 0$ : Load the layer information
- $op_{dma} = 1$ : Load an input FM tile, referenced by its pixel of reference. The DMA needs as extra information the coordinate and the memory address of that pixel of reference.
- $op_{dma} = 2$ : Load the  $1 \times 1$  kernels used to produce the intermediate fetching group  $group_{int}$  of size  $N_{par}$ . The DMA needs as extra information the first kernel and the memory address of its first weight to correctly load the  $N_{par}$  kernels.
- $op_{dma} = 3$ : Load the depthwise kernels corresponding to the intermediate fetching group  $group_{int}$  of size  $N_{par}$ . The DMA needs as extra information the first kernel of the group and the memory address of its first weight to correctly load the  $N_{par}$  kernels.
- $op_{dma} = 4$ : Load the weight fetching group (of size  $N_{np}$ ) of all pointwise kernels corresponding to the intermediate fetching group  $group_{int}$ . The DMA needs as extra information the position of the first weight in the fetching group and its memory address.
- $op_{dma} = 5$ : Write the content of the **FMO Buffer** into the external memory, which corresponds to a tile of output FM. As this tile is referenced by its pixel of reference, the DMA needs as extra information the coordinate and the memory address of that pixel of reference.

When the main controller asks the DMA for a transfer, it sets its start signal, which operation to execute ( $op_{dma}$ ), and all extra information related to the external memory required. If the DMA reads or writes into a buffer, the initial address  $r_{addr-i}/w_{addr-i}$  can be set to 0. Indeed, buffer addresses start from 0 to  $Buffer_{N_{elem}} - 1$ .

The behavior of the DMA can be expressed using an FSM, as found in Figure 3.15. A general FSM is illustrated because each operation shares the same structure. As the DMA only fetches data from one memory and writes the data into another one, each operation is composed of 2 states:

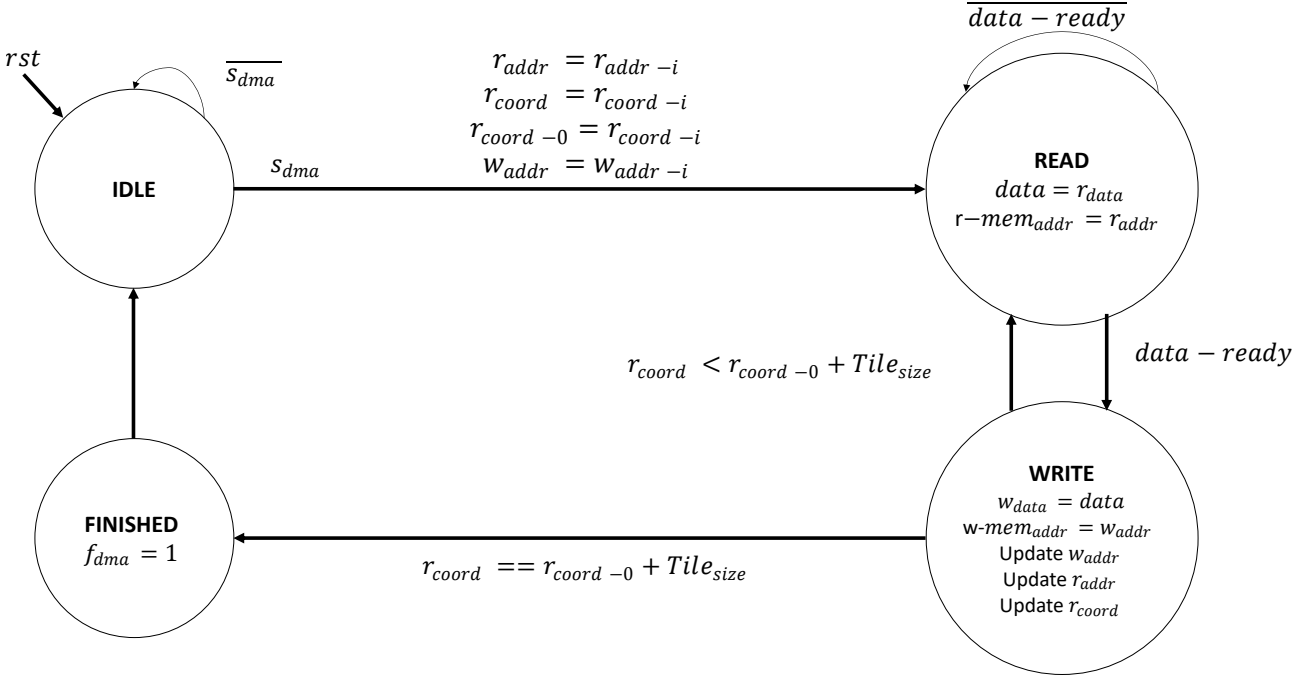


Figure 3.15: FSM of the DMA

- **READ**: the DMA fetches the required data at an address  $r_{addr}$ . Once the data is loaded by the DMA, it moves to the next state to write the data in the on-chip memory.

When loading the input FM tile, if the address corresponds to a padding pixel, the value to write is 0 and the DMA directly moves to the **WRITE** state.

If the data is fetched from the external memory, the DMA sets the read request signal  $r_{request_{extmem}}$  (enabling a transfer) and waits for the data read from the external memory to be valid ( $r_{valid_{extmem}}$ ). But if the data is fetched from a buffer, the process can be pipelined. That is why, before the DMA reads data from the on-chip memory, it is in a state called **RAM\_LOADING** to initialize the pipeline process (not illustrated).

- **WRITE**: the DMA sends a write signal and the associated data  $w_{data}$  and address  $w_{address}$  to the corresponding memory. If the operation is finished (the tile has already been fetched), the DMA goes into the **FINISHED** state. Otherwise, it updates the address of the next data to fetch and goes back to the **READ** state.

The DMA reads the external memory data with address  $address_{extmem}$  sent with signal  $r_{extmem}$ . Since the DMA performs one transfer at a time, we can share the fetched data (to be written)  $w_{data}$  between all memory components and we set the write signal to the corresponding component. Similarly, the signal to access or write data in a buffer  $ram_{addr}$  can also be shared between the different buffers. As the DMA reads the content of only one buffer, we define the corresponding signal as  $r_{ram}$  (and hence at address  $ram_{addr}$ ).

### 3.2.5 1×1 convolution PE

The purpose of this component is to execute Algorithm 2. The behavior of the PE can be expressed using an FSM, as found in Figure 3.16.

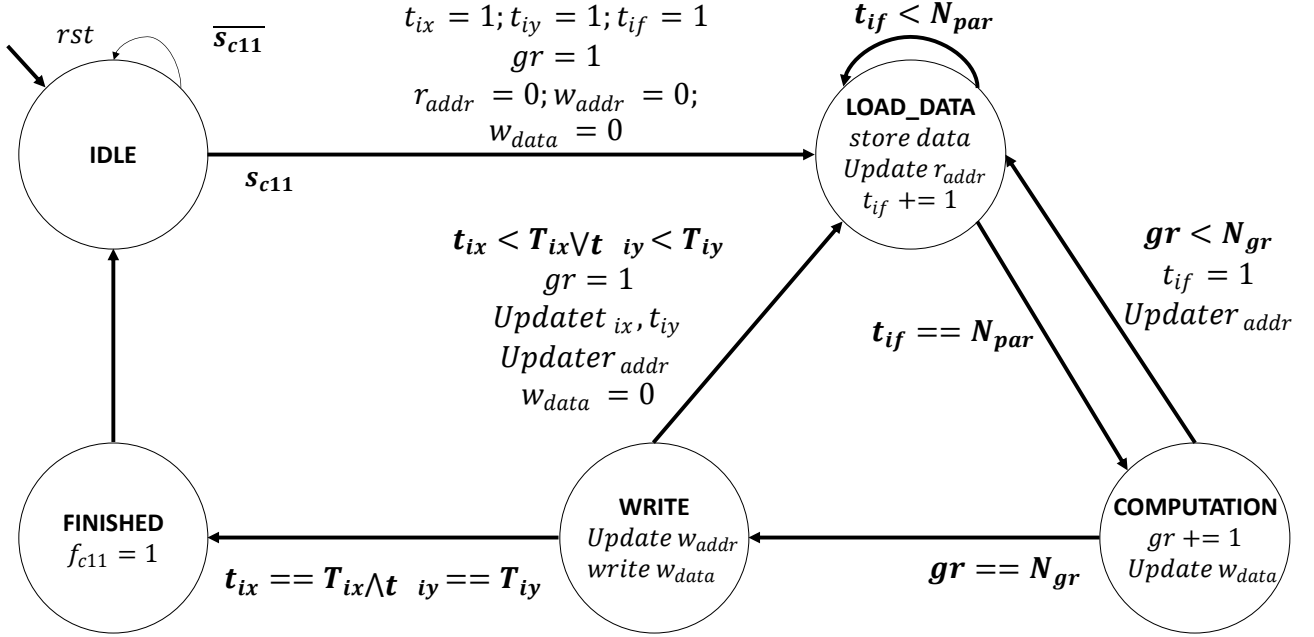


Figure 3.16: FSM of the 1×1 convolution PE

When the PE receives a starting signal, it means that the **FMI BUFFER** and **KEX BUFFER** contain the data to execute the next  $1 \times 1$  convolution. For each fetching group corresponding to an intermediate FM pixel at address  $w_{addr}$ , the PE loads the corresponding  $N_{par}$  inputs and  $N_{np}$  weights into its registers (**LOAD\_DATA** state).

Once it is done, the computation can be performed (**COMPUTATION** state). Since the convolution is fully-unrolled, we can use the PE structure from Kang [75], which is shown in Figure 3.17. The output result is then summed with the content of the accumulator (which is initialized at zero for the first fetching group).

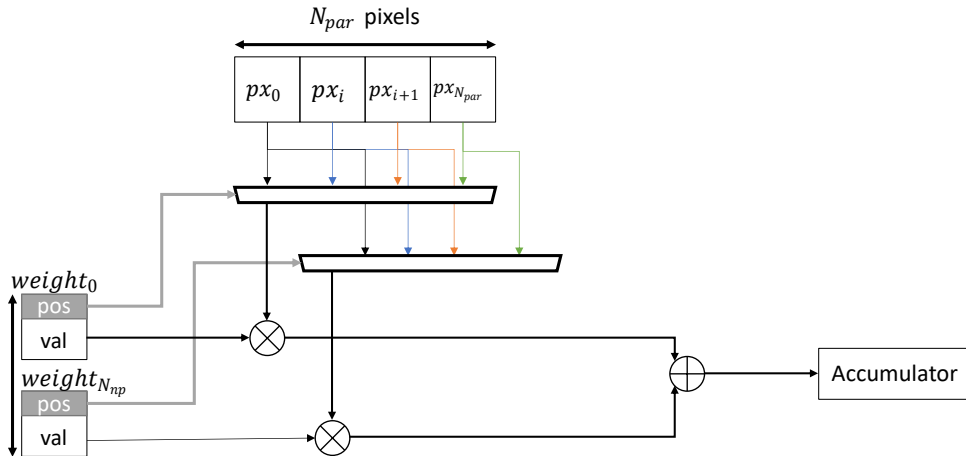


Figure 3.17: PE structure performing the convolution where  $N_{par} = 4$  and  $N_{np} = 2$ , inspired by [75]



two different states because since the depthwise convolution is fully-unrolled, all weights are stored into the registers. We have to only fetch them once from the external memory.

- After the input pixels and the depthwise weights have been loaded, the depthwise convolution can be performed. To execute the depthwise convolution for each intermediate channel, we can use the same PE used by Bai et al. [39], as illustrated in Figure 3.19. First, we do an element-wise multiplication using a multiplier array between pixels and weights (**DW\_CONV\_MUL** state). Then, we simply have to sum the products corresponding to the same channel to obtain the  $N_{par}$  depthwise products used as input for the pointwise convolution (**DW\_CONV\_ADD** state). Then the registers can store the results of the ReLU6 activation function with the obtained results to be used as input for the pointwise convolution.
- We can perform the pointwise convolution to produce partial results of the  $N_{of}$  channels of the spatial coordinate  $o_x, o_y$ . For each kernel, we load the  $N_{np}$  weights corresponding to the intermediate fetching group (**LOAD\_K\_PW** state). We also load the partial sum stored in the **FMO Buffer** to sum it with the results of the convolution (**PW\_CONV** state). However, if it is the first intermediate fetching group, this value is set to 0 (*first\_par\_i* signal, set by the main controller).

The structure of the PE performing the pointwise convolution is the same as in Figure 3.17. However, the sum between the partial sum stored and the result of the convolution is stored in the **FMO Buffer** instead of a register (**WRITE** state).

- When all the  $N_{of}$  partial sums have been produced, the PE moves to the **FINISHED** state and sets its finishing signal.

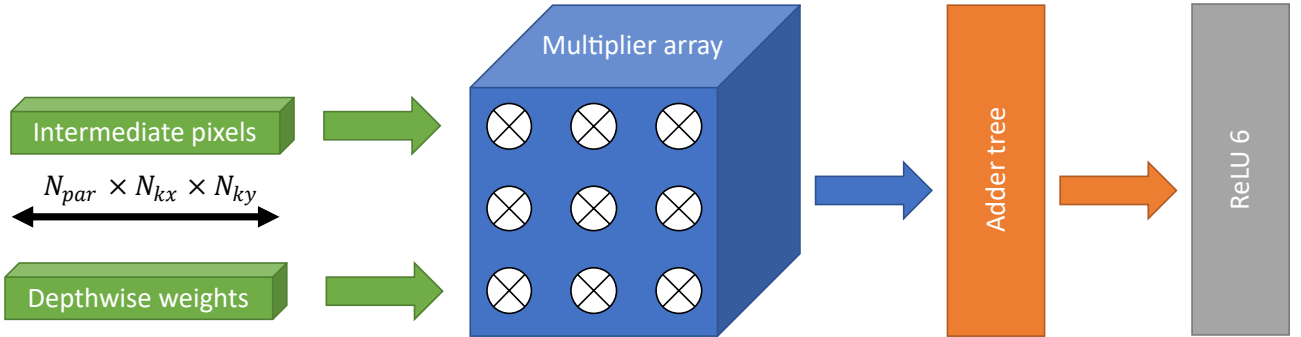


Figure 3.19: PE structure performing the depthwise convolution, inspired by [39]

The DSC PE could be extended to support the standard convolution, as pointed out by Bai et al. [39]. Indeed, if we consider summing all products in the multiplier array instead of only those belonging to the same channel, we perform the standard convolution.

### 3.2.7 Buffer

The buffers compose the on-chip memory of the FPGA. As mentioned previously, the structure is composed of six buffers that share the same structure, as illustrated in Figure 3.20. At each clock period, the output signal *res* is equal either to the value stored at address *addr*, or if the write signal *write* is enabled, to the value that is currently written. In the same way, if the

write signal is set, the value in the buffer at address  $addr$  is replaced by the input data  $w_{data}$ . As multiple components may read one same buffer, a multiplexer is added to select from which address to read or write.

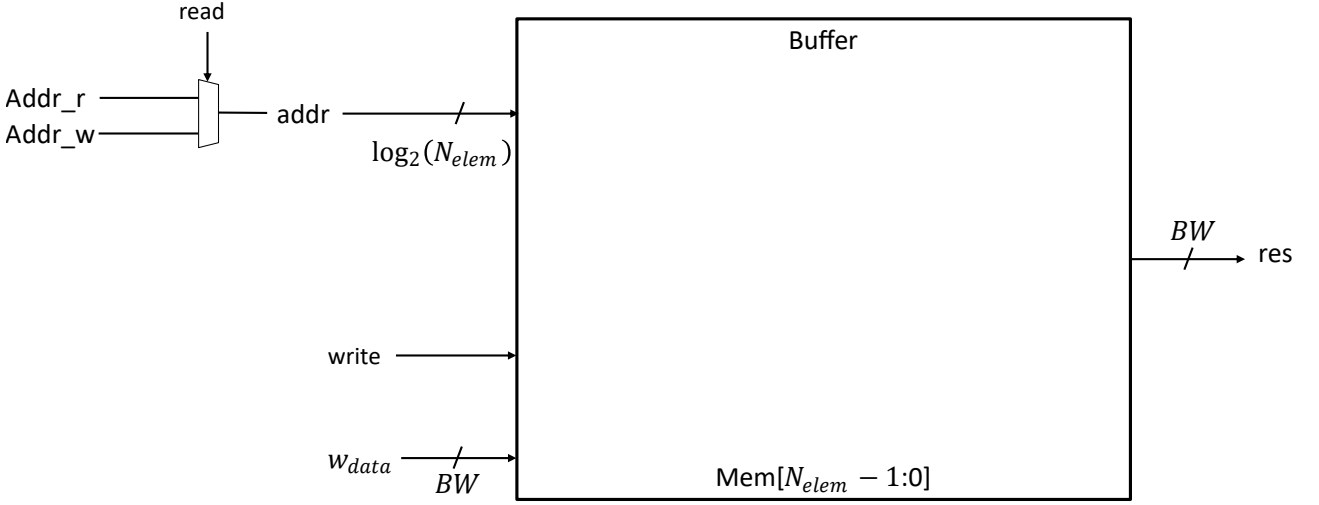


Figure 3.20: General architecture of a buffer component

The value of  $addr_r$  and  $addr_w$  depends on the type of buffer. For **data buffer** the  $addr_r$  is the PE address of the corresponding buffer, and  $addr_w$  is the DMA  $ram_{addr}$  signals. For **result buffer**,  $addr_w$  is the corresponding PE output address and  $addr_r$  can be either a PE address requesting a result (**FMINT Buffer** and **FMO Buffer**) or the DMA  $ram_{addr}$  signals to write final results to external memory (**FMO Buffer**)

However, the main differences between each buffer are the maximum number of elements that can be stored  $N_{elem}$  and the bitwidth  $BW$  of their data stored. We explore for each buffer these two parameters.

- **FMI Buffer:** we have determined from the loops analysis that we buffer each input FM fetching group. Since the number of input FM channels vary across the different layers of the network, we can determine  $N_{elem}$  using Equation (3.26). The number of bits to store a pixel is equal to  $BW_{pixel}$ . We can therefore express  $BW_{fmi}$  such as Equation (3.27).

$$N_{elem-fmi} = \forall l \in layers : Max \left( N_{if}^l \times Min \left( T_{ix}, N_{ix}^l \right) \times Min \left( T_{iy}, N_{iy}^l \right) \right) \quad (3.26)$$

$$BW_{fmi} = BW_{pixel} \quad (3.27)$$

- **KEX Buffer:** this buffer stores the  $N_{par} \times 1 \times 1$  kernels required to perform this convolution, which expands the number of input channels. Therefore, we can express  $N_{elem}$  using Equation (3.28), where  $N_{gr-max} = \left\lceil \frac{1280}{N_{par}} \right\rceil$  is the maximum number of fetching groups in the whole network. Since the weights are expressed in the proposed compressed format, we also have to store the weight position in the fetching group. As a result,  $BW_{KEX}$  can be expressed using Equation (3.29), where  $BW_{weight}$  is the number of bits to represent a weight and  $\log_2(N_{par})$  the number of bits to represent the position.

$$N_{elem-keX} = N_{par} \times N_{np} \times N_{gr-max} \quad (3.28)$$

$$BW_{keX} = BW_{weight} + \log_2(N_{par}) \quad (3.29)$$

- **FMINT Buffer:** since the DSC needs  $N_{par}$  intermediate FM channels and the  $1 \times 1$  convolution does not reduce the spatial dimension of the input FM, we can express the  $N_{elem}$  using Equation (3.30). The bitwidth of a pixel is the bitwidth used to represent a pixel, as in Equation (3.31).

$$N_{elem-fmint} = N_{par} \times T_{iy} \times T_{ix} \quad (3.30)$$

$$BW_{fmint} = BW_{pixel} \quad (3.31)$$

- **KDW Buffer:** applying the same methodology as **FMINT Buffer** and since the depth-wise kernels are fully buffered, we can express the  $N_{elem}$  using Equation (3.32). The bitwidth of a pixel is the bitwidth used to represent a weight, as in Equation (3.33).

$$N_{elem-kdw} = N_{par} \times N_{ky} \times N_{kx} \quad (3.32)$$

$$BW_{kdw} = BW_{weight} \quad (3.33)$$

- **KPW Buffer:** for each time we perform a DSC, we need to convolve in each pointwise the corresponding weight fetching group with  $N_{np}$  pixels in the  $N_{par}$  channels. The maximum number of elements in the buffer can be computed using Equation (3.34). As the pointwise convolution is a  $1 \times 1$  convolution, the bitwidth associated to each pointwise kernel is the same as the **KPW Buffer** and can be expressed using Equation 3.35

$$N_{elem-kpw} = N_{np} \times N_{of} \quad (3.34)$$

$$BW_{kpw} = BW_{weight} + \log_2(N_{par}) \quad (3.35)$$

- **FMO Buffer:** using the same methodology as for determining  $N_{elem}$  and  $BW$  for the **FMI Buffer**, they are expressed using Equation (3.36) and (3.37)

$$N_{elem-fmo} = \forall l \in layers : \text{Max} \left( N_{of}^l \times \text{Min} \left( T_{oy}, N_{oy}^l \right) \times \text{Min} \left( T_{ox}, N_{ox}^l \right) \right) \quad (3.36)$$

$$BW_{fmo} = BW_{pixel} \quad (3.37)$$

### 3.3 Results and discussion

In the previous section, we detailed our implementation based on the pruning scheme and the algorithm developed in Section 3.1. In this section, we determine if the design objectives are met by measuring the outputs and performance of the proposed accelerator.

First, we describe the experimental setup to measure the performances of the proposed architecture. Second, we show the obtained results and what observations can be made. Finally, a discussion of the obtained results is done.

#### 3.3.1 Experimental setup

After having developed the architecture on SystemVerilog, we can perform a functional simulation using a testbench (see Section 1.2). This functional simulation allows us to first verify if the

output of the design is logically correct (fourth design objective) and if the pruning increases the performance of the inference by measuring the number of cycles required to execute the layer (fifth design objective). Indeed, if we increase the sparsity, we should expect a reduction in the number of cycles to execute the inference.

This functional simulation was done using MODELSIM-ALTERA, an HDL simulation environment. The software simulates the testbench and proposed accelerator as implemented on an FPGA. Using this simulation, we can extract the required results.

Before checking if the design objectives are met, we must choose the target platform on which the design is implemented. According to Liu et al. [12], the most commonly used architecture for DL is a heterogeneous system. It combines both a CPU and an accelerator (FPGA). Therefore, we chose DE10-Nano for this reason. This platform integrates a Hard Processor System (HPS) (Dual-core ARM\* Cortex\*-A9 MPCore), an FPGA (Cyclone V 5CSEBA6U23I7N), and a 1GB DDR3 memory [100]. The external memory is large enough to store the CNN data, according to the conclusions made in Section 3.2.2. However, in the frame of the thesis, we focus only on the accelerator part.

Since we perform a functional simulation of the accelerator part of the design, we have to also simulate the behavior of the external memory (DDR3) using the testbench. This was done by first creating files using a Python script which contains the weights and input data of the bottleneck convolution (one file for each type of data). A Python script was written to produce random input FM and weight filters. Therefore when the simulation is performed, if the DMA requests data at a particular address, the testbench reads the data at this address in the corresponding file. Since each data type is specified by its offset (see Section 3.2.2), we can easily determine based on the address which files to read.

In the same manner, when the DMA writes the output FM tile into the external memory, the testbench writes the tile into a file with the corresponding address to reconstruct the output FM. To verify if the output FM is logically correct, a Python script was developed to compute the correct output. Therefore, we can compare both results and determine the correctness of the accelerator.

As said in Section 2.1.2, all pixels and weights are represented using 16 bits. Since the DDR3 memory stores words of 32 *bits*, it is sufficient to store each type of data in using a single address.

Table 3.4 lists the parameters chosen to produce the results. Since the goal is to assess the correctness of the output of the network and measure its performance, we can choose arbitrary dimensions for input and output FMs. I chose parameters that are not too small to be able to observe distinct results and not too large for the sake of the latency of the functional simulation. However, the design made is parametrizable and those parameters can be easily modified.

|                  |    |
|------------------|----|
| $N_{ix}, N_{iy}$ | 7  |
| $N_{if}$         | 32 |
| $N_{of}$         | 64 |
| $S$              | 1  |
| $t$              | 6  |
| $T_{ox}, T_{oy}$ | 16 |

Table 3.4: Parameters of the implemented layer

### 3.3.2 Results

In this section are presented the results obtained from the functional simulation of the design using parameters from Table 3.4. First, we present the comparason between expected and obtained output FM, to check if the proposed architecture provides a logically correct output (fourth design objective). Second, we compare the performance of the network by modifying the pruning parameters to observe if an increase in the sparsity improves the performance of the architecture (fifth objective). Finally, we look at the resource usage of the network to see if the design can be implemented on the target platform.

#### Logical correctness of the design

As explained previously, the logical correctness is measured by comparing the expected output FM of the network (generated by a Python script that performs the proposed bottleneck convolution using floating-point numbers) with the obtained output FM (generated by the functional simulation of the network using the same input FM and filters as the Python script). Since fixed-point arithmetic is more subject to arithmetic error than floating-point numbers, the obtained output pixels can be less precise than the expected ones. We consider therefore that an obtained output pixel is correct if the difference between the obtained and expected output pixel is below a threshold, which was at 0.1.

We illustrate the ratio of correct output pixel depending on the number of partial sums per output pixel in Figure 3.21, where the number of partial sums per output pixel represents the number of partial products we need to add to the partial result of the depthwise convolution to obtain a final output pixel. In other words, it is equal to  $N_{np} \times N_{gr-int}$ .

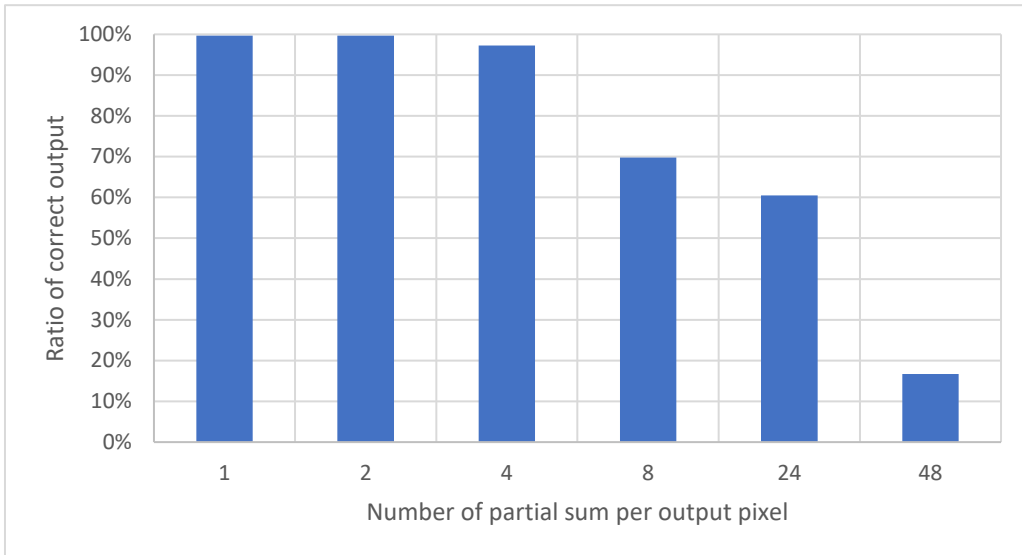


Figure 3.21: Ratio of correct output depending on the number of partial sums per output pixel

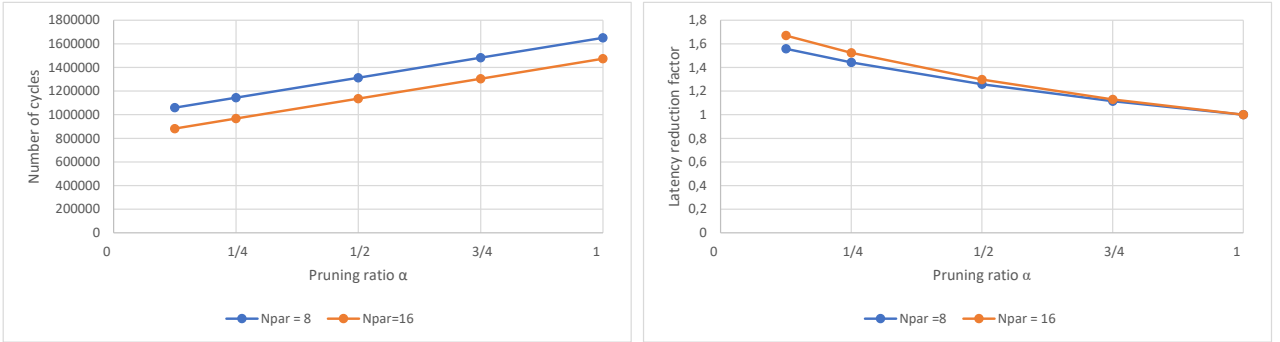
We can observe in Figure 3.21 that the ratio of correct output is almost maximal for small numbers of partial sum, but as this number increases the ratio of correct output drops rapidly. It can be explained by the accumulation of small rounding errors which leads to a drop in precision. This is verified by the fact that a bigger threshold increases the ratio of correct output.

For example, if the threshold = 2, the ratio of correct output becomes equal to 80% when the number of partial sums is equal to 48. In Figure 3.21, where threshold = 0.1, the ratio of correct output is equal to 17%.

## Measuring accelerator performance

We measured the performance of the accelerator using the functional simulation, where input FM and filters are randomly generated by a Python script. As said previously, the performance is computed by measuring the number of cycles to perform the inference of the bottleneck convolution.

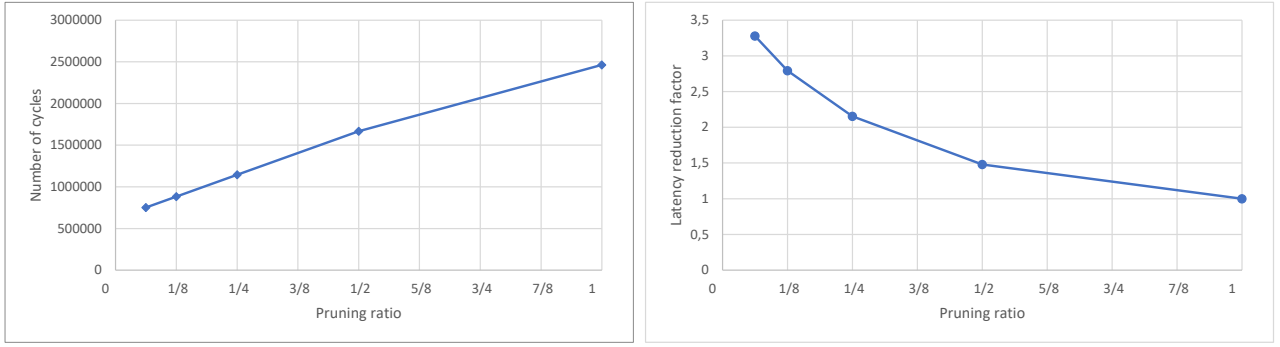
- First, for a particular  $N_{par}$ , we vary the number of non-pruned weights to check if a smaller pruning ratio means better performance. The number of cycles measured can be found in Figure 3.22a. We have also presented the latency reduction factor compared to the number of cycles when no pruning is applied ( $\alpha = 1$ ) in Figure 3.22b. We observe from those two figures that the number of cycles increases as the pruning ratio grows, for both values of  $N_{par}$ . If we compare the performance of both  $N_{par}$ , we see that a bigger  $N_{par}$  provides a smaller latency and a slightly higher latency reduction factor.



(a) Number of cycles of the design depending on the pruning ratio for a  $N_{par}$  fixed,  $N_{par} = 16$  in orange and  $N_{par} = 8$  in blue  
(b) Latency reduction factor depending on the pruning ratio for a  $N_{par}$  fixed,  $N_{par} = 16$  in orange and  $N_{par} = 8$  in blue

Figure 3.22: Number of cycles and Reduction factor of the number of cycles

- Second, we conduct similar measurements as above but instead of keeping constant the number of non-pruned weights  $N_{np}$ , we varied the fetching group size  $N_{par}$ . The number of cycles depending on the pruning ratio for an  $N_{np}$  fixed can be found in Figure 3.23a and the latency reduction factor is shown in Figure 3.23b. We can observe that as the pruning ratio increases, the latency increases as well (and hence the latency reduction factor decreases). We can also note that we can achieve a bigger latency reduction factor by varying the pruning ratio with  $N_{par}$  instead of  $N_{np}$ . For example, in Figure 3.22b, the reduction factor corresponding to  $\alpha = \frac{1}{8}$  is equal to 1.66. On the other hand, in Figure 3.22b, the reduction factor corresponding to  $\alpha = \frac{1}{8}$ , is equal to 2.79.



(a) Number of cycles of the design when  $N_{np} = 2$       (b) Latency Reduction factor when  $N_{np} = 2$

Figure 3.23: Number of cycles and Reduction factor of the number of cycles for an  $N_p$  fixed

- Finally, we compare the proportion of each component in the overall latency for various pruning parameters. The results can be seen in Figure 3.24. We observe that if we increase the sparsity, the ratio of cycles spent in the  $1 \times 1$  convolution PE is increased. The latency in the DSC can therefore be reduced by reducing the pruning ratio. We can also see that the proportion of cycles spent in the DMA is reduced by increasing the fetching group size  $N_{par}$ .

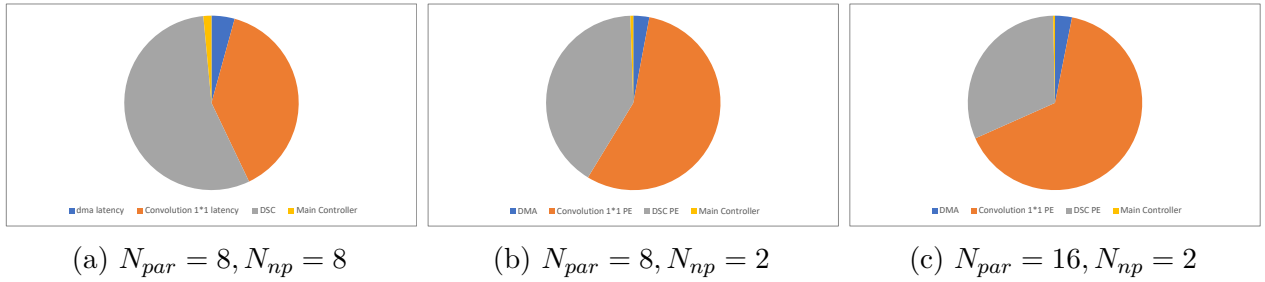


Figure 3.24: Repartition of the latency across the different components for various pruning parameters

## Resource usage

Once we have measured the output correctness and the performance of the accelerator, we can measure the resource usage on the target platform. The compilation was performed on Quartus Prime 18.1 Lite Edition. We measured the DSP utilization, the logic utilization, the number of Block RAM (BRAM), and the maximum frequency reachable. The results of the simulations are found in Table 3.5.

|                             | DSP           | Logic           | BRAM               | Frequency [MHz] |
|-----------------------------|---------------|-----------------|--------------------|-----------------|
| $N_{par} = 4, N_{np} = 2$   | 45 ( 40 % )   | 2,006 ( 5 % )   | 2,104,960 ( 37 % ) | 114.39          |
| $N_{par} = 8, N_{np} = 1$   | 81 ( 72 % )   | 2,381 ( 6 % )   | 2,067,200 ( 37 % ) | 99.57           |
| $N_{par} = 8, N_{np} = 2$   | 81 ( 72 % )   | 2,519 ( 6 % )   | 2,115,840 ( 37 % ) | 103.36          |
| $N_{par} = 8, N_{np} = 4$   | 83 ( 74 % )   | 2,678 ( 6 % )   | 2,213,120 ( 39 % ) | 85.03           |
| $N_{par} = 8, N_{np} = 6$   | 85 ( 76 % )   | 2,836 ( 7 % )   | 2,310,400 ( 41 % ) | 82.61           |
| $N_{par} = 8, N_{np} = 8$   | 87 ( 78 % )   | 2,925 ( 7 % )   | 2,407,680 ( 43 % ) | 79.85           |
| $N_{par} = 16, N_{np} = 2$  | 112 ( 100 % ) | 3,484 ( 8 % )   | 2,132,480 ( 38 % ) | 97.68           |
| $N_{par} = 16, N_{np} = 4$  | 112 ( 100 % ) | 3,831 ( 9 % )   | 2,234,880 ( 39 % ) | 82.65           |
| $N_{par} = 16, N_{np} = 8$  | 112 ( 100 % ) | 4,266 ( 10 % )  | 2,439,680 ( 43 % ) | 76.31           |
| $N_{par} = 16, N_{np} = 12$ | 112 ( 100 % ) | 4,789 ( 11 % )  | 2,644,480 ( 47 % ) | 66.01           |
| $N_{par} = 16, N_{np} = 16$ | 112 ( 100 % ) | 5,255 ( 13 % )  | 2,849,280 ( 50 % ) | 65.16           |
| $N_{par} = 32, N_{np} = 2$  | 112 ( 100 % ) | 11,595 ( 28 % ) | 2,160,640 ( 38 % ) | 75.79           |

Table 3.5: Resource utilization of the network

We can deduce, from the results of the compilation results, some relevant information:

- The compilation proves the design can be implemented on the target platform.
- The increase in  $N_{np}$  and  $N_{par}$  means bigger on-chip buffers and it is confirmed in the Table.
- An increase in the number of  $N_{np}$  decreases the operational frequency of the FPGA. This makes sense since an increase in  $N_{np}$  means an increase in the number of additions in the convolution. Therefore the critical path becomes larger and the frequency is reduced.
- An increase in the number of  $N_{par}$  increases also the number of DSP. This is explained by the depthwise PE who must performs  $N_{par} \times N_{ky} \times N_{kx}$  MAC. According to Bai et al. [39], the tree adder in Figure 3.19 is responsible for the limiting frequency. As  $N_{par}$  grows, the size of the adder tree increases and the frequency is more limited because the FPGA does not have enough DSP which are suited for such operation (when  $N_{par} = 16$ , all DSP were fully used).

### 3.3.3 Discussion

In this section, we overview the performance of the accelerator and discuss the obtained results compared to the design objectives.

The ratio of correct outputs indicates that the accelerator provides a logically correct output. Therefore, the implemented design satisfies the fourth design objective: “**The proposed architecture provides a logically correct output**”. However, the precision of the output varies according to the pruning parameters, and the accuracy of outputs decreases as  $N_{np} \times N_{gr-int}$  increases, which could be a problem if the layer requires a high number of partial sums per output pixel. Still, quantization and integer-arithmetic are known in the literature to suffer from loss of accuracy [101, 102]. Indeed, Wu et al. [101] and Jacob et al. [102] show that if the training simulates integer arithmetic (and hence the drop of accuracy), the drop of accuracy can be reduced during the training phase.

The performance measurements show that an increase in the sparsity reduces the number of cycles and hence improves the performance of the design. This is explained by two reasons. First, the reduction of  $N_{np}$  means that the DSC PE has to load fewer weights for each pointwise fetching group. As shown in Figure 3.17,  $N_{np}$  impacts only the number of parallel multiplication and addition in one cycle when performing a  $1 \times 1$  convolution. The cycle reduction is not impacted in the  $1 \times 1$  convolution PE because weights and input pixels are loaded in parallel. Second, the increase in the fetching group size  $N_{par}$  means that we have fewer fetching groups. We can conclude that  $N_{par}$  directly controls the number of convolution and therefore the number of operations. This is why the choice of  $N_{par}$  has much impact than  $N_{np}$ . A bigger  $N_{par}$  also means that operation for fetching pixels and weights from the on-chip memory can be more efficiently pipelined, meaning more efficient data accesses. The selection of the highest  $N_{par}$  possible should be prioritized over  $N_{np}$ .

We deduced from the performance measurements that we can improve the latency of the accelerator by applying a smaller pruning ratio and a high  $N_{par}$ . However, the result of the simulation suggests that the optimal value of  $N_{par}$  depends on the resource of the target FPGA. Therefore when implementing the design of the FPGA, the user should find a tradeoff between the pruning parameters, and the constrained resources of the FPGA.

# Conclusion

In the last decade, advances in the domain of CNN have led to more accurate networks. This came at the price of an increase in the number of parameters and computational complexity. To perform an efficient inference of the CNN model, the platform on which the model is executed needs to possess enough computational resources. For this reason, GPUs are the dominant platform to execute CNN thanks to their high throughput and reprogrammability. However, GPUs are not suited for embedded and mobile applications due to their high energy consumption. A better solution would be to implement CNN on FPGA which provides better energy efficiency, better performance than CPU, and better reconfigurability than ASIC. The downside of FPGAs is their constrained resource, and the challenge is then to find the mapping of large CNN models to the FPGA.

In the frame of this thesis, we looked at different solutions to reduce the size of the model. Two promising optimizations to reduce the size of the CNN retained our interest: pruning and model optimizations, in particular the use of depthwise separable convolution. The goal of this work was then to combine both approaches by designing an efficient FPGA-based accelerator architecture integrating sparse DSC in a CNN targeting the embedded space.

Before starting the design phase of a structured pruned scheme of the DSC and the FPGA-based accelerator, we defined several design objectives that matches with the purpose of using pruning:

1. The pruning scheme was as fine-grained as possible.
2. The pruning scheme reduced computational complexity.
3. The pruning scheme allowed a reduction of memory required to store the weights.
4. The proposed architecture provided a logically correct output.
5. An increase in the sparsity improved the performance of the architecture.

First, we designed a pruning scheme that prevented the irregular data access pattern while imposing the least constraints, satisfying the first design objective. As this pruning scheme is finer-grained than kernel and channel-wise pruning, it would not be efficiently implementable on CPU and GPU. The analysis of the reduction factors of the pruning scheme allowed us to assess the reduction of the size of the model and its computational complexity.

Second, we analyzed the sparsity created by the pruning scheme to propose a compressed format reducing memory utilization depending on the ratio of sparsity. The results showed that, for the quantization chosen (16 bits), we needed to prune at least 40% of the weights to have a reduction of memory required, and then satisfying the third design objective.

Third, we developed an FPGA-based accelerator that handled the pruning scheme. In the light of the results achieved, the architecture was logically correct and its performance was not degraded with a higher sparsity, satisfying the fourth and fifth design objectives.

This showed that the proposed five design objectives were met. At the end of this thesis, we can conclude that we have proposed a CNN optimization (DSC and pruning) decreasing the number of parameters and computational complexity, that can be implemented more efficiently on FPGA, compared to GPU and CPU. This reduction of weights and computational complexity lead to an acceleration of the inference phase, which was the main objective of this thesis. As a conclusion, we have accelerated convolutional neural networks on FPGA using depthwise separable convolution and pruning.

## Future Works

Multiple paths can be explored to extend this work.

First, since we only designed an accelerator of the Bottleneck residual block of MobileNetV2, we can extend its architecture to support all types of layers of the model and compare it with other implementations to see the real impact of the pruning scheme. Also, several improvements can be applied to the proposed design to accelerate the inference further:

- We can increase the temporal parallelization (pipelining) of the design by using the ping-pong weight buffer presented in Section 2.2.2. It can be applied when fetching weights from the external memory, but we can also use this structure to pipeline both  $1 \times 1$  and depthwise separable convolutions. Indeed, while an intermediate fetching group is fed to the DSC, the next  $1 \times 1$  convolution can be computed.
- In the proposed design, each buffer has one read and write port. We can implement a multi-port buffer by stacking block RAM, each one providing one port. It would allow us to reduce the latency of fetching data from the on-chip memory.
- The algorithmic optimizations from Section 2.1.1 could be used to accelerate the depthwise convolutions.
- An analysis using the roofline model presented in Section 2.2 could be used to determine the optimal hardware design parameters for the target platform.

Second, we need to study the impact of the proposed pruning scheme and compare it with channel pruning to determine if there is a significant accuracy gap between the two approaches. Furthermore, we can investigate learning algorithms that conduct the pruning scheme of the  $1 \times 1$  kernels. As mentioned in Section 3.3.3, we also need to consider in the learning phase the fact that we used fixed-point arithmetic instead of floating-point arithmetic, to lower the loss of accuracy driven by the integer algorithms.

Finally, based on the pruning scheme, the pruning scheme could be extended to reduce further the number of weights and operations. Indeed, to keep all fetching groups at the same size, we did not consider the fact that a channel can be pruned in each kernel and that the depthwise and  $1 \times 1$  kernels corresponding to that channel can be safely removed. Therefore, a possible extent would be to study if added pruning can be efficiently handled on FPGA.

# Appendix A

## MobileNetV2

The structure of MobileNetV2 is illustrated in Figure A.1. According to Sandler et al. [22], the structure can be described as follow: “Each line describes a sequence of 1 or more identical (modulo stride) layers, repeated  $n$  times. All layers in the same sequence have the same number  $c$  of output channels. The first layer of each sequence has a stride  $s$  and all others use stride 1. All spatial convolutions use  $3 \times 3$  kernels. The expansion factor  $t$  is always applied to the input size”.

| Input                    | Operator    | $t$ | $c$  | $n$ | $s$ |
|--------------------------|-------------|-----|------|-----|-----|
| $224^2 \times 3$         | conv2d      | -   | 32   | 1   | 2   |
| $112^2 \times 32$        | bottleneck  | 1   | 16   | 1   | 1   |
| $112^2 \times 16$        | bottleneck  | 6   | 24   | 2   | 2   |
| $56^2 \times 24$         | bottleneck  | 6   | 32   | 3   | 2   |
| $28^2 \times 32$         | bottleneck  | 6   | 64   | 4   | 2   |
| $14^2 \times 64$         | bottleneck  | 6   | 96   | 3   | 1   |
| $14^2 \times 96$         | bottleneck  | 6   | 160  | 3   | 2   |
| $7^2 \times 160$         | bottleneck  | 6   | 320  | 1   | 1   |
| $7^2 \times 320$         | conv2d 1x1  | -   | 1280 | 1   | 1   |
| $7^2 \times 1280$        | avgpool 7x7 | -   | -    | 1   | -   |
| $1 \times 1 \times 1280$ | conv2d 1x1  | -   | k    | -   | -   |

Figure A.1: MobileNetV2 structure, from [22]

# Appendix B

## Weight and operation reduction factors

This appendix describes the development to find the expression of the weight and operation reduction factors when applying the proposed pruning scheme on DSC, inspired by Bai et al. [39] and Liu et al. [12].

The amount of weights  $W_{DSC}$  and operations  $O_{DSC}$  of the DSC can be determined as follow:

$$\begin{aligned} W_{DSC} &= N_{kx} \times N_{ky} \times N_{if} + N_{if} \times N_{of} \\ O_{DSC} &= N_{ix} \times N_{iy} \times N_{kx} \times N_{ky} \times N_{if} + N_{ix} \times N_{iy} \times N_{if} \times N_{of} \end{aligned}$$

The same way, the amount of weights  $W_{PR\_DSC}$  and operations  $O_{PR\_DSC}$  of the DSC with pruning can be determined as follow:

$$\begin{aligned} W_{PR\_DSC} &= N_{kx} \times N_{ky} \times N_{if} + N_{np} \times N_{gr} \times N_{of} \\ O_{PR\_DSC} &= N_{ix} \times N_{iy} \times N_{kx} \times N_{ky} \times N_{if} + N_{ix} \times N_{iy} \times N_{gr} \times N_{np} \times N_{of} \end{aligned}$$

The weight reduction factor can be expressed as the ratio between the number of weights in the DSC and the pruned number of weights:

$$\begin{aligned} F_{Wg} &= \frac{W_{DSC}}{W_{PR\_DSC}} \\ &= \frac{N_{kx} \times N_{ky} \times N_{if} + N_{if} \times N_{of}}{N_{kx} \times N_{ky} \times N_{if} + N_{np} \times N_{gr} \times N_{of}} \\ &= \frac{N_{kx} \times N_{ky} \times N_{if} + N_{if} \times N_{of}}{N_{kx} \times N_{ky} \times N_{if} + N_{np} \times \frac{N_{if}}{N_{par}} \times N_{of}} \\ &= \frac{N_{kx} \times N_{ky} \times N_{if} + N_{if} \times N_{of}}{N_{kx} \times N_{ky} \times N_{if} + \alpha \times N_{if} \times N_{of}} \\ &= \frac{N_{kx} \times N_{ky} + N_{of}}{N_{kx} \times N_{ky} + \alpha \times N_{of}} \\ &= \frac{1 + \frac{N_{kx} \times N_{ky}}{N_{of}}}{\alpha + \frac{N_{kx} \times N_{ky}}{N_{of}}} \end{aligned}$$

We can use the same method to find the operation reduction factor.

$$\begin{aligned}
F_{Op} &= \frac{O_{DSC}}{O_{PR\_DSC}} \\
&= \frac{N_{ix} \times N_{iy} \times N_{kx} \times N_{ky} \times N_{if} + N_{ix} \times N_{iy} \times N_{if} \times N_{of}}{N_{ix} \times N_{iy} \times N_{kx} \times N_{ky} \times N_{if} + N_{ix} \times N_{iy} \times N_{gr} \times N_{np} \times N_{of}} \\
&= \frac{N_{kx} \times N_{ky} \times N_{if} + N_{if} \times N_{of}}{N_{kx} \times N_{ky} \times N_{if} + N_{gr} \times N_{np} \times N_{of}} \\
&= \frac{1 + \frac{N_{kx} \times N_{ky}}{N_{of}}}{\alpha + \frac{N_{kx} \times N_{ky}}{N_{of}}}
\end{aligned}$$

# Appendix C

## Minimal pruning ratio required to reduce memory utilization

To reduce the memory utilization, the number of bits required to represent a sparse kernel in the proposed compressed format must be less than if we store this kernel without compressed format. We can, therefore, express this requirement as in Equation (C.1), where  $BW_{weight}$  is the bitwidth of a weight.

$$N_{np} \times N_{gr} \times (BW_{weight} + \log_2(N_{par})) < N_{if} \times BW_{weight} \quad (C.1)$$

Equation (C.1) can be developed to express the maximal pruning ratio  $\alpha$  in order to have a reduction of the memory usage:

$$\begin{aligned} N_{np} \times N_{gr} \times (BW_{weight} + \log_2(N_{par})) &< N_{if} \times BW_{weight} \\ N_{np} \times \frac{N_{if}}{N_{par}} \times (BW_{weight} + \log_2(N_{par})) &< N_{if} \times BW_{weight} \\ \frac{N_{np}}{N_{par}} \times (BW_{weight} + \log_2(N_{par})) &< BW_{weight} \\ \alpha &< \frac{BW_{weight}}{(BW_{weight} + \log_2(N_{par}))} \end{aligned}$$

As a result, to save memory utilization, given a  $N_{par}$ , we need a pruning ratio that satisfies the relation in Equation C.2.

$$\alpha < \frac{BW_{weight}}{(BW_{weight} + \log_2(N_{par}))} \quad (C.2)$$

# Appendix D

## Details of the different signals of each component of the architecture

We detail here the different input and output signals of each component of the overall architecture described in Section 3.2. Each Table details the signals of one component.

| Signals                                                                                    | Input/Output | Description                                                                                                                                              |
|--------------------------------------------------------------------------------------------|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| clk, rst                                                                                   | Input        | Clock and Reset signals                                                                                                                                  |
| inf_conv                                                                                   | Input        | convolution information of the layer                                                                                                                     |
| start                                                                                      | Input        | Tells the main controller that it can perform the bottleneck convolution                                                                                 |
| $s_{dma}$<br>$s_{c11}$<br>$s_{dsc}$                                                        | Output       | Starting signals telling that the corresponding component can start its operation                                                                        |
| finish                                                                                     | Output       | Enabled by the main controller indicating the output FM has been produced                                                                                |
| $f_{dma}$<br>$f_{c11}$<br>$f_{dsc}$                                                        | Input        | Finishing signals telling that the corresponding component has accomplished its operation                                                                |
| $dma_{op}$<br>$dma_{info_1}$<br>$dma_{info-mem_1}$<br>$dma_{info_2}$<br>$dma_{info-mem_2}$ | Output       | Since the DMA can handle multiple operations, theses signals specify which operations to perform and which tile and its external memory address to fetch |
| first_par                                                                                  | Output       | Control signal telling the DSC PE that loads its first fetching group<br>Therefore, it does not read the data in the output FM buffer                    |

Table D.1: Input and output signals of the main controller

| Signals                                                                     | Input/Output | Description                                                                                                            |
|-----------------------------------------------------------------------------|--------------|------------------------------------------------------------------------------------------------------------------------|
| clk, rst                                                                    | Input        | Clock and Reset signals                                                                                                |
| inf_conv                                                                    | Output       | Convolution information of the layer<br>fetched from the external memory                                               |
| $s_{op}$<br>$op$                                                            | Input        | Starting signals telling the DMA that it can perform<br>the operation referenced by $op$                               |
| $r_{valid\_extmem}$<br>$data_{extmem}$                                      | Input        | If enabled, the signal $data_{extmem}$ corresponds<br>to a data fetched from external memory                           |
| $r_{request\_extmem}$<br>$addr_{extmem}$                                    | Output       | If enabled, requesting a data transfer<br>at address $addr_{extmem}$                                                   |
| $e_{op}$                                                                    | Output       | Enabled by the DMA<br>indicating the transaction is done                                                               |
| $w_{fmi}$<br>$w_{kex}$<br>$w_{kpw}$<br>$w_{kdw}$<br>$w_{ext}$<br>$w_{data}$ | Output       | Write signals, enabled when<br>we write $w_{data}$ into the corresponding memory                                       |
| $addr_{ram}$                                                                | Output       | When reading FMO buffer,<br>indicates the address of the data                                                          |
| $ram_{data-i}$                                                              | Input        | Data reads at address $addr_{ram}$<br>in the FMO buffer                                                                |
| $r_{fmo-buff}$                                                              | Output       | Enabled by the DMA when it reads<br>data from the FMO buffer                                                           |
| $tx_i$<br>$x_{mem-i}$<br>$ty_i$<br>$ty_{mem-i}$                             | Input        | Theses signals specify which tile<br>and its external memory address to fetch<br>(the data type is specified by $op$ ) |

Table D.2: Input and output signals of the DMA

| Signals                                     | Input/Output | Description                                                                                          |
|---------------------------------------------|--------------|------------------------------------------------------------------------------------------------------|
| clk, rst                                    | Input        | Clock and Reset signals                                                                              |
| $N_{if}$                                    | Input        | Number of channels of the Input FM                                                                   |
| start                                       | Input        | Tells the PE that it can perform the $1 \times 1$ convolution                                        |
| finish                                      | Output       | Enabled by the PE indicating the intermediate FM tile has been produced                              |
| $fmi_{data}$<br>$kex_{data}$<br>$kex_{pos}$ | Input        | Output data signal of the corresponding buffers                                                      |
| $fmi_{addr}$<br>$kex_{addr}$                | Output       | Buffer address of the data to fetch                                                                  |
| $res$<br>$fmint_{addr}$<br>$write$          | Output       | If <i>write</i> enabled, write the signal <i>res</i> into the FMINT buffer at address $fmint_{addr}$ |
| $Size_{KEX}$                                | Input        | Size of one $1 \times 1$ convolution kernel                                                          |

Table D.3: Input and output signals of the  $1 \times 1$  convolution PE

| Signals                                                                     | Input/Output | Description                                                                                                                           |
|-----------------------------------------------------------------------------|--------------|---------------------------------------------------------------------------------------------------------------------------------------|
| clk, rst                                                                    | Input        | Clock and Reset signals                                                                                                               |
| $N_{if}$<br>$N_{of}$<br>$N_{ox}$<br>$N_{oy}$<br>$T_{ox}$<br>$T_{oy}$<br>$S$ | Input        | Layer information required by the PE                                                                                                  |
| start                                                                       | Input        | Tells the PE that it can perform the DSC convolution                                                                                  |
| finish                                                                      | Output       | Enabled by the PE indicating the output FM tile has been produced                                                                     |
| $fmint_{data}$<br>$kdw_{data}$<br>$kpw_{data}$<br>$kpw_{pos}$               | Input        | Output data signal of the corresponding buffers                                                                                       |
| $fmint_{addr}$<br>$kdw_{addr}$<br>$kpw_{addr}$                              | Output       | Buffer address of the data to fetch                                                                                                   |
| $res$<br>$fmo_{addr}$<br>$write$                                            | Output       | If <i>write</i> enabled, write the signal <i>res</i> into the FMINT buffer at address $fmint_{addr}$                                  |
| $Size_{KEX}$                                                                | Input        | Size of one $1 \times 1$ convolution kernel                                                                                           |
| first_par_i                                                                 | Output       | Control signal telling the DSC PE that loads its first fetching group<br>Therefore, it does not read the data in the output FM buffer |

Table D.4: Input and output signals of the DSC convolution PE

# Bibliography

- [1] Peter Norvig Stuart Russell. *Artificial Intelligence: A Modern Approach*. English. 3rd ed. United States: Prentice Hall Press, Dec. 2009. ISBN: 0-13-604259-7. URL: <https://dl.acm.org/doi/book/10.5555/1671238>.
- [2] Md Zahangir Alom et al. “The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches”. In: *CoRR* abs/1803.01164 (Sept. 2018). arXiv: 1803.01164. URL: <http://arxiv.org/abs/1803.01164> (visited on 05/13/2020).
- [3] Tom M. Mitchell. *Machine Learning*. en. McGraw-Hill series in computer science. New York: McGraw-Hill, 1997. ISBN: 978-0-07-042807-2.
- [4] Ludovic Arnold et al. “An Introduction to Deep Learning”. en. In: *Proceedings of the European Symposium on Artificial Neural Networks (ESANN)*. Vol. 1. Bruges, Belgium, 2011, pp. 477–488. URL: <https://hal.archives-ouvertes.fr/hal-01352061> (visited on 06/12/2020).
- [5] Ritika Wason. “Deep learning: Evolution and expansion”. en. In: *Cognitive Systems Research* 52 (Dec. 2018), pp. 701–708. ISSN: 13890417. DOI: 10.1016/j.cogsys.2018.08.023. URL: <https://linkinghub.elsevier.com/retrieve/pii/S1389041717303546> (visited on 05/13/2020).
- [6] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. en. In: *International Journal of Computer Vision* 115.3 (Dec. 2015), pp. 211–252. ISSN: 0920-5691, 1573-1405. DOI: 10.1007/s11263-015-0816-y. URL: <http://link.springer.com/10.1007/s11263-015-0816-y> (visited on 05/13/2020).
- [7] *ImageNet Large Scale Visual Recognition Competition 2012 (ILSVRC2012)*. English. URL: <http://image-net.org/challenges/LSVRC/2012/results.html#t1> (visited on 08/12/2020).
- [8] Ahmad Shawahna, Sadiq M. Sait, and Aiman El-Maleh. “FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review”. In: *IEEE Access* 7 (2019). Conference Name: IEEE Access, pp. 7823–7859. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2890150.
- [9] Christian Szegedy et al. “Going deeper with convolutions”. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. ISSN: 1063-6919. June 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594. URL: <https://ieeexplore.ieee.org/document/7298594>.
- [10] Asifullah Khan et al. “A Survey of the Recent Architectures of Deep Convolutional Neural Networks”. en. In: *Artificial Intelligence Review* (Apr. 2020). arXiv: 1901.06032. ISSN: 0269-2821, 1573-7462. DOI: 10.1007/s10462-020-09825-6. URL: <http://arxiv.org/abs/1901.06032> (visited on 06/06/2020).
- [11] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. ISSN: 1063-6919. June 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

- [12] Bing Liu et al. “An FPGA-Based CNN Accelerator Integrating Depthwise Separable Convolution”. en. In: *Electronics* 8.3 (Mar. 2019). Number: 3 Publisher: Multidisciplinary Digital Publishing Institute, p. 281. DOI: 10.3390/electronics8030281. URL: <https://www.mdpi.com/2079-9292/8/3/281> (visited on 06/18/2020).
- [13] Zhiqiang Liu et al. “A Uniform Architecture Design for Accelerating 2D and 3D CNNs on FPGAs”. en. In: *Electronics* 8.1 (Jan. 2019). Number: 1 Publisher: Multidisciplinary Digital Publishing Institute, p. 65. DOI: 10.3390/electronics8010065. URL: <https://www.mdpi.com/2079-9292/8/1/65> (visited on 05/13/2020).
- [14] Murad Qasaimeh et al. “Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels”. In: *2019 IEEE International Conference on Embedded Software and Systems (ICCESS)*. June 2019, pp. 1–8. DOI: 10.1109/ICCESS.2019.8782524.
- [15] Mohammad Motamedi, Philipp Gysel, and Soheil Ghiasi. “PLACID: A Platform for FPGA-Based Accelerator Creation for DCNNs”. In: *ACM Transactions on Multimedia Computing, Communications, and Applications* 13.4 (Sept. 2017), 62:1–62:21. ISSN: 1551-6857. DOI: 10.1145/3131289. URL: <https://doi.org/10.1145/3131289> (visited on 06/19/2020).
- [16] Eriko Nurvitadhi et al. “Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?” en. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*. Monterey, California, USA: ACM Press, 2017, pp. 5–14. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021740. URL: <http://dl.acm.org/citation.cfm?doid=3020078.3021740> (visited on 06/19/2020).
- [17] Raghid Morcel et al. “FeatherNet: An Accelerated Convolutional Neural Network Design for Resource-constrained FPGAs”. In: *ACM Transactions on Reconfigurable Technology and Systems* 12.2 (Mar. 2019), 6:1–6:27. ISSN: 1936-7406. DOI: 10.1145/3306202. URL: <https://doi.org/10.1145/3306202> (visited on 08/12/2020).
- [18] Kamel Abdelouahab et al. “Accelerating CNN inference on FPGAs: A Survey”. English. France, Mar. 2018. URL: <https://hal.archives-ouvertes.fr/hal-01695375> (visited on 06/12/2020).
- [19] Y. LeCun et al. “Backpropagation Applied to Handwritten Zip Code Recognition”. In: *Neural Computation* 1.4 (Dec. 1989). Conference Name: Neural Computation, pp. 541–551. ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.4.541.
- [20] Chen Zhang et al. “Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks”. en. In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '15*. Monterey, California, USA: ACM Press, 2015, pp. 161–170. ISBN: 978-1-4503-3315-3. DOI: 10.1145/2684746.2689060. URL: <http://dl.acm.org/citation.cfm?doid=2684746.2689060> (visited on 06/18/2020).
- [21] Jian Cheng et al. “Recent advances in efficient computation of deep convolutional neural networks”. en. In: *Frontiers of Information Technology & Electronic Engineering* 19.1 (Jan. 2018), pp. 64–77. ISSN: 2095-9230. DOI: 10.1631/FITEE.1700789. URL: <https://doi.org/10.1631/FITEE.1700789> (visited on 06/13/2020).
- [22] Mark Sandler et al. “MobileNetV2: Inverted Residuals and Linear Bottlenecks”. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. ISSN: 2575-7075. June 2018, pp. 4510–4520. DOI: 10.1109/CVPR.2018.00474. (Visited on 06/14/2020).
- [23] Frank F. Rosenblatt and F. Rosenblatt. “The Perceptron: A Probabilistic Model for Information Storage and Organization”. English. In: *Psychological review* 65.6 (1958), pp. 386–408. URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.335.3398&rep=rep1&type=pdf> (visited on 06/15/2020).

- [24] Matteo Matteucci and Giacomo Boracchi. “Artificial Neural Networks and Deep Learning”. English. Lecture notes. Politecnico di Milano, 2019. (Visited on 07/10/2020).
- [25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf> (visited on 06/28/2020).
- [26] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. en. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. ISSN: 1938-7228 Section: Machine Learning. Mar. 2010, pp. 249–256. URL: <http://proceedings.mlr.press/v9/glorot10a.html> (visited on 06/15/2020).
- [27] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. English. Adaptive Computation and Machine Learning serie. MIT Press, Nov. 2016. ISBN: 978-0-262-03561-3. URL: <http://www.deeplearningbook.org> (visited on 06/28/2020).
- [28] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. “Rectifier Nonlinearities Improve Neural Network Acoustic Models”. en. In: *ICML Workshop on Deep Learning for Audio, Speech and Language Processing 30* (June 2013), p. 6. URL: [https://awnihannun.com/papers/relu\\_hybrid\\_icml2013\\_final.pdf](https://awnihannun.com/papers/relu_hybrid_icml2013_final.pdf) (visited on 07/02/2020).
- [29] Alex Krizhevsky. “Convolutional Deep Belief Networks on CIFAR-10”. English. Unpublished manuscript. 2010. URL: <https://www.cs.toronto.edu/~kriz/conv-cifar10-aug2010.pdf> (visited on 07/16/2020).
- [30] Andrew G. Howard et al. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”. In: *arXiv:1704.04861 [cs]* (Apr. 2017). arXiv: 1704.04861. URL: <http://arxiv.org/abs/1704.04861> (visited on 06/17/2020).
- [31] Jean Pierre David, Kassem Kalach, and Nicolas Tittley. “Hardware Complexity of Modular Multiplication and Exponentiation”. In: *IEEE Transactions on Computers* 56.10 (Oct. 2007). Conference Name: IEEE Transactions on Computers, pp. 1308–1319. ISSN: 1557-9956. DOI: 10.1109/TC.2007.1084.
- [32] Marvin Lee Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. English. Cambridge, MA, USA: Mit Press, 1969. ISBN: 978-0-262-63022-1. URL: <https://mitpress.mit.edu/books/perceptrons> (visited on 07/16/2020).
- [33] Ruizhe Zhao et al. “Towards Efficient Convolutional Neural Network for Domain-Specific Applications on FPGA”. English. In: arXiv: 1809.03318. Dublin, Ireland: IEEE, Sept. 2018. DOI: 10.1109/FPL.2018.00033. URL: <http://hdl.handle.net/10044/1/62190> (visited on 07/06/2020).
- [34] Yufei Ma et al. “Optimizing the Convolution Operation to Accelerate Deep Neural Networks on FPGA”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.7 (July 2018). Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, pp. 1354–1367. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2018.2815603.
- [35] Chaoyang Zhu et al. “An Efficient Hardware Accelerator for Structured Sparse Convolutional Neural Networks on FPGAs”. English. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (Jan. 2020), pp. 1–13. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2020.3002779. URL: <https://www.computer.org/csdl/journal/si/5555/01/09130762/116013eFFok> (visited on 06/14/2020).

- [36] Kaiyuan Guo et al. “A Survey of FPGA-Based Neural Network Accelerator”. In: *arXiv:1712.08934 [cs]* (Dec. 2018). arXiv: 1712.08934. URL: <http://arxiv.org/abs/1712.08934> (visited on 06/18/2020).
- [37] Laurent Sifre and Prof Stéphane Mallat. “Rigid-Motion Scattering For Image Classification Author:” English. Supervisor: Prof. Stéphane Mallat. Ph.D. Thesis. Ecole Polytechnique, 2014. URL: [https://www.di.ens.fr/data/publications/papers/phd\\_sifre.pdf](https://www.di.ens.fr/data/publications/papers/phd_sifre.pdf) (visited on 06/27/2020).
- [38] François Chollet. “Xception: Deep Learning with Depthwise Separable Convolutions”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. ISSN: 1063-6919. July 2017, pp. 1800–1807. DOI: 10.1109/CVPR.2017.195.
- [39] Lin Bai, Yiming Zhao, and Xinming Huang. “A CNN Accelerator on FPGA Using Depthwise Separable Convolution”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 65.10 (Oct. 2018). Conference Name: IEEE Transactions on Circuits and Systems II: Express Briefs, pp. 1415–1419. ISSN: 1558-3791. DOI: 10.1109/TCSII.2018.2865896.
- [40] Ke Zhang et al. “A Channel Pruning Algorithm Based on Depth-Wise Separable Convolution Unit”. In: *IEEE Access* 7 (2019). Conference Name: IEEE Access, pp. 173294–173309. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2956976.
- [41] Naveen Suda et al. “Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks”. en. In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '16*. Monterey, California, USA: ACM Press, 2016, pp. 16–25. ISBN: 978-1-4503-3856-1. DOI: 10.1145/2847263.2847276. URL: <http://dl.acm.org/citation.cfm?doid=2847263.2847276> (visited on 06/28/2020).
- [42] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *arXiv:1409.1556 [cs]* (Apr. 2015). arXiv: 1409.1556. URL: <http://arxiv.org/abs/1409.1556> (visited on 06/30/2020).
- [43] Kaiming He et al. “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”. In: *2015 IEEE International Conference on Computer Vision (ICCV)*. ISSN: 2380-7504. Dec. 2015, pp. 1026–1034. DOI: 10.1109/ICCV.2015.123.
- [44] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *arXiv:1609.04747 [cs]* (June 2017). arXiv: 1609.04747. URL: <http://arxiv.org/abs/1609.04747> (visited on 06/29/2020).
- [45] Sparsh Mittal and Jeffrey S. Vetter. “A Survey of Methods for Analyzing and Improving GPU Energy Efficiency”. English. In: *ACM Computing Surveys* 47.2 (Jan. 2015), pp. 1–23. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/2636342. URL: <https://dl.acm.org/doi/10.1145/2636342> (visited on 07/11/2020).
- [46] Mário P. Véstias et al. “Fast Convolutional Neural Networks in Low Density FPGAs Using Zero-Skipping and Weight Pruning”. en. In: *Electronics* 8.11 (Nov. 2019). Number: 11 Publisher: Multidisciplinary Digital Publishing Institute, p. 1321. DOI: 10.3390/electronics8111321. URL: <https://www.mdpi.com/2079-9292/8/11/1321> (visited on 07/11/2020).
- [47] Sarah Harris and David Harris. *Digital Design and Computer Architecture: ARM Edition*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2015. ISBN: 978-0-12-800056-4.

- [48] G.B. Barte. *Value-added Learning Platform for Undergraduate Electrical and Electronics Engineering Education*. 2011. URL: <https://books.google.be/books?id=yKeRAQAACAAJ>.
- [49] Vivienne Sze et al. “Efficient Processing of Deep Neural Networks: A Tutorial and Survey”. In: *Proceedings of the IEEE* 105.12 (Dec. 2017). Conference Name: Proceedings of the IEEE, pp. 2295–2329. ISSN: 1558-2256. DOI: 10.1109/JPROC.2017.2761740.
- [50] Yun Liang et al. “Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.4 (Apr. 2020). Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pp. 857–870. ISSN: 1937-4151. DOI: 10.1109/TCAD.2019.2897701.
- [51] Shmuel Winograd. *Arithmetic Complexity of Computations*. English. CBMS-NSF regional conference series in applied mathematics 33. <https://doi.org/10.1137/1.9781611970364>. Philadelphia: Society for Industrial and Applied Mathematics, 1980. ISBN: 978-0-89871-163-9 0-89871-163-0. URL: <http://epubs.siam.org/doi/book/10.1137/1.9781611970364> (visited on 06/14/2020).
- [52] Andrew Lavin and Scott Gray. “Fast Algorithms for Convolutional Neural Networks”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. ISSN: 1063-6919. June 2016, pp. 4013–4021. DOI: 10.1109/CVPR.2016.435. (Visited on 06/14/2020).
- [53] James W. Cooley and John W. Tukey. “An algorithm for the machine calculation of complex Fourier series”. en. In: *Mathematics of Computation* 19.90 (1965), pp. 297–301. ISSN: 0025-5718, 1088-6842. DOI: 10.1090/S0025-5718-1965-0178586-1. URL: <https://www.ams.org/mcom/1965-19-090/S0025-5718-1965-0178586-1/> (visited on 06/21/2020).
- [54] Steven W. Smith. *The Scientist and Engineer’s Guide to Digital Signal Processing’s Table of Content*. 1st ed. California Technical Pub, 1997. ISBN: 0-9660176-3-3. URL: <https://www.dspsguide.com/pdfbook.htm> (visited on 06/14/2020).
- [55] Afzal Ahmad and Muhammad Adeel Pasha. “Towards Design Space Exploration and Optimization of Fast Algorithms for Convolutional Neural Networks (CNNs) on FPGAs”. In: *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. ISSN: 1558-1101. Mar. 2019, pp. 1106–1111. DOI: 10.23919/DATE.2019.8715272.
- [56] Kamran Chitsaz et al. “Acceleration of Convolutional Neural Network Using FFT-Based Split Convolutions”. In: *arXiv:2003.12621 [cs]* (Apr. 2020). arXiv: 2003.12621. URL: <http://arxiv.org/abs/2003.12621> (visited on 06/21/2020).
- [57] Utku Aydonat et al. “An OpenCL™ Deep Learning Accelerator on Arria 10”. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’17. Monterey, California, USA: Association for Computing Machinery, Feb. 2017, pp. 55–64. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021738. URL: <https://doi.org/10.1145/3020078.3021738> (visited on 06/19/2020).
- [58] Qingcheng Xiao et al. “Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs”. en. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. Austin TX USA: ACM, June 2017, pp. 1–6. ISBN: 978-1-4503-4927-7. DOI: 10.1145/3061639.3062244. URL: <https://dl.acm.org/doi/10.1145/3061639.3062244> (visited on 06/20/2020).

- [59] Hanqing Zeng, Ren Chen, and Viktor K Prasanna. “Optimizing Frequency Domain Implementation of CNNs on FPGAs”. en. In: (2017), p. 12. URL: <http://ceng.usc.edu/techreports/2017/Prasanna%20CENG-2017-3.pdf> (visited on 06/21/2020).
- [60] Abhinav Podili, Chi Zhang, and Viktor Prasanna. “Fast and efficient implementation of Convolutional Neural Networks on FPGA”. In: *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. ISSN: 2160-052X. July 2017, pp. 11–18. DOI: 10.1109/ASAP.2017.7995253.
- [61] Chen Zhang et al. “Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks”. en. In: *Proceedings of the 35th International Conference on Computer-Aided Design*. Austin Texas: ACM, Nov. 2016, pp. 1–8. ISBN: 978-1-4503-4466-1. DOI: 10.1145/2966986.2967011. URL: <https://dl.acm.org/doi/10.1145/2966986.2967011> (visited on 07/09/2020).
- [62] Forrest N. Iandola et al. “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size”. In: *arXiv:1602.07360 [cs]* (Nov. 2016). arXiv: 1602.07360. URL: <http://arxiv.org/abs/1602.07360> (visited on 06/17/2020).
- [63] Song Han, Huizi Mao, and William J. Dally. “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding”. In: *arXiv:1510.00149 [cs]* (Feb. 2016). arXiv: 1510.00149. URL: <http://arxiv.org/abs/1510.00149> (visited on 06/17/2020).
- [64] Xiangyu Zhang et al. “ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices”. en. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. Salt Lake City, UT: IEEE, June 2018, pp. 6848–6856. ISBN: 978-1-5386-6420-9. DOI: 10.1109/CVPR.2018.00716. URL: <https://ieeexplore.ieee.org/document/8578814/> (visited on 06/17/2020).
- [65] Ningning Ma et al. “ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design”. en. In: *Computer Vision – ECCV 2018*. Ed. by Vittorio Ferrari et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 122–138. ISBN: 978-3-030-01264-9. DOI: 10.1007/978-3-030-01264-9\_8. (Visited on 06/22/2020).
- [66] Barret Zoph et al. “Learning Transferable Architectures for Scalable Image Recognition”. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. ISSN: 2575-7075. June 2018, pp. 8697–8710. DOI: 10.1109/CVPR.2018.00907.
- [67] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. “An Analysis of Deep Neural Network Models for Practical Applications”. en. In: *arXiv:1605.07678 [cs]* (Apr. 2017). arXiv: 1605.07678. URL: <http://arxiv.org/abs/1605.07678> (visited on 07/08/2020).
- [68] Zhuang Liu et al. “Rethinking the Value of Network Pruning”. In: *arXiv:1810.05270 [cs, stat]* (Mar. 2019). arXiv: 1810.05270. URL: <http://arxiv.org/abs/1810.05270> (visited on 06/24/2020).
- [69] Emily Denton et al. “Exploiting linear structure within convolutional networks for efficient evaluation”. In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*. NIPS’14. Cambridge, MA, USA: MIT Press, Dec. 2014, pp. 1269–1277. (Visited on 08/13/2020).
- [70] Baoyuan Liu et al. “Sparse Convolutional Neural Networks”. en. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Boston, MA, USA: IEEE, June 2015, pp. 806–814. ISBN: 978-1-4673-6964-0. DOI: 10.1109/CVPR.2015.7298681. URL: <http://ieeexplore.ieee.org/document/7298681/> (visited on 06/26/2020).

- [71] Jonathan Frankle and Michael Carbin. “The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks”. In: Sept. 2018. URL: <https://openreview.net/forum?id=rJl-b3RcF7> (visited on 06/24/2020).
- [72] Jonathan Frankle, David J. Schwab, and Ari S. Morcos. “The Early Phase of Neural Network Training”. In: *arXiv:2002.10365 [cs, stat]* (Feb. 2020). arXiv: 2002.10365. URL: <http://arxiv.org/abs/2002.10365> (visited on 06/24/2020).
- [73] Song Han et al. “Learning both weights and connections for efficient neural networks”. In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*. NIPS’15. Cambridge, MA, USA: MIT Press, Dec. 2015, pp. 1135–1143. (Visited on 08/13/2020).
- [74] Huizi Mao et al. “Exploring the Granularity of Sparsity in Convolutional Neural Networks”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. ISSN: 2160-7516. July 2017, pp. 1927–1934. DOI: 10.1109/CVPRW.2017.241. (Visited on 06/24/2020).
- [75] Hyeong-Ju Kang. “Accelerator-Aware Pruning for Convolutional Neural Networks”. In: *IEEE Transactions on Circuits and Systems for Video Technology* (2020). arXiv: 1804.09862, pp. 1–1. ISSN: 1051-8215, 1558-2205. DOI: 10.1109/TCSVT.2019.2911674. URL: <http://arxiv.org/abs/1804.09862> (visited on 06/24/2020).
- [76] Masayuki Shimoda, Youki Sada, and Hiroki Nakahara. “Filter-Wise Pruning Approach to FPGA Implementation of Fully Convolutional Network for Semantic Segmentation”. en. In: *Applied Reconfigurable Computing*. Ed. by Christian Hochberger et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 371–386. ISBN: 978-3-030-17227-5. DOI: 10.1007/978-3-030-17227-5\_26.
- [77] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. “Structured Pruning of Deep Convolutional Neural Networks”. en. In: *ACM Journal on Emerging Technologies in Computing Systems* 13.3 (May 2017), pp. 1–18. ISSN: 1550-4832, 1550-4840. DOI: 10.1145/3005348. URL: <https://dl.acm.org/doi/10.1145/3005348> (visited on 07/11/2020).
- [78] Wei Wen et al. “Learning Structured Sparsity in Deep Neural Networks”. In: *Advances in Neural Information Processing Systems 29*. Ed. by D. D. Lee et al. Curran Associates, Inc., 2016, pp. 2074–2082. URL: <http://papers.nips.cc/paper/6504-learning-structured-sparsity-in-deep-neural-networks.pdf> (visited on 06/26/2020).
- [79] Cheng-Hao Tu et al. “Pruning Depthwise Separable Convolutions for Extra Efficiency Gain of Lightweight Models”. In: (Sept. 2019). URL: <https://openreview.net/forum?id=B1g5qyHYPS> (visited on 06/24/2020).
- [80] Darren Ward, Mike Brookes, and Peter Y. K. Cheung. “Real-time Digital Signal Processing with the TMS320C6000”. English. Lecture notes. Imperial College of science, technology and medicine, University of London, 2001. URL: [http://www.ee.ic.ac.uk/pcheung/teaching/ee3\\_Study\\_Project/](http://www.ee.ic.ac.uk/pcheung/teaching/ee3_Study_Project/) (visited on 07/26/2020).
- [81] Jiantao Qiu et al. “Going Deeper with Embedded FPGA Platform for Convolutional Neural Network”. In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’16. Monterey, California, USA: Association for Computing Machinery, Feb. 2016, pp. 26–35. ISBN: 978-1-4503-3856-1. DOI: 10.1145/2847263.2847265. URL: <https://doi.org/10.1145/2847263.2847265> (visited on 06/23/2020).

- [82] Shouyi Yin et al. “A High Energy Efficient Reconfigurable Hybrid Neural Network Processor for Deep Learning Applications”. In: *IEEE Journal of Solid-State Circuits* 53.4 (Apr. 2018). Conference Name: IEEE Journal of Solid-State Circuits, pp. 968–982. ISSN: 1558-173X. DOI: 10.1109/JSSC.2017.2778281.
- [83] Tao Sheng et al. “A Quantization-Friendly Separable Convolution for MobileNets”. In: *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)* (Mar. 2018). arXiv: 1803.08607, pp. 14–18. DOI: 10.1109/EMC2.2018.00011. URL: <http://arxiv.org/abs/1803.08607> (visited on 06/18/2020).
- [84] Huimin Li et al. “A high performance FPGA-based accelerator for large-scale convolutional neural networks”. In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. ISSN: 1946-1488. Aug. 2016, pp. 1–9. DOI: 10.1109/FPL.2016.7577308.
- [85] Yu-Hsin Chen et al. “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks”. en. In: *IEEE Journal of Solid-State Circuits* 52.1 (Jan. 2017), pp. 127–138. ISSN: 0018-9200, 1558-173X. DOI: 10.1109/JSSC.2016.2616357. URL: <http://ieeexplore.ieee.org/document/7738524/> (visited on 06/18/2020).
- [86] Clement Farabet et al. “CNP: An FPGA-based processor for Convolutional Networks”. In: *2009 International Conference on Field Programmable Logic and Applications*. ISSN: 1946-1488. Aug. 2009, pp. 32–37. DOI: 10.1109/FPL.2009.5272559.
- [87] Vinayak Gokhale et al. “A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks”. In: *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*. ISSN: 2160-7516. June 2014, pp. 696–701. DOI: 10.1109/CVPRW.2014.106.
- [88] Victor Joos de ter Beerst and Antoine Vanderschueren. “Accelerating ternary quantized convolutional neural networks using OpenCL for FPGA”. Prom: De Vleeschouwer, Christophe ; Legat, Jean-Didier. PhD thesis. Louvain-la-Neuve: Ecole polytechnique de Louvain, Université catholique de Louvain, 2019. URL: <http://hdl.handle.net/2078.1/thesis:17752> (visited on 06/18/2020).
- [89] Sparsh Mittal. “A survey of FPGA-based accelerators for convolutional neural networks”. en. In: *Neural Computing and Applications* 32.4 (Feb. 2020), pp. 1109–1139. ISSN: 1433-3058. DOI: 10.1007/s00521-018-3761-1. URL: <https://doi.org/10.1007/s00521-018-3761-1> (visited on 06/22/2020).
- [90] Xuechao Wei et al. “Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs”. en. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. Austin TX USA: ACM, June 2017, pp. 1–6. ISBN: 978-1-4503-4927-7. DOI: 10.1145/3061639.3062207. URL: <https://dl.acm.org/doi/10.1145/3061639.3062207> (visited on 06/22/2020).
- [91] Mark Horowitz. “1.1 Computing’s energy problem (and what we can do about it)”. In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. ISSN: 2376-8606. Feb. 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.
- [92] Lin Li et al. “Low power design methodology for signal processing systems using lightweight dataflow techniques”. In: *2016 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. Oct. 2016, pp. 82–89. DOI: 10.1109/DASIP.2016.7853801.

- [93] Edward Lee and David Messerschmitt. “Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing”. en. In: *IEEE Transactions on Computers* C-36.1 (1987), pp. 24–35. URL: <https://ptolemy.berkeley.edu/publications/papers/87/staticscheduling/> (visited on 06/22/2020).
- [94] Stylianos I. Venieris and Christos-Savvas Bouganis. “Latency-driven design for FPGA-based convolutional neural networks”. In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. ISSN: 1946-1488. Sept. 2017, pp. 1–8. DOI: 10.23919/FPL.2017.8056828.
- [95] Kamel Abdelouahab et al. “Tactics to Directly Map CNN graphs on Embedded FPGAs”. In: *IEEE Embedded Systems Letters* 9.4 (2017). Publisher: Institute of Electrical and Electronics Engineers, pp. 113–116. DOI: 10.1109/LES.2017.2743247. URL: <https://hal.archives-ouvertes.fr/hal-01626462> (visited on 06/15/2020).
- [96] Samuel Williams, Andrew Waterman, and David Patterson. *Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures*. en. Tech. rep. 1407078. Sept. 2009, p. 1407078. DOI: 10.2172/1407078. URL: <http://www.osti.gov/servlets/purl/1407078/> (visited on 06/18/2020).
- [97] Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. “ZeNA: Zero-Aware Neural Network Accelerator”. In: *IEEE Design Test* 35.1 (Feb. 2018). Conference Name: IEEE Design Test, pp. 39–46. ISSN: 2168-2364. DOI: 10.1109/MDAT.2017.2741463.
- [98] Aydin Buluç et al. “Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks”. en. In: *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures - SPAA '09*. Calgary, AB, Canada: ACM Press, 2009, p. 233. ISBN: 978-1-60558-606-9. DOI: 10.1145/1583991.1584053. URL: <http://portal.acm.org/citation.cfm?doid=1583991.1584053> (visited on 08/05/2020).
- [99] *Cyclone V Device Overview*. English. May 2018. URL: [https://www.intel.com/content/dam/altera-www/global/en\\_US/pdfs/literature/hb/cyclone-v/cv\\_51001.pdf](https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-v/cv_51001.pdf) (visited on 08/11/2020).
- [100] Terasic Technologies. *Terasic - SoC Platform - Cyclone - DE10-Nano Kit*. URL: <https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=1046> (visited on 08/11/2020).
- [101] Jiaxiang Wu et al. “Quantized Convolutional Neural Networks for Mobile Devices”. In: *arXiv:1512.06473 [cs]* (May 2016). arXiv: 1512.06473. URL: <http://arxiv.org/abs/1512.06473> (visited on 08/12/2020).
- [102] Benoit Jacob et al. “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference”. In: *arXiv:1712.05877 [cs, stat]* (Dec. 2017). arXiv: 1712.05877. URL: <http://arxiv.org/abs/1712.05877> (visited on 08/12/2020).



