SoC-FPGA Deep Neural Network for Object Detection

Pedro Duarte Cotrim Oliveira Direita

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

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

Examination Committee

Chairperson: Prof. Teresa Maria Sá Ferreira Vazão Vasques
Supervisor: Prof. Horácio Cláudio De Campos Neto
Member of the Committee: Prof. Paulo Ferreira Godinho Flores

September 2021

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to thank Prof. Horácio Neto for his around the clock availability to support the development of this work with his knowledgeful insights, and for his patience and encouragement.

I would also like to thank my family for their support in life, for giving everything that I need, for believing in my capabilities and helping me achieve even my most ambitious goals.

Finally, I would like to thank my friends for sharing this journey with me, and the encouragement that they gave me.

Resumo

O objetivo deste trabalho é desenvolver um sistema embebido de deteção de objetos usando redes neuronais profundas (DNNs), numa plataforma SoC-FPGA. O sistema desenvolvido deve ser o mais eficiente possível, alcançando um bom equilíbrio entre a exatidão na deteção de objetos e os tempos de inferência.

As redes neuronais profundas desencadearam novos desenvolvimentos na deteção de objetos, alcançando melhores exatidões do que nunca, mas com altos custos computacionais. No entanto, os sistemas de deteção de objetos são amplamente aplicados em sistemas embebidos, onde os recursos disponíveis são limitados. Lidar com DNNs em sistemas embebidos é um desafio e avanços nesta área são necessários.

Foi projetado um sistema hardware/software para implementar a RetinaNet, uma DNN de deteção de objetos topo de gama, num dispositivo Xilinx Zynq-7020. Para alcançar tempos de inferência mais baixos, foi desenvolvido e integrado no sistema um processador específico para executar as camadas convolucionais da rede.

A RetinaNet foi treinada usando *quantization aware training*, uma técnica que gerou pesos de 4 bits e ativações de 8 bits com uma queda de exatidão de apenas 9 % comparando com a utilização de virgula flutuante de 32 bits. Esta quantização reduz em cerca de 78 % a memória usada por cada camada convolucional. O acelerador de convoluções em hardware desenvolvido processa a convolução 359 vezes mais rápido que um processador ARM embebido. Quando o acelerador é integrado no sistema hardware/software final, a parte implementada da RetinaNet é processada cinco vezes mais rápido que a implementação da DNN usando apenas software.

Palavras-chave: Deteção de Objectos, Redes Neuronais Convolucionais, Quantização, Co-Projecto Hardware/Software, Acelerador em Hardware, High-Level Synthesis.

Abstract

The objective of this work is to develop an object detection deep neural network (DNN) targeting SoC-FPGA based embedded systems. The developed system should be as efficient as possible achieving a good balance between object detection accuracy and inference time.

Deep neural networks unleash new developments in object detection, achieving greater accuracies than ever, but with high computational costs. Although object detection systems may be largely applied in embedded systems, dealing with the computer intensive DNNs in systems with limited resources is challenging and advancements in this area are highly required.

A hardware/software system was designed to implement RetinaNet, a top of the range object detection DNN, in a Xilinx Zynq-7020 device, a low-cost Soc-FPGA. To accelerate the DNN inference, a dedicated convolution hardware component was developed and integrated in the system.

RetinaNet was trained using quantization aware training, a technique that allowed to generate 4 bit-width weights and 8 bit-width activations with an accuracy decrease of only 9 % in comparison with RetinaNet using floating-point 32 bits. This quantization reduced in about 78 % the memory used by each convolution layer. The developed convolution hardware accelerator computes a convolution 359 times faster than the embedded ARM processor. When the accelerator is integrated into the final hardware/-software system, the implemented part of RetinaNet is processed five times faster than the software only implementation of the DNN.

Keywords: Object Detection, Convolution Neural Networks, Quantization, Hardware/Software Co-Design, Hardware Accelerator, High-Level Synthesis.

Contents

Acknowledgments . . . v

Resumo . . . vii

Abstract . . . ix

List of Tables . . . xv

List of Figures . . . xvii

List of Acronyms . . . xix

1 Introduction 1

1.1 Motivation . . . 1

1.2 Outline . . . 2

2 Deep Neural Networks 3

2.1 Artificial Neuron . . . 3

2.2 Neural Network . . . 4

2.2.1 Fully-connected Layer . . . 5

2.3 Convolutional Neural Network . . . 5

2.3.1 Convolution Layer . . . 6

2.3.2 Pooling Layer . . . 7

2.3.3 Upsampling Layer . . . 8

2.4 Datasets for DNN training and testing . . . 8

2.5 Image classification DNN Models . . . 8

2.5.1 LeNet . . . 9

2.5.2 AlexNet . . . 9

2.5.3 VGG . . . 9

2.5.4 ResNet . . . 10

2.5.5 Performance comparison . . . 11

2.6 Object detection DNN Models . . . 12

2.6.1 R-CNN Family . . . 12

2.6.2 YOLO - You Only Look Once . . . 13

2.6.3 RetinaNet . . . 13

2.6.4 Performance comparison . . . 15

2.7 Conclusion 16

3 DNN implementations on FPGAs **17**

3.1 Parallelism Opportunities in DNNs 17

3.2 DNN accelerator 18

3.3 Data Quantization 19

3.3.1 Performance estimation models 20

3.3.2 Existing work experimental evaluation 21

4 RetinaNet embedded software implementation **23**

4.1 Quantization aware training framework 23

4.1.1 Brevitas 24

4.1.2 Quantization results 25

4.2 Baseline implementation of RetinaNet in C 28

4.2.1 Convolution layer implementation 29

4.2.2 Overall implementation of RetinaNet 32

4.2.3 Time profiling results for baseline RetinaNet 33

4.3 Embedded implementation of RetinaNet in C 36

4.3.1 Time profiling results for embedded software RetinaNet 38

4.4 Conclusions 38

5 Convolution accelerator hardware architecture **39**

5.1 Architecture Overview 39

5.1.1 Data storage format 41

5.1.2 Operations Scheduling 42

5.2 Processing Elements 43

5.2.1 Multiply and accumulate operators 43

5.2.2 Sum tree 46

5.2.3 Quantizer and ReLU 47

5.3 Activation memory 48

5.4 Weight memory 51

5.5 Convolution IP 51

5.6 Convolution accelerator results 54

5.7 Conclusion 56

6 RetinaNet hardware/software implementation **57**

6.1 Hardware/software system 57

6.1.1 Hardware architecture 57

6.1.2 Embedded software 60

6.2 Experimental results 61

6.3 Conclusion 63

7 Conclusion **65**

7.1 Future Work 66

7.1.1 Expand convolution layer acceleration to more layers 66

7.1.2 Optimize hardware accelerator 67

Bibliography **69**

A RetinaNet specifications **A.1**

B RetinaNet detailed time profiling **B.5**

List of Tables

2.1 Comparison of popular DNN's with the Top-5 error rate obtained with ImageNet dataset. . 11
2.2 Comparison of popular Object Detection models with mAP obtained using the COCO dataset. 16

3.1 Expected cost per operation for each precision type. 20
3.2 Classification accuracy for different implementations of AlexNet using different quantization strategies. 22

4.1 Comparison of trade-offs between different quantization models. 26
4.2 Time profiling of RetinaNet DNN. 34
4.3 Memory map of the DDR embedded system memory. 36
4.4 Time profiling of RetinaNet software implementation on embedded device. 38

5.1 Time profiling across the different loops of convolution accelerator. 54
5.2 Resource usage of convolution accelerator. 56

6.1 Resource usage of RetinaNet hardware/software implementation. 61
6.2 Time profiling of RetinaNet hardware/software implementation and comparison with software only implementation. 64

A.1 RetinaNet layer by layer specification part 1. . . . . A.1
A.2 RetinaNet layer by layer specification part 2. . . . . A.2
A.3 RetinaNet layer by layer specification part 3. . . . . A.3
A.4 RetinaNet layer by layer specification part 4. . . . . A.4

B.1 RetinaNet detailed time profiling part 1. . . . . B.5
B.2 RetinaNet detailed time profiling part 2. . . . . B.6

List of Figures

2.1 Artificial Neuron representation 4

2.2 Transfer function of activation functions 4

2.3 Neural Network with two hidden layers 5

2.4 Local receptive field of an hidden neuron. 6

2.5 Receptive fields of the neurons of a convolution layer 6

2.6 Convolution layer with three filters that inputs an image of 28x28 input neurons and outputs three feature maps of 24x24 neurons 7

2.7 Pooling layer in action, condensing 4 features to only 1 7

2.8 Representation of LeNet-5 architecture. 9

2.9 Representation of AlexNet architecture. 10

2.10 Representation of VGG architecture. 10

2.11 Training and testing error in a 20 layer and 56 layer CNN. Showing higher errors in the deeper network due to the vanishing gradient problem. 11

2.12 Representation of ResNet-18 architecture. 11

2.13 R-CNN algorithm architecture, consisting of several modules serially connected 13

2.14 YOLO algorithm architecture, consisting of one single DNN. 13

2.15 RetinaNet algorithm architecture, consisting of a feature pyramid network and fully connected sub-networks for classifying and produce bounding box localization 14

2.16 Convolution weights distribution by different parts of RetinaNet. 15

3.1 Algorithm for computing convolutional layers, composed by nested loops. 18

3.2 Overview of an accelerator general architecture. 19

3.3 Trade-offs for different quantization techniques using DarkNet as DNN model. 21

4.1 Representation of a DNN architecture prepared for quantization aware training. 24

4.2 Algorithm that introduces the quantization error in training, making it a quantization aware training. 24

4.3 Convolution and ReLU layer declaration in PyTorch. 25

4.4 Convolution and ReLU layer declaration using Brevitas library. 26

4.5 Accuracy of different quantization models along increasingly trained networks. 27

4.6 RetinaNet usage workflow. 28

4.7 Quantized convolution layer function definition in C. 30
4.8 Quantized convolution layer implementation in C. 31
4.9 4 bits reading function implementation in C. 31
4.10 Static allocation of memory for the C implementation of RetinaNet. 32
4.11 Function defining the memory arrangement of weights, bias and activations in the memory arrays. 33
4.12 Main function of quantized RetinaNet implementation in C. 35
4.13 Function defining the memory mapping in the embedded system implementation. 37

5.1 General overview of the accelerator hardware architecture. 40
5.2 HLS code that implements the interface of the hardware architecture. 41
5.3 Timeline of PE's operations. 42
5.4 PE hardware architecture. 43
5.5 Timeline of MAC's operations. 44
5.6 HLS code that implements the MACs in the PEs. 45
5.7 HLS code that implements the sum tree in the PEs. 46
5.8 HLS code that implements the PE's quantizer. 47
5.9 Matrices representing in 2D the 3D input activations, showing with colors the composition of the activation memory when different output pixels are being calculated 49
5.10 Activation memory hardware architecture. 50
5.11 HLS code that declares the activation memory. 50
5.12 HLS code that declares a load of a new set of values to the activation memory. 50
5.13 Weight memory hardware architecture. 52
5.14 HLS code that declares the weight memory. 52
5.15 General structure of the HLS code of convolution accelerator. 53

6.1 Hardware architecture of the RetinaNet hw/sw implementation. 59
6.2 Simplified hardware architecture of the RetinaNet hw/sw implementation. 59
6.3 Functions that issues the execution of the convolution hardware accelerator. 61
6.4 Functions that implements a convolution stage using the convolution hardware accelerator. 62
6.5 Critical path of RetinaNet hardware/software implementation shown in the floor plan of the targeted device. 63

List of Acronyms

ARM Advanced RISC Machine

ATM Automated Teller Machine

AXI Advanced eXtendible Interface

BRAM Block Random Access Memory

CNN Convolution Neural Network

COCO Common Objects in Context

CPU Central Processing Unit

DDR Double Data Rate

DMA Direct Memory Access

DNN Deep Neural Network

DSP Digital Signal Processor

FF Flip-Flops

FIFO First In First Out

FP False Positive

FPGA Field-Programmable Gate Array

FPN Feature Pyramid Network

GPU Graphics Processing Unit

HLS High-Level Synthesis

HW Hardware

IoU Intersection over Union

IP Intelectual Property

JPEG Joint Photographic Experts Group

LRN Local Response Normalization

LSVRC Large Scale Visual Recognition Challenge

LUT Look-Up Table

MAC Multiply And Accumulate

mAP Mean Average Precision

PC Personal Computer

PE Processing Element

PL Programmable Logic

PS Processing System

RAM Random Access Memory

R-CNN Region-based Convolutional Neural Network

ReLU Rectified Linear Unit

ResNet Residual Network

RGB Red Green Blue

SoC System-on-a-Chip

SW Software

SVM Support Vector Machines

TP True Positive

VGG Visual Geometry Group

VOC Visual Object Classes

YOLO You Only Look Once

Chapter 1

Introduction

The goal of this work is to develop an object detection deep neural network embedded system using a SoC-FPGA platform. The developed system should be as efficient as possible having a good balance between object detection accuracy and inference time.

A review of the existing object detection DNN is made, and their performances analyzed. The selected object detection DNN is RetinaNet due to its high accuracy and high inference times that show potential for hardware acceleration. RetinaNet is trained using quantization aware training, to reduce the model size with a minimal loss of accuracy. The results of the quantized models with different weights and activations bit-widths are analyzed, and a quantization with 4 bit-width weights and 8 bit-width is selected.

In this work implementations of RetinaNet in C are made form scratch. From the time profiling results of those implementations, the parts of RetinaNet that most need to be hardware accelerated are identified. To achieve this acceleration a dedicated hardware component is designed, and integrated in a hardware/software system. The hardware/software co-designed system results are compared with a software-only implementation. The final results are very promising, introducing a proof of concept for the expansion of the convolution layer acceleration to more layers.

Section 1.1 describes why the goal of this work is important, and section 1.2 presents the content in each chapter of this document.

1.1 Motivation

The evolution in deep neural networks (DNN) and it's application in image processing, unleash new developments in computer vision, namely in the object detection field, where greater accuracy is achieved [1]. There is a broad range of applications where object detection is applied and can be applied in the future, such as autonomous driving [2], video surveillance [3][4], Smart Cities [5], industrial robots and quality control [6], medical image [7] and people counting [8]. The combination of deep neural networks with object detection will further enhance those applications and open new application possibilities. This technological developments ultimately will improve society and human life.

In spite of the advantages brought by deep neural networks, they are very computer intensive [1] and applying them in embedded systems environments is still a challenge. In embedded systems both power and cost are important factors for a feasible solution, which forces the use of devices with limited computer resources. To run a deep neural network in a system with limited resources and strict real-time restrictions a customized hardware architecture is of highly importance. The customized hardware architecture implemented in a Field-programmable gate array (FPGA) can act as an accelerator of the deep neural network layers, and therefore achieve the inference time needed for the applications.

1.2 Outline

This document is divided in the following chapters:

- Chapter 2 introduces deep neural networks concepts, and the working principles of the main layers that compose them. Existing image classification DNNs and object detection DNNs are reviewed and compared, and an object detection DNN is chosen to be implemented in the developed system, as described in chapter 4.
- Chapter 3 explores how DNNs can be implemented on FPGAs, describes the parallelism opportunities in DNNs and reviews previous works implementing DNNs on FPGAs. Data quantization techniques are introduced, and existing studies made on the quantization of DNNs are reviewed.
- Chapter 4 describes the quantization training framework of the selected object detection DNN, RetinaNet, and presents the performance results of different quantizations. A quantization is chosen to be used in the implementations of RetinaNet developed in this work. An implementation of RetinaNet in C is described, showing the workflow used for its development and the obtained results. The most compute-intensive part of the RetinaNet DNN is identified to be accelerated on hardware, based in the analysis of these results. The performance of this embedded software implementation of RetinaNet is fully analyzed, and its results are used as a baseline for the subsequent analysis of the hw/sw system.
- Chapter 5 describes the architecture of the convolution accelerator hardware component that processes the convolution layers of RetinaNet DNN. A top-down approach is used, and every sub-component is described in detail. The performance and implementation results of the accelerator are fully analyzed.
- Chapter 6 describes the integration of the convolution hardware accelerator in the embedded system. The complete hardware/software architecture is detailed, and the achieved results are analyzed and compared with the baseline software-only embedded implementation.
- Chapter 7 presents the work conclusions and suggestions for future work.

Chapter 2

Deep Neural Networks

Traditionally computational problems are solved by defining explicitly an algorithm that can produce the intended output for a certain input. Neural networks open a new approach to the way computational problems are solved. Inspired by biology and the way human brain works, in neural networks we don't explicitly define the algorithm, instead we feed the neural network with sets of inputs and their corresponding outputs, and it learns the algorithm by itself, this process is called training. A well trained neural network can then produce an adequate output for any given input that do not belong to the training set.

This chapter introduces neural networks, starting by defining the basic unit, the artificial neuron in section 2.1, and then showing how this basic unit is used as a component to produce a neural network in section 2.2. Section 2.3, introduces the type of neural networks that will be used in this work. In the end of the chapter is reviewed the popular deep neural network models used for image classification (2.5) and object detection (2.6). In-spite of this thesis being focused on object detection, it is important to review the image classification DNN models, since they are sometimes used as the backbone of some Object detection models.

2.1 Artificial Neuron

Artificial neurons are the basic unit of neural networks. A neuron describes a mathematical function with several inputs but only one output. The most basic type of artificial neuron is the perceptron (figure 2.1), that takes as inputs j binary outputs and produces as output a single binary output [9]. Each of the binary inputs is multiplied by a different weight that quantifies the importance of the input for the final outcome. The sum of all the weighted inputs with an additional parameter called bias passes through the activation function to generate the output value, known as activation value. The simplest activation function is the step function depicted in figure 2.2 (a). The step function outputs one when its input is above zero, and zero otherwise. In this way perceptrons can model decision making systems and even compute simple logical functions, such as logic gates.

To effectively train a neural network, a small change in any weight or bias should also cause a

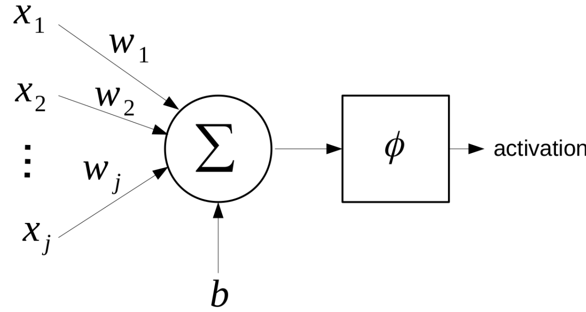


Figure 2.1: Artificial Neuron representation

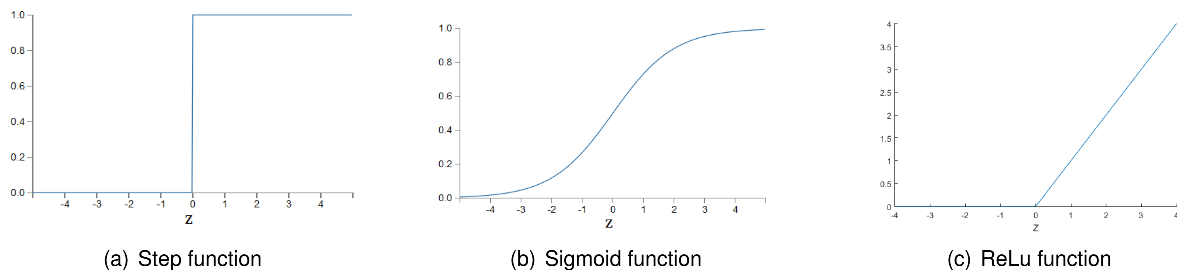
small corresponding change in the output, however perceptrons are not good at these, because a small change in any weight or bias can completely flip the output of the neuron. To solve this issue other artificial neuron types take as input any value between 0 and 1 and use non binary activation function, instead of having binary inputs and outputs.

For instance one of the possible activation function is the sigmoid function defined in equation 2.1. The sigmoid function smooths out the output of the artificial neuron, making it possible to always have a small change in the output as a result of a small change in a bias or weight. Comparing the transfer function of both the sigmoid function and the step function depicted in figure 2.2, it can be seen how much smoother is the sigmoid function. The activation value of the sigmoid neuron, this is a artificial neuron that uses as activation function the sigmoid function is mathematically defined in equation 2.2.

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{2.1}$$

$$activation = \sigma(\sum_j w_j x_j - b) = \frac{1}{1 + e^{-\sum_j w_j x_j - b}}. \tag{2.2}$$

Another example of an activation function is the rectified linear unit (ReLU) showed in figure 2.2 (c).



(a) Step function (b) Sigmoid function (c) ReLu function

Figure 2.2: Transfer function of activation functions, adapted from [9]

2.2 Neural Network

Artificial neurons can already make some basic logic functions, however to produce complex logic systems that can solve big computational problems we need to join several artificial neurons together, to

form a neural network. A neural network, as the one depicted in figure 2.3, consists in layers of artificial neurons whose outputs are connected to the inputs of the artificial neurons of the next layer. The left-most layer of the network is called the input layer, the inputs of the neurons of this layer are used as the inputs of the system. The rightmost layer is called the output layer, and the outputs of the neurons in this layer are used as the outputs of the system. In the middle, the hidden layers have their outputs and inputs connected to adjacent layers.

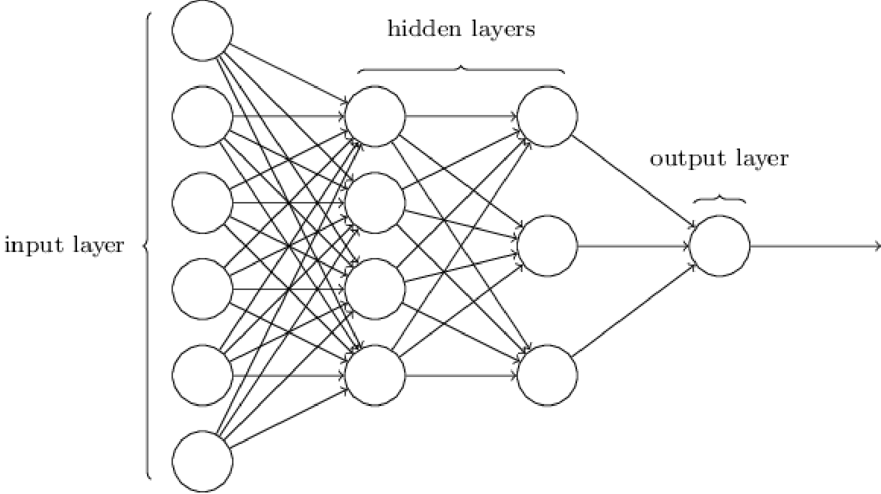


Figure 2.3: Neural network with two hidden layers [9]

2.2.1 Fully-connected Layer

The most basic type of neural network layer is a fully connected. In a fully-connected layer, all inputs are connected to every output. This means that all the neurons that compose this layer have a connection to all the inputs of this layer, as shown by the neural network model in figure 2.3.

2.3 Convolutional Neural Network

A convolutional neural network (CNN) is the most used type of neural network for image processing, since it takes into account the spatial structure of the image and allows deeper neural networks for the same amount of memory and computational resources than neural networks composed only by fully-connected layers.

In a CNN each hidden neuron is only connected to some of the outputs of the previous layer, instead of being connected to all of them, which in turn reduces the number of operations that have to be dealt in a single neuron. For instance when we are dealing with images, an artificial neuron of the first hidden layer, is only connected to some localized region of the input image, as we can see in figure 2.4, this is what takes into account the spatial structure of the image. The spatial extent of this connectivity is a parameter called the receptive field of the neuron. In this way from now on the array of inputs and outputs of a given layer can be seen as a matrix in order to represent that spatial structure, this matrix are called input feature map and output feature map respectively.

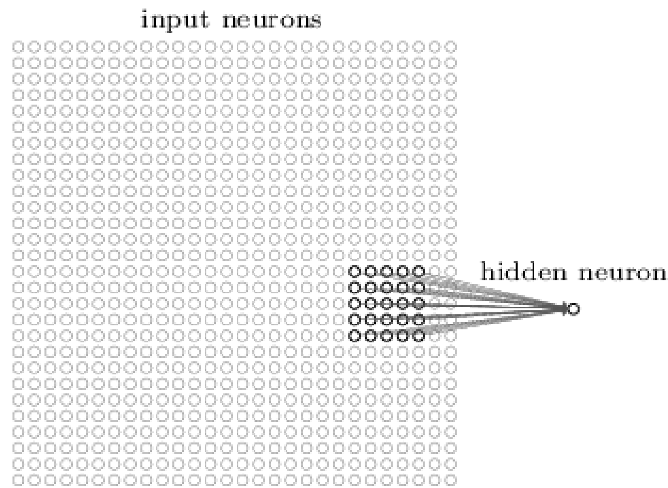


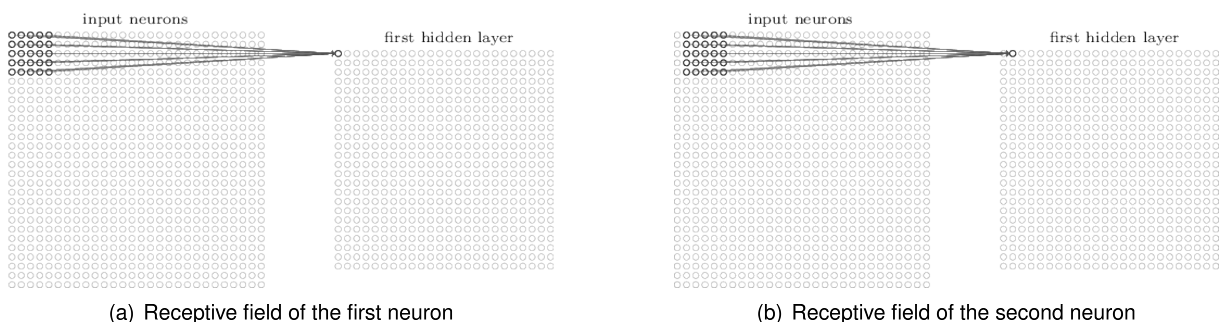
Figure 2.4: Local receptive field of an hidden neuron [9].

Furthermore, CNN also share the weights and bias between different neurons of the same layer, which also reduces the memory consumption of the CNN.

The following subsections 2.3.1, 2.3.2, 2.3.3 detail the main functional layers of a CNN.

2.3.1 Convolution Layer

The convolution layer is the main layer of a CNN. This layer is composed by several filters, each filter represents a set of neurons that share the same weights and bias. If we choose to study a specific filter we see a matrix of neurons where each one of those neurons use a different local receptive field that is similar to the neighbor neuron receptive field but shifted, this can be seen in figure 2.5. In this way, each neuron of a filter uses the same weights and bias, in spite of having different inputs of equal size, this contributes to reduce both the memory usage and the number of operations without losing the spatial structure of the image.



(a) Receptive field of the first neuron (b) Receptive field of the second neuron

Figure 2.5: Receptive fields of the neurons of a convolution layer [9]

The step of the shift of the receptive field when changing between neighbor neurons, it is called stride, and can assume different values depending on the architecture of the CNN. The outputs of the neurons of a filter is called the feature map, that is usually smaller than the input image because the displacement of the receptive field through the input image will be limited to its borders, and so the number of times you can shift the receptive field that translates directly to the number of neurons. However a convolution

layer has a set of filters, that produce a set of feature maps, that can make the total number of outputs produced higher than the inputs, as for example can be seen in figure 2.6.

Each filter uses different weights and bias for their neurons. Should also be taken into account that the input of a convolution layer, can also be an output of a previous convolution layer with several filter, and as a result outputs several feature maps (these can be also called channels). In this situation the receptive field will be tri-dimensional, where the dimension in the channel axis will be equal to the number of channels, traversing all the channels. Following the receptive field dimension, the weighs will have to be tri-dimensional too.

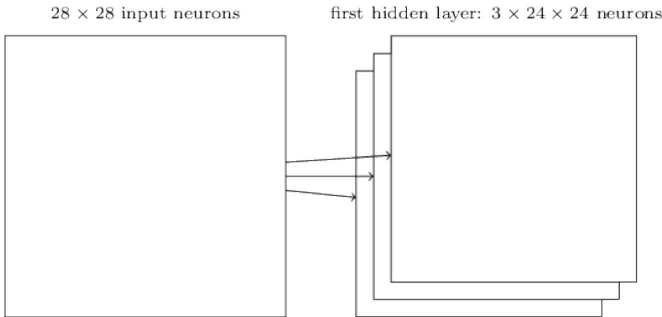


Figure 2.6: Convolution layer with three filters that inputs an image of 28x28 input neurons and outputs three feature maps of 24x24 neurons [9]

Considering a convolution layer whose receptive fields are of size KxK, and with an input (g) with N channels where w is the tensor of weights and b the tensor of bias, the output of the neuron (f) in the position i row, j column of the filter l is modeled by equation 2.3.

$$f(i,j,l) = b(l) + \sum_{x=1}^{K}\sum_{y=1}^{K}\sum_{z=1}^{N} g(x+i, y+j, z).w(l,x,y,z) \tag{2.3}$$

2.3.2 Pooling Layer

The pooling layer is used usually after the convolution layer to reduce the amount of information stored in the output feature map and consequently reducing the computational load of the next layers. Pooling downsamples the results of a layer by aggregating neighboring results in a single result, as seen in figure 2.7.

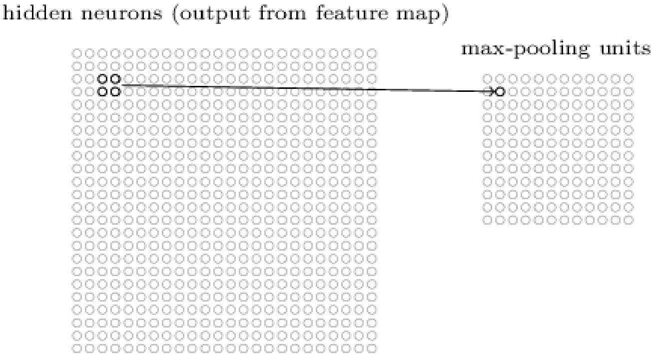


Figure 2.7: Pooling layer in action, condensing 4 features to only 1 [9]

In practice one of the ways that this compression can be achieved might be by choosing the biggest of those regional values to represent that region, this is called *max-pooling*.

2.3.3 Upsampling Layer

An upsampling layer do the opposite operation of the pooling layer, getting a bigger output feature map than the input feature map. There are different implementations of upsampling layers:

- Nearest neighbors — mirrors the value of an input pixel into several output pixels neighboring each other;
- Bi-Linear Interpolation — determine the value of an output pixel by interpolating the value of neighboring input pixels.

This layer is usually used in CNNs with multiple data paths that are joined together. For example when summing two feature maps of different dimensions, the smaller feature map is upsampled to have the same dimension of the other.

2.4 Datasets for DNN training and testing

To test new DNN models and their performance, it was introduced competitions such as the Pascal VOC challenge (yearly between 2005 and 2012) [10] and the ImageNet Large Scale Visual Recognition Challenge (LSVRC) (yearly between 2010 and 2017) [11]. Challenges are composed of two components: a public dataset with training and validation data; and an annual competition with a workshop. The first part is used to train the developed DNN with the training dataset and test its performance against the validation dataset, helping out in the development of the algorithms. The second part provides a way to track the progress and discuss the lessons learned from that year entries.

This contests are divided in several categories and they both present categories destined to image classification and object detection. In each one of the next two sections there is a subsection (2.5.5 and 2.6.4) intended to compare the performance of the introduced models. This comparisons are made possible using as equal base point the challenges datasets.

More recently a new more demanding dataset was presented, this is the Common Objects in Context (COCO) dataset [12].

2.5 Image classification DNN Models

Image classification is a process where is identified the object depicted in the image, in accordance to a set of possible classes [13]. The implementation of image classifiers with DNN is done using as the input layer the matrix of pixels that compose the image, and as output a set of values that show the confidence of that image belonging to each one of the classes.

2.5.1 LeNet

The first convolutional neural networks for image classification was the LeNet model, that was introduced in 1989 [13]. This was a very shallow network, the latter version of this neural network, LeNet-5 was composed of a total of 6 layers (2 convolutional, 2 average pooling and 2 fully-connected) [11]. LeNet uses as activation the sigmoid function. This network was designed to classify grayscale images of handwritten digits with a resolution of 32 by 32. This was the first CNN that led to a commercial success, since it was deployed in the ATM machines for recognizing digits in check deposits. Figure 2.8 shows the architecture and some specifications of this neural network.

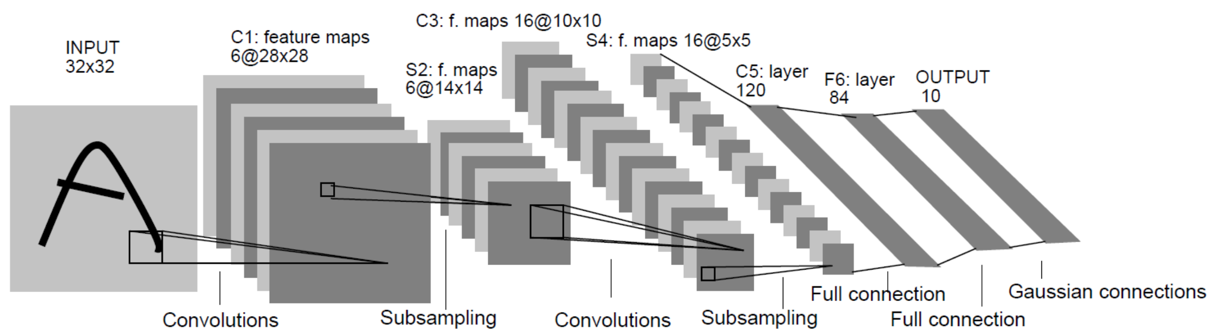


Figure 2.8: Representation of LeNet-5 architecture [11].

2.5.2 AlexNet

AlexNet was the winner of 2012 image classification contest of the ImageNet LSVRC. It has a total of 11 layers (5 convolutional, 3 Max pooling and 3 fully-connected) [14]. It receives as input a 227x227 image with 3 channels one for each RGB color. AlexNet uses a ReLU as activation function decreasing the complexity of the activation function, compared to the sigmoid or tanh function. This DNN also introduces the use of Local Response Normalization (LRN), performed after the ReLU in the first two convolutional layers. A LRN normalizes the output of a neuron in accordance to the neighbor neurons activation value. This makes sure to attenuate local uniformly large activations and enhance the differences between neighbor neurons, increasing the sensitivity to high frequency features. This network architecture is depicted in figure 2.9 along with the specifications of each layer.

Several techniques were used in this network training to increase it's accuracy, such as feeding the network with crops of 256x256 images and duplicating the number of images by mirroring them horizontally, increasing the training set by reusing the same image but with different crops and mirroring settings. To avoid overfitting a technique called dropout was used. This is dropping out arbitrarily neurons from the network temporarily, so that every input image goes throught a different architecture.

2.5.3 VGG

The VGG DNN participated in the ImageNet LSVRC contest of 2014, winning in some of the categories. This network is composed by 24 layers (16 convolutional, 5 Max Polling and 3 Fully-connected). This

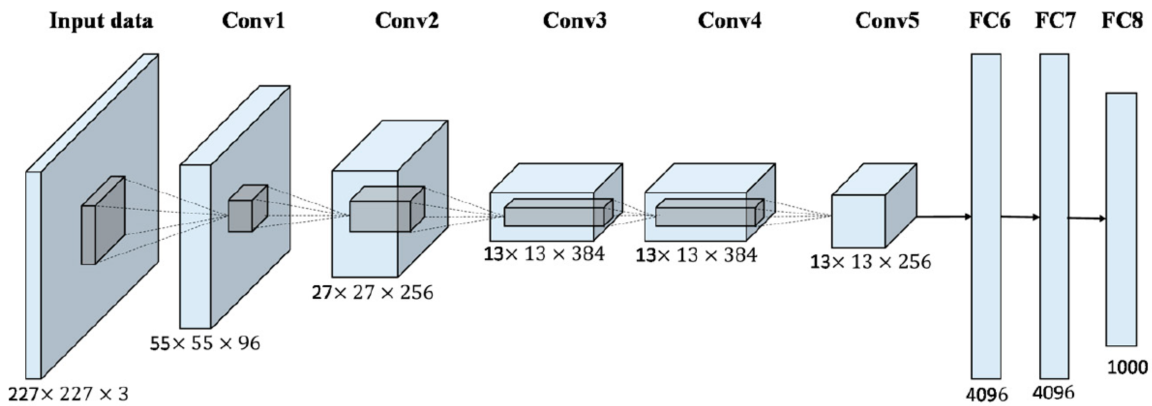


Figure 2.9: Representation of AlexNet architecture [15].

network architecture is very uniform. In figure 2.10 it is shown this network model.

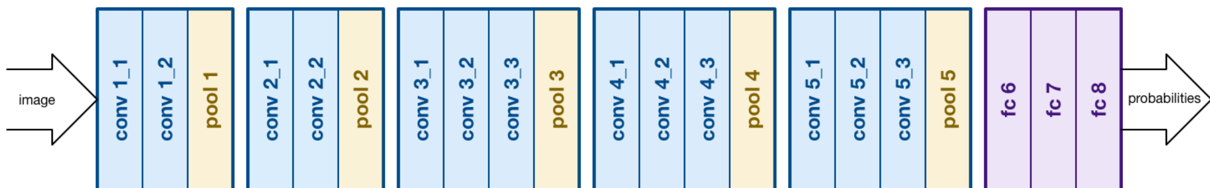


Figure 2.10: Representation of VGG architecture.

2.5.4 ResNet

ResNet introduces even more layers than the previous networks, reaching a total of 152 layers in its bigger version [16]. This deep neural network was the winner of the 2015 ImageNet challenge in the classification task, exceeding for the first time human-level accuracy.

The deeper the network, the harder it is to train it. Deeper networks suffer from vanishing gradient during training, degrading the accuracy of the network as seen in figure 2.11 [13]. To understand the vanishing gradient problem, we have to take into account that activation functions usually squishes a large input space into a tiny output, having a small derivative. The vanishing gradient problems is caused by the impact of the successive accumulation of several activation functions through back propagation, making it difficult to adjust the weights of the first layers.

ResNet accomplishes a effective very deep neural network making use of shortcut connections. The shortcut connections are used to skip some layers, adding up the input of a set of layers directly with those layers output. The ResNet is organized in blocks of layers that are bypassed by shortcuts as showed in figure 2.12. The shortcut connections diminish the impact of small derivative activation functions through the backpropagation, effectively solving the vanishing gradient problem.

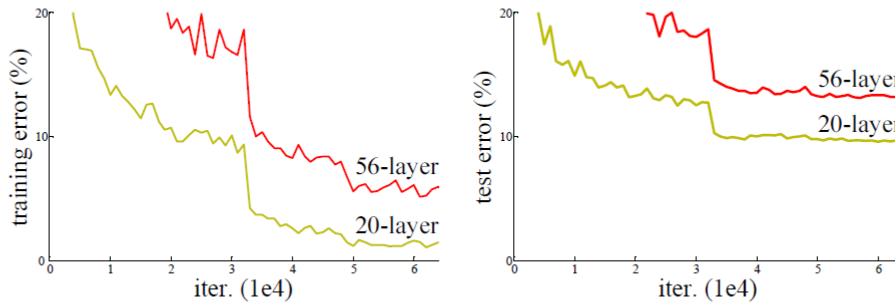


Figure 2.11: Training and testing error in a 20 layer and 56 layer CNN. Showing higher errors in the deeper network due to the vanishing gradient problem [16].

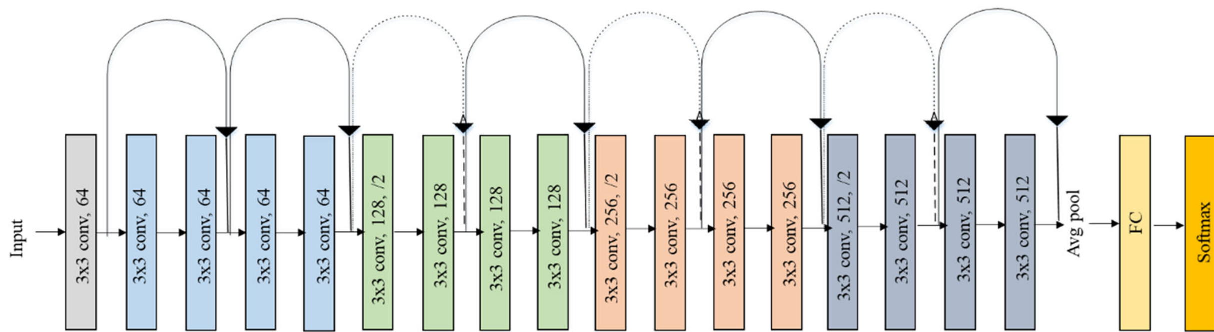


Figure 2.12: Representation of ResNet-18 architecture [17].

2.5.5 Performance comparison

In the category of image classification of image processing challenges usually it is used the Top-5 or Top-1 error metric. The Top-5 is the fraction of test images whose label was among the 5 most probable labels for that predication. Top-1 evaluates if the predicted class was in fact the truth class of the input image, being a similar but more demanding accuracy metric than Top-5.

Characteristics	LeNet 5	AlexNet	VGG 16	ResNet 18	ResNet 50
Input Size	28x28	227x227	224x224	224x224	224x224
No. of Convolutional layers	2	5	13	20	53
No. of Fully-Connected layers	2	3	3	1	1
No. of Channels of Conv. layers	1, 20	3-256	3-512	3-512	3-2048
No. of Filters of a Conv. layers	20, 50	96-384	64-512	64-512	64-2048
Weights	60k	61M	138M	11.1M	25.5M
MACs	341k	724M	15.5G	1.8G	3.9G
Top-5 error	-	19.8	8.8	10.9	7.0

Table 2.1: Comparison of popular DNN's with the Top-5 error rate obtained with ImageNet dataset.

Table 2.1 shows a comparison between the different CNN introduced earlier. Showing the Top-5 error rate for each one of the models apart from the LeNet, obtained using the ImageNet dataset.

Table 2.1 shows that in general deeper networks achieve higher accuracies. From LeNet 5 to AlexNet the number of weights and the number of multiply and accumulate (MAC) operations have increased about a thousand times and two thousand times respectively, in spite of only having doubled the number of layers. The disproportionate increase of computational resources is due to the increase resolution of

input image and the increased number of filters per convolutional layer. This table also shows that from VGG 16 to ResNet 50 the number of weights and MACs has fallen despite ResNet 50 having a better accuracy and being a deeper network then VGG 16, this is because the first layers of VGG have a high number of filters.

2.6 Object detection DNN Models

Object detection is used to locate the object position and size, and predict its class [18]. The output of the system is composed of a set of variables that specify the dimensions and location of the box that frames the object, such as x position, y position, weight and height. In addition, for each one of the detected boxes there is an array of confidence values stating the confidence of the framed image belonging to each one of the possible classes. There are several ways of implementing an object detection system, some of them make use of hybrid system, this is system with both convolutional neural networks and traditional image processing algorithms.

2.6.1 R-CNN Family

One way to architect a object detection system, is to divide it in several stages. Region-based Convolutional Network (R-CNN) consists of three modules [19]. The first module generate category independent region proposals of different sizes. The second module is a deep convolutional neural network that extracts from a region of an image a feature map. The third module is composed by a set of support vector machines (SVM) [20] that classify the image using the feature map. All of these modules are serially connected, as shown in figure 2.13. Each one of the proposed regions by the first module are independently fed to the second module to produce a different feature map for each one of the regions. In this way for each test image the second module will have to produce as many output feature maps as the number of proposed regions, making this model computational intensive. R-CNN is a hybrid solution between CNN and traditional algorithms, being the second module the only module with a CNN.

R-CNN was introduced in 2014 and since then there were multiple variations of this algorithm, one of them was Fast R-CNN [21]. Fast R-CNN improves both the accuracy and the computational resources used by the model. This model uses the entire input image to extract the features and then uses the region proposals to extract from the entire image feature map a subset of features corresponding to that region. By sharing the feature map the computational operations can be greatly reduced.

Another variation of R-CNN and the most advanced one is the Faster R-CNN [22]. Faster R-CNN is similar to the Fast R-CNN being the main difference, the way that region proposals are computed. Faster R-CNN uses a separate CNN to compute the region proposals, instead of the time consuming selective search algorithm used in predecessors models.

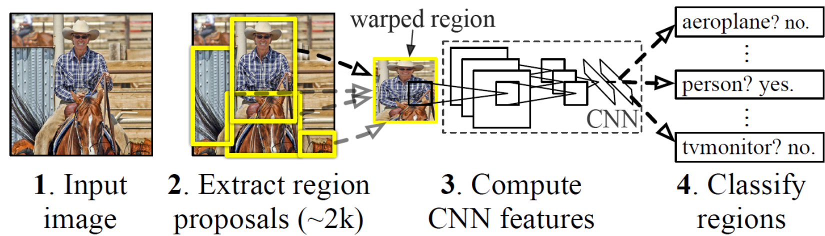


Figure 2.13: R-CNN algorithm architecture, consisting of several modules serially connected [19]

2.6.2 YOLO - You Only Look Once

You Only Look Once (YOLO) approaches object detection in a different way, using a single CNN to both locate and classify objects in an image [23]. This unified approach is only composed by a single stage as seen on figure 2.14.

To understand the output of this model, take into account that it divides the input image in several regions of the same size. Each region can only output a limited predefined number of objects, lets say B objects. However an object can spread over several regions, being outputted only by the region where its center is located. Associated with each object there are outputted 5 predictions to characterize the bonding box of the object (x position, y position, weight, height and confidence) and N predictions with the probability of that object belonging to one of the N existent classes. In this way each region outputs $B * (N + 5)$ predictions.

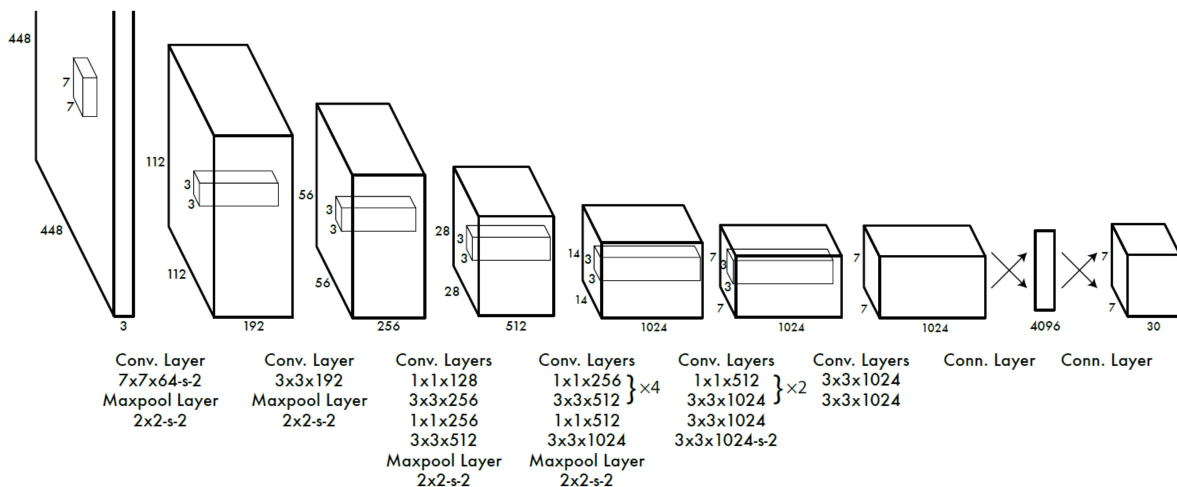


Figure 2.14: YOLO algorithm architecture, consisting of one single DNN [23].

2.6.3 RetinaNet

RetinaNet, just like YOLO uses a single DNN to both classify and locate objects. Most of the layers of this model are convolution layers and no fully connected layers are used [24]. This DNN is grouped in smaller sub-networks:

- ResNet backbone — This is where the input image is received. The ResNet used here differs

from the one described earlier in subsection 2.5.4, since the fully connected layer in the end is not implemented. This DNN is composed by 5 stages that output smaller and smaller feature maps as the stage number increases.

- Feature Pyramid Network (FPN) [25] — This DNN enhance the capability to detected small objects [25]. In the FPN, the output feature map size is being increased along the network as seen in figure 2.15. In each stage of the FPN a new feature map from the backbone is added to the results from the previous stage and upsampled as shown in figure 2.15. This technique counteracts the vanishing of small objects from the successive downsampling of the ResNet by successively upsampling the backbone last stage with the help of lateral connection to middle layers of the backbone.

- Classification — There is a network of this type attached to each one of the stage outputs of the FPN as shown in figure 2.15. In this way the classification sub-networks receives input feature maps of different dimensions, each one being prone to detect objects of different sizes. These subnetworks are composed by 5 convolution layers that classify the objects. The output of each one of the subnetworks are concatenated to achieve a final result with objects of different sizes. The weighs and bias are shared among all the classification sub-networks.

- Regression — These sub-networks are very similar to the ones used for classification, there are several instances of the sub-network connected to different stages of the FPN and the weights and bias are shared among all of them. The regression sub-network is responsible for getting the bonding boxes parameters of every object. The result of each subnetwork is concatenated.

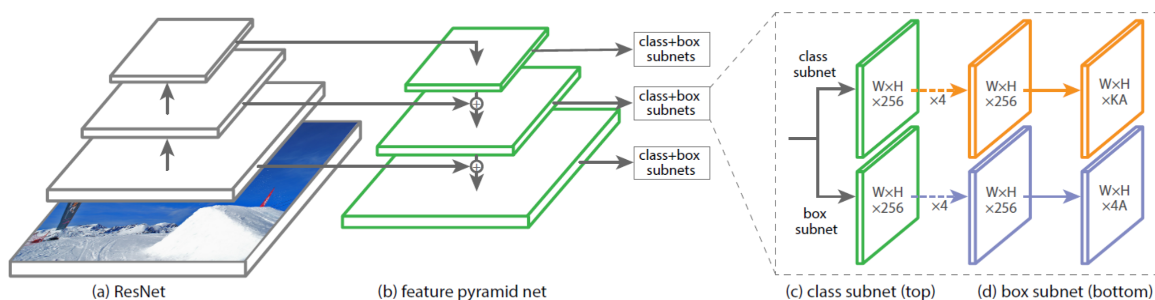


Figure 2.15: RetinaNet algorithm architecture, consisting of a feature pyramid network and fully connected sub-networks for classifying and produce bounding box localization [24].

The RetinaNet output format is similar to the one used in YOLO presented in subsection 2.6.2. The classification sub-networks output an array of activations with probability of an object belonging to each one of the classes, and the regression sub-networks output an array of activations that characterize the bonding box around each object.

The convolution weight distribution across the different sub-network of RetinaNet is shown in figure 2.16. A more detailed view of the layers that compose RetinaNet is showed in appendix A.

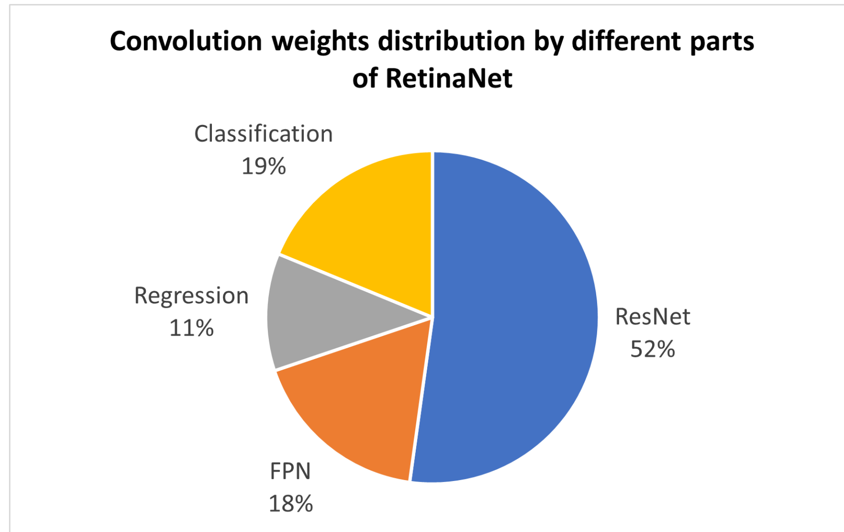


Figure 2.16: Convolution weights distribution by different parts of RetinaNet.

2.6.4 Performance comparison

In object detection the evaluation of the effectiveness of the models has to take into account the capacity to correctly classify objects as well as the ability to tell their localization. In this way there is a metric called Intersection over union (IoU) that quantifies how good the predicted bounding box (B_p) location corresponds to the true bounding box (B_t). IoU works by comparing the area of the intersection with the area of the union of the bounding boxes that were predicted and the bounding boxes that should have been predicted as show in equation 2.4. This metric is used as a threshold, so that any prediction above a certain value of IoU is considered a True Positive (TP), this is a correct prediction, and in the contrary any prediction with a value below that threshold is considered a False Positive (FP), this is a bad prediction.

$$IoU = \frac{area(B_t \cap B_p)}{area(B_t \cup B_p)}. \tag{2.4}$$

The most used metric for the evaluation of object detection models is Mean Average Precision (mAP). This metric takes into account the confidence of the classification, and uses a IoU threshold to distinguish between a TP and a FP. Table 2.2 can be seen the mAP metric for different models using COCO as dataset.

Along with the mAP metric in table 2.2 it is showed the measured inference time for each one of the models, apart from the R-CNN family models, when executing these models in a Nvidia M40 GPU.

By the performance results in table 2.2 it can be concluded that as the accuracy increases the inference time that it takes for the model to make its predictions also increases. YOLO is a fast model and ideal for real time applications with strict temporal restrictions, but its biggest implementation has an mAP of 4.8% lower than the biggest implementation of RetinaNet showed in this table.

Models	No of Convolutional layers	Input image width	mAP (%)	Inference time (s)
Fast R-CNN	16	-	19.3	-
Faster R-CNN	32	-	21.5	-
YOLO v2	19	224	21.6	25
YOLO v3	53	320	28.2	22
	53	416	31.0	29
	53	608	33.0	51
RetinaNet	50	500	32.5	73
	101	500	34.5	90
	101	800	37.8	198

Table 2.2: Comparison of popular Object Detection models with mAP obtained using the COCO dataset [24][22].

2.7 Conclusion

In this chapter different image classification neural networks and object detection neural networks were compared with each other.

The chosen object detection network to be implemented in chapter 4 is RetinaNet-18 (RetinaNet with a ResNet-18 backbone), due to the high accuracy and high inference times that show potential for hardware acceleration as seen in table 2.2. To facilitate an embedded implementation the RetinaNet version chosen uses the smaller available backbone for this DNN, ResNet-18.

Chapter 3

DNN implementations on FPGAs

The computation of a DNN is highly intensive. To achieve high performance computation with low power consumption can be used specific hardware accelerators for DNN processing. To implement this architecture a FPGA can be used. The FPGA is an integrated circuit with a programmable hardware architecture, that can exploit the parallelism opportunities in the computation of DNNs.

The first section of this chapter, section 3.1, identifies the parallelism opportunities of DNNs, and section 3.2 presents possible hardware architectures that have been proposed to exploit those opportunities. At last, section 3.3, identifies a way to improve computation speed using different data quantizations and presents existing work results using quantization techniques.

3.1 Parallelism Opportunities in DNNs

Deep Neural Networks present several opportunities to explore parallelism and pipelining, to maximize the throughput of the available hardware. A convolution layer can be seen as a set of nested loops that can be computed with the algorithm in figure 3.1 which is based in equation 2.3. A technique called spacial unrolling can be used, which is the hardware equivalent of loop unrolling in software, where a certain loop is eliminated or shortened by making those operations in parallel in different hardware rather than sequentially [26]. These loops and the existing data dependencies, promote different forms of parallelism [27]:

- Inter-layer parallelism.

  The sequence of layers of a DNN can be pipelined starting the computation of a given layer before finishing the previous layer.

- Intra-layer parallelism.

  Inside a convolutional layer there are almost no data dependencies, therefore there are several sources of parallelism:

  - Inter-Feature map parallelism (Unrolling of the output channel loop).

The generation of a output feature map is independent from the generation of the other output feature maps that are being produced in the same layer. As such, all the filters can be processed in parallel, at the same time.

– Intra-Feature map parallelism.

  Inside a given feature map there are nested two more sources of parallelism:

  * Inter-Activation parallelism (Unrolling of feature map height and width loops).
    Each activation of a given feature map is calculated independently of each other and can be computed simultaneously.
  * Intra-Activation parallelism (Unrolling of filter height and width loops).
    An activation is calculated as set of multiplications summed together. Each one of those multiplications can be done independently and computed simultaneously.

for $k = 0 : K$ **do** *(Output channel)*
  for $c = 0 : C$ **do** *(Input channel)*
    for $y = 0 : Y$ **do** *(fmap height)*
      for $x = 0 : X$ **do** *(fmap width)*
        for $f_y = -\frac{F_Y - 1}{2} : \frac{F_Y - 1}{2}$ **do** *(Filter height)*
          for $f_x = -\frac{F_X - 1}{2} : \frac{F_X - 1}{2}$ **do** *(Filter width)*
            $O[k][x][y]$ +=
              $I[c][x + f_x][y + f_y] \times W[k][c][f_x][f_y]$

Figure 3.1: Algorithm for computing convolutional layers, composed by nested loops, adapted from [26].

3.2 DNN accelerator

Most existing work follow the DNN accelerator general architecture depicted in figure 3.2. The basic component of the accelerator is the processing element (PE), which performs the computation for the most important layers, such as the convolutional and fully-connected layers [13]. The accelerator must have several PEs to make it possible to compute operations in parallel. The composition of a PE varies with the implemented architecture and one or more multiply and accumulate (MAC) units may be available per PE.

Generally a large enough external memory is required to store all the weights and partial results of a complete DNN. As such, the data should be transferred to the FPGA in order to be processed. The big latencies in acessing the external memory are a big drawback when randomly accessing the memory. In this way a memory inside the FPGA is created so that chunks of data can be transfered from external memory and take advantage of pipelined memory access, that can achieve reasonable bandwidths. The on FPGA memory is usually very limited and therefore, a carefull choice of the data to be buffered is important. The data that can be stored in the local FPGA memory are weights, bias, input activations, partial sums, output activations and any misture of the previous data.

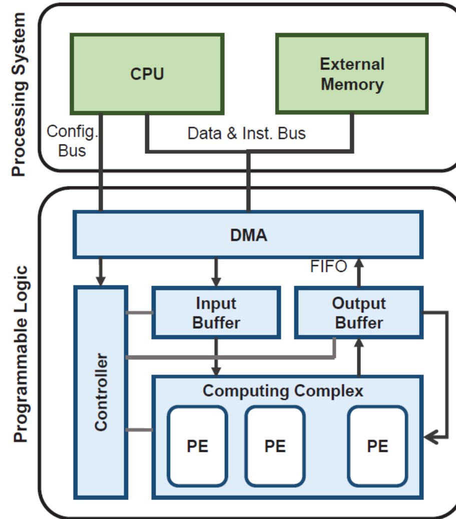


Figure 3.2: Overview of a DNN accelerator system general architecture [28].

In some cases not even the totality of a single type of data fit in the internal memory, in those data tiling are usually used. Data tiling is a technique that divides a set of data in smaller sets, following a specific strategy, so that each set can be transfered individually to limited capacity memories.

In figure 3.2 is also shown a direct memory access (DMA) that enables the FPGA to access the external memory independently of the CPU. This is usually used to enable better bandwiths, and offload the CPU of memory transferences.

The controller showed in figure 3.2 schedules the operations to be made in every clock cycle by adressing the internal memories in a specific way. In any case the CPU have some degree of control usually issuing which tiles of data are being transfered into the internal memory.

Existing works take different approaches to the specific dataflow used, scheduling the order of operation in different ways [13] [29].

3.3 Data Quantization

Data quantization changes the data type used, reducing the number of bits and using fixed point instead of floating point. This technique was introduced to reduce the complexity of the implementation. A reduced number of bits uses less storage capacity and reduces the size of the operators. The representation of data with a reduced number of bits can increase the throughput of the accelerator but may degrade the accuracy of the model, thus being important to study the best trade-off.

Quantization aware training is a technique used to minimize accuracies drops. This technique takes into account in the training process the quantization errors produced by quantizing weights and activations, so that the produced weights are already tweaked to minimize the quantization error [30].

Existing works study the trade-off between different quantizations and accuracy of the network, compared to the model implemented using 32 bit floating point. Subsection 3.3.1 introduces theoretical estimation models used as framework to compare different quantizations. Subsection 3.3.2 shows some

Datatype	$LUTs_{/op}$	$DSPs_{/op}$	C_{avg} x 10^{-6}	C_{rel}
Binary	5.58	0	12.02	1
INT2	13.52	0	29.12	2.42
INT4	30.06	0	64.76	5.39
INT8	86.38	0	186.02	15.48
INT16	28.66	1	181.16	15.07
FP32	-	-	766.6	63.79
Total resources for Xilinx KU115	663360	5520	-	-

Table 3.1: Expected cost per operation for each precision type [31].

experimental results obtained in previous works.

3.3.1 Performance estimation models

Two theoretical models were proposed by [31], to formulate a framework for the trade-off study between different data quantizations.

The first model relates hardware cost with parameter precision type. The model depicted in equation 3.1, calculates the average cost (C_{avg}) that takes to complete an operation, where $LUTs_{total}$ and $DSPs_{total}$ are the total number of LUTs and DSPs on the target device respectively, LUT_{usage} and DSP_{usage} are the percentage of LUTs and DSPs that can be used for arithmetic operations, $LUTs_{/op}$ and $DSPs_{/op}$ are the needed LUTs and DSPs to perform an operation for a given quantization. [31] defines an operation as a XNOR logic and popcount structure when referring to binary quantizations and a MAC when referring to other quantization types. The average cost can be interpreted as the fraction of the target device resources required to perform one operation in a given quantization. Table 3.1 shows the expected cost per operation for each precision type using as a target device the Xilinx Kintex UltraScale 115, considering the given constants: $LUT_{usage} = 0.7$, $DSP_{usage} = 1$, $LUTs_{total} = 663,360$ and $DSPs_{total} = 5,520$. The C_{rel} showed in table 3.1 is used to compare the cost of binary quantization with other datatypes, for e.g. a quantization using as datatype INT4, where $C_{rel} = 5.39$, uses 5.39 more resources than binary datatype.

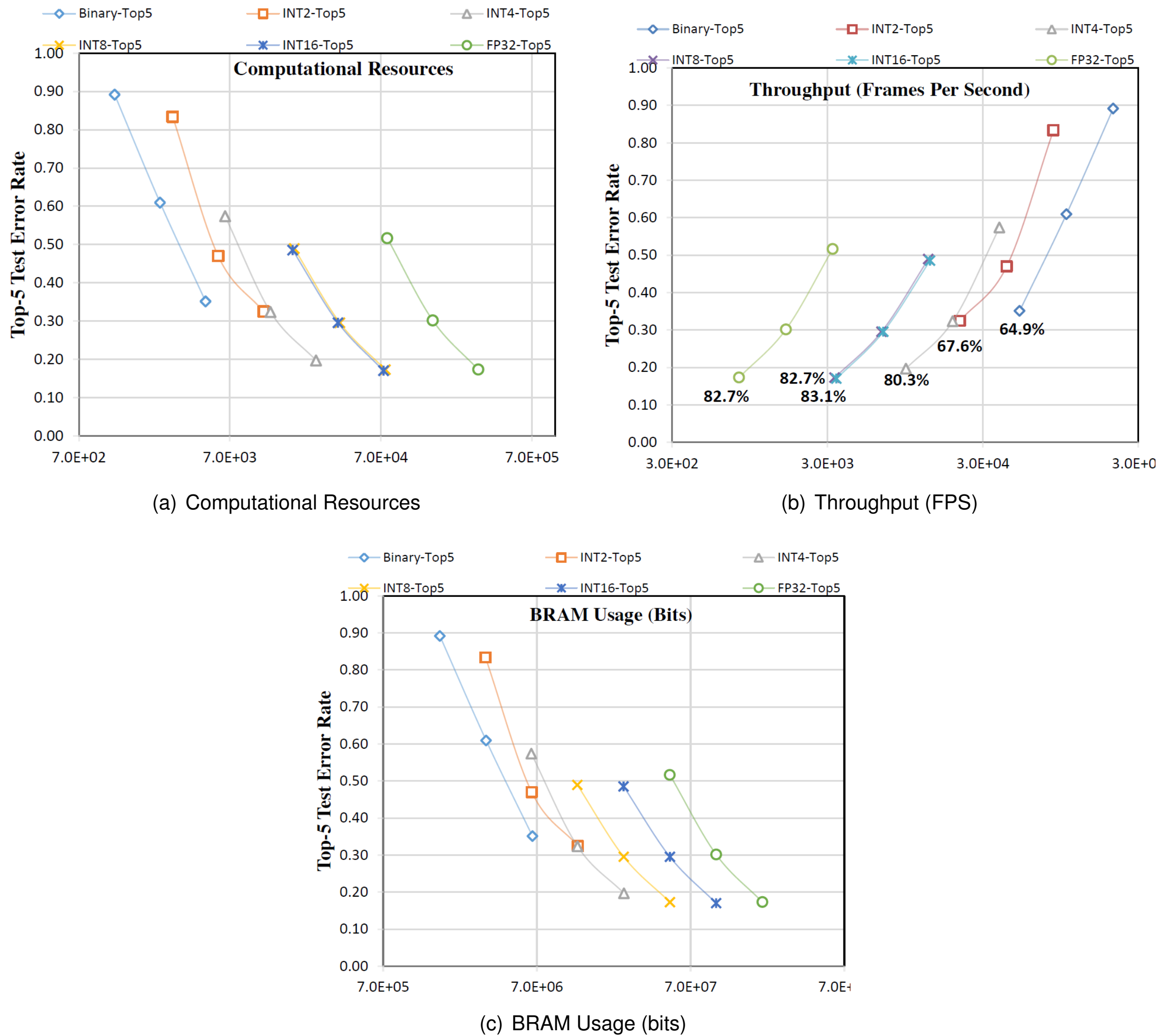
$$C_{avg} = max(\frac{LUTs_{/op}}{LUT_{usage} * LUTs_{total}}, \frac{DSPs_{/op}}{DSP_{usage} * DSPs_{total}}) \tag{3.1}$$

The second model presented by [31], relates the throughput with the hardware cost, that is estimated in the previous model. This relationship is formulated theoretically by equation 3.2, where $Freq$ is the clock frequency, $\#OP$ is the number of operations needed to conclude a complete processing of the DNN for a single input frame, C_{avg} is the average cost determined on the previous model and Δ is to count for extra resource overhead used for control logic. This model considers that an operation as defined earlier takes a single clock cycle to conclude. The C_{avg} is a ratio between the required resources for one operation and the total budget of resources, taking this in mind this model uses C_{avg} to apply a folding effect to the number of operations, this makes the model take into account the capability of the target device to parallelize operations, according to the available resource. The throughput is measured in frames per second.

$$Throughput \simeq \frac{Freq}{\#OP * C_{avg} * \Delta} \tag{3.2}$$

3.3.2 Existing work experimental evaluation

Using the models defined in section 3.3.1, [31] estimated the throughput in frames per second, the hardware cost as computational resources and block RAM (BRAM) usage. This evaluation was made using DarkNet as the DNN model, trained with the ImageNet dataset. Darknet is the model used by YOLO as backbone for image classification. The number of filters in each convolutional layer was multiplied by a scaling factor in order to expand or shrink the model obtaining implementations with different accuracies, throughputs and computational resources to evaluate the accuracy/computation trade-off at different DNN sizes. The DNN was implemented for this tests using all the combinations of different scalling factors and quantizations, obtaining figure 3.3.



(a) Computational Resources

(b) Throughput (FPS)

(c) BRAM Usage (bits)

Figure 3.3: Trade-offs for different quantization techniques using DarkNet as DNN model [31].

The results in figure 3.3, indicate that both binary and INT2 data types cannot achieve comparable accuracy to the other data types. 32 bit floating point (FP32) does not show better accuracy than INT8

Activation bitwidth	Weights bitwidth				Accuracy (Top-1)
	First CONV	Mid CONV	Mid FC	Last FC	
FP32	FP32	FP32	FP32	FP32	55.9%
8	8	8	8	8	54.6%
8	8	2	2	8	53.3%
8	8	2	1	8	52.6%
8	8	1	1	8	51.1%
4	8	2	1	8	49.3%
2	8	2	1	8	46.1%

Table 3.2: Classification accuracy for different implementations of AlexNet using different quantization strategies [32].

and INT16, having a highest top-5 accuracy of 82.7% comparing to the INT8 and INT16 highest accuracy of 82.7% and 83.1% respectively. In addition FP32 uses more computational resources, consuming more BRAM, has a lower throughput and does not have any advantage over the INT8 and INT16 data types. Both INT8 and INT16 have very similar patterns of computational resource usage and throughput, however INT8 spends slight less BRAM. INT4 may also be a good choice if the application does not require the very best accuracy, having a highest top-5 accuracy of 80.3%, with higher throughput and lower usage of BRAM and computational resources than the INT8 and INT16 data types. This results shows the big opportunity that quantization brings by reducing resources spent and improving the throughput without damaging the accuracy of the model.

In the work of [31] the quantization strategy was symmetrically used for activations and weights, using in each implementation the same quantization for all data. Other studies have exploited different quantizations for activations and weights and also different quantizations along the depth of the DNN model. Table 3.2 shows the results of one study [32], of different quantization strategies using as DNN model AlexNet.

The results from table 3.2 show that FP32 only attain 1.3% more accuracy comparing to the quantization using only 8 bitwidth activations and weights, which might not make up for the higher computational complexity of its implementation.

The loss of accuracy between the quantization strategy 8-8218 and the quantization strategy 8-8118 is of 1.5%, being x-xxxx respectively the weights bitwidth of: activations, first convolutional layer, middle convolutional layers, middle fully-connected layers and last fully-connected layer. Comparing that loss of accuracy with the loss of accuracy of 3.3% when the activation bithwidth is reduced rather than decreasing the middle convolutional layers bitwidth, is noticeable the big impact that the reduction in the bitwidth of the activation values have in the accuracy comparing to the reduction of bitwidth of the weights.

Chapter 4

RetinaNet embedded software implementation

The first step in this work is to train an object detection DNN. RetinaNet-18, was the chosen deep neural network for this work. In section 4.1 the training framework used is explained, showing the training results for different quantization in subsection 4.1.2. A quantization model is chosen in subsection 4.1.2. To implement the RetinaNet in an embedded environment, a program in C is developed and initially implemented in a GNU/Linux platform. The workflow used for the program development, a brief description of the code and results of the baseline C implementation are shown in section 4.2. The developed C program is then adapted to the embedded environment and implemented in section 4.3.

4.1 Quantization aware training framework

A quantization aware training is used to quantize data in this work in order to reduce as much as possible the accuracy loss in the quantization process.

The forward pass of the training can still be implemented using 32 bit floating point operations, however before each mathematical operation, the activations, weights and bias are quantized in the chosen bit width as seen in figure 4.1. In this way the quantization error is introduced and propagated along the network being included in the output of the DNN. This forces the backward propagation algorithm to keep the quantization error as low as possible by optimizing the weights taking that into account. Figure 4.1 also shows that the quantization is done to the output in order to take the quantization of the last operation into account.

To quantize data the input data is divided by a scale factor and rounded to the nearest integer. The rounded value saturate if it is higher than the maximum value possible to represent with that quantization type (MAXIMUM_VALUE) or if it is lower than the minimum value possible to represent with that quantization type (MINIMUM _VALUE). This value is then multiplied by the same scale factor used in the beginning. The quantization algorithm is shown in figure 4.2. The round function and the saturation on the algorithm showed in figure 4.2 are responsible for creating the quantization error.

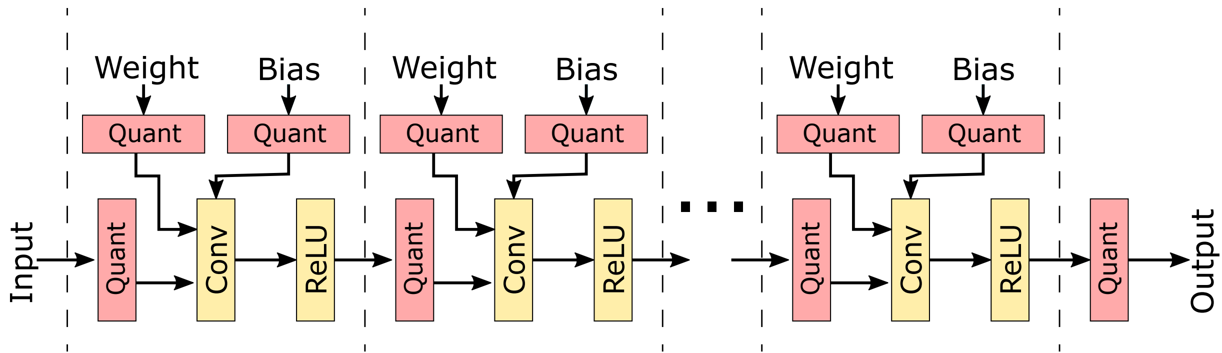


Figure 4.1: Representation of a DNN architecture prepared for quantization aware training.

```
1  output_int = round(input_float/scale);
2
3  if(output_int>MAXIMUM_VALUE){
4    output_int=MAXIMUM_VALUE;
5  }else if(ouput_int<MINIMUM_VALUE){
6    output_int=MINIMUM_VALUE;
7  }
8
9  output_float = output_int * scale;
```

Figure 4.2: Algorithm that introduces the quantization error in training, making it a quantization aware training.

The scale factor relates to the number of fractional bits in a fixed point representation in the following way: $scale = 2^{-factionalBits}$. The number of fractional bits used in a fixed point representation are chosen in way to make the right balance between the resolution and the dynamic range for the magnitude of values that would be represented. The scale factor is the same for all the weights of a given layer, but can be different from layer to layer, the same happens to bias and activations values. A single layer can have three different values of scale factor, one for the weights, one for the bias and one for the activations. The quantization aware framework used in this work is able to choose scale factors automatically during training.

4.1.1 Brevitas

This work uses Brevitas python library to make the quantization aware training [33]. This library extends the PyTorch library giving quantization capabilities to neural network layers. Brevitas implements quantization aware training using the technique mentioned earlier, all operations are made using floating point 32 bit, but the data is quantized in order to add the quantization error.

The implementation of RetinaNet in python using PyTorch developed by Yann Henon [34] was used as baseline for this work. Yann Henon implementation was not prepared for quantization so in this work the implementation was adapted to use Brevitas libraries, so that it can be used for quantization aware training.

In PyTorch each neural network layer is an object, to declare a convolution layer the parameters like number of input channels, number of output channels, kernel size and padding are passed as arguments

to the object constructor as shown in figure 4.3.

```
1  self.conv1 = nn.Conv2d(input_channels=256,
2                         output_channels=256,
3                         kernel_size=3,
4                         padding=1)
5
6  self.act1 = nn.ReLU()
```

Figure 4.3: Convolution and ReLU layer definition in PyTorch.

To use Brevitas, the declaration of the neural network layers is made using the constructor provided by the Brevitas library that extends the original layers class with a new class compatible with quantization aware training. The new convolution layer constructor includes additional parameters related to quantization as shown in figure 4.4:

- *weight_quant_type* - Used to specify if the weights should be quantized and if so what type of quantization should be used. These are the possible options:
  - *QuantType.FP* - No quantization is done to weights;
  - *QuantType.BINARY* - Weights are quantized as binary weights, having only one bit to represent each weight;
  - *QuantType.TERNARY* - Weights are quantized as ternary weights, each weight being represented by three different symbols (+1,0,-1);
  - *QuantType.INT* - Weights are represented using the bit width specified in the argument *weight_bit_width*.
- *weight_bit_width* - Specifies the bit width of the number that represents the weights;
- *weight_restrict_scaling_type* - The scale factor in Brevitas is automatically chosen during training. This argument enables the restriction of quantization scale factor to be a power of two, enforcing it to respect fixed point representation.

The activations are quantized in the ReLU layer, after the convolution. In this way the quantization error produced by the convolution layer can be accounted, and the output activations are ready to be fed to the next layer. This is achieved passing to the constructor of the ReLU layer, arguments similar to the ones used to define the weights quantization strategy in the convolution layer, as shown in figure 4.4.

This work did not quantize bias and batch normalization weights during training, since at the time of this work Brevitas was still in alpha stage and this feature was not fully supported. All the bias were quantized after training before being used in the hardware implementation, so that all values are represented as fixed point numbers.

4.1.2 Quantization results

Different quantizations were evaluated by changing the bit width of weights in different parts of the DNN and comparing its impact on the overall accuracy of the network. The terminology used to refer to each

```
1  self.conv1 = qnn.QuantConv2d(input_channels=256,
2                               output_channels=256,
3                               kernel_size=3,
4                               padding=1,
5                               weight_quant_type=QuantType.INT,
6                               weight_bit_width=4,
7                               weight_restrict_scaling_type=
8  RestrictValueType.POWER_OF_TWO)
9
10 self.act1 = qnn.QuantReLU(max_val=6,
11                           quant_type=QuantType.INT,
12                           bit_width=8,
13                           restrict_scaling_type=RestrictValueType.
                                 POWER_OF_TWO)
```

Figure 4.4: Convolution and ReLU layer declaration using Brevitas library.

Quantization model	mAP (%)	Size of weights (Mb)	Normalized accuracy (%)	Normalized size of weights (%)
RetinaNet_FP	25.7	685	100.0	100.0
RetinaNet_8_8_8	25.8	171	100.4	25.0
RetinaNet_8_4_8	25.7	156	100.0	22.8
RetinaNet_8_8_4	25.6	145	99.6	21.2
RetinaNet_8_4_4	24.9	130	96.9	19.0
RetinaNet_4_8_8	23.7	126	92.2	18.5
RetinaNet_4_4_8	23.4	111	91.1	16.3
RetinaNet_4_8_4	23.6	101	91.8	14.7
RetinaNet_4_4_4	23.4	86	91.1	12.5

Table 4.1: Comparison of trade-offs between different quantization models.

one of the quantized models is: RetinaNet_*b1_b2_b3* where:

- *b1* is the bit width of weights used in the ResNet backbone of RetinaNet;
- *b2* is the bit width of weights used in the FPN part of RetinaNet;
- *b3* is the bit width of weights used in the Regression and Classification sub-networks of Retinanet.

All activations in all layers are quantized using 8 bits.

The last layers of each stage of the regression and classification sub-networks were found very sensitive to quantization and could not be successfully quantized. These are the only convolution layers not quantized in this work.

Figure 4.5 compares 8 different quantized models in respect to their accuracy over an increasing training period. The result of RetinaNet without any quantization is referenced as RetinaNet_FP.

The results in figure 4.5 and table 4.1 show that the RetinaNet that uses 8 bits in all weights (RetinaNet_8_8_8) is able to achieve the same mAP as the non-quantized RetinaNet FP, which demonstrates the success of quantization aware training and indicates that there is no need to try a higher bit width quantization than 8 bits.

From the graph can be seen two distinctive groups in the evolution of accuracy over training time. The quantizations that use 8 bits for the ResNet backbone weights (RetinaNet_8_8_8, RetinaNet_8_4_8,

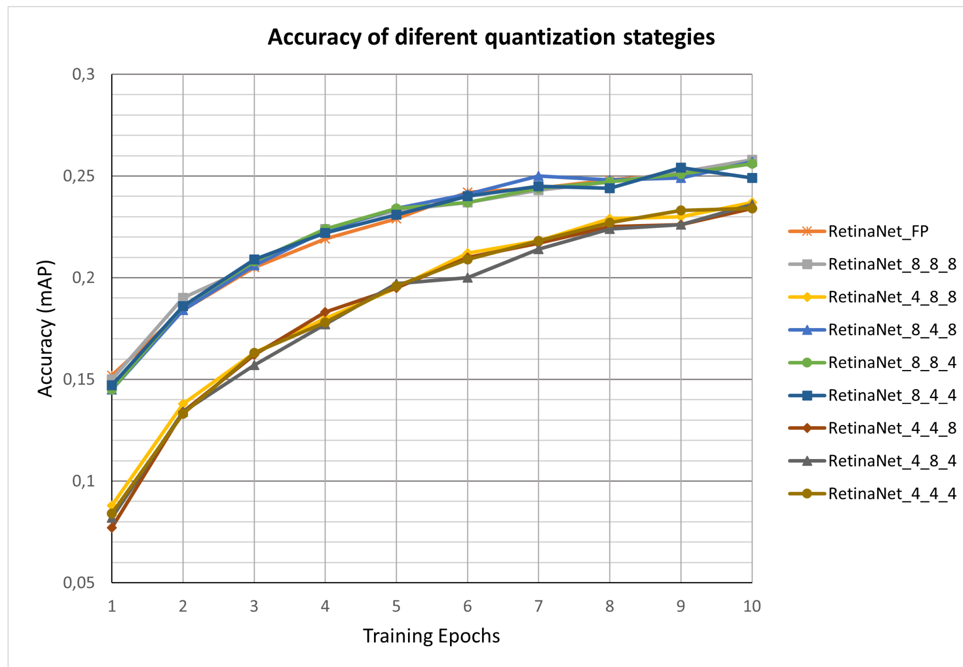


Figure 4.5: Accuracy of different quantization models along increasingly trained networks.

RetinaNet_8_8_4 and RetinaNet_8_4_4) produce a group that achieve higher accuracy than the quantizations that use only 4 bits for the same weights (RetinaNet_4_8_8, RetinaNet_4_4_8, RetinaNet_4_8_4 and RetinaNet_4_4_4). However, the accuracy differences between the two groups decreases along the training epochs. In training epoch 10 the difference in normalized accuracy between quantizations that use 8 bits and 4 bits for:

- ResNet backbone is an average of 7.7 %;
- FPN is an average of 1.2 %;
- Regression and Classification sub-networks is an average of 1.1 %;

Table 4.1 shows a more significant reduction in memory usage than in accuracy when reducing the bit width of weights. The quantization with the biggest reductions in accuracy correspond to the ones that use less memory space. For example the quantization of ResNet backbone with 4 bits weights has the biggest drop in accuracy but at the same time introduces the biggest reduction in the size of weights, since it is accounted with 52 % of the total convolution weights in RetinaNet-18 as seen on figure 2.16. The weights size is reduced by half when using 4 bits weights across the whole network instead of 8 bits weights.

The chosen quantization for the following work was RetinaNet_4_4_4. The 8.9 % drop in accuracy of this quantization in comparison with the original RetinaNet without quantization does not compromise the practical results, producing as output similar bounded boxes and classifications. The accuracy of 23.4 % of this quantization model exceeds the accuracy of YOLO v2 floating-point implementation that has a similar size backbone as shown in table 2.2. The reduction in weight size enables less resource utilization and the possibility to choose a low cost target device.

4.2 Baseline implementation of RetinaNet in C

At first, a version without quantization was implemented. After making sure the implementation in C had the same results of the python implementation, a version of the C implementation of DNN using the ResNet_4_4_4 quantization was developed. The developed program *retinanet.out* was made for inferences only, so the bias and weights must be loaded from an already trained RetinaNet model.

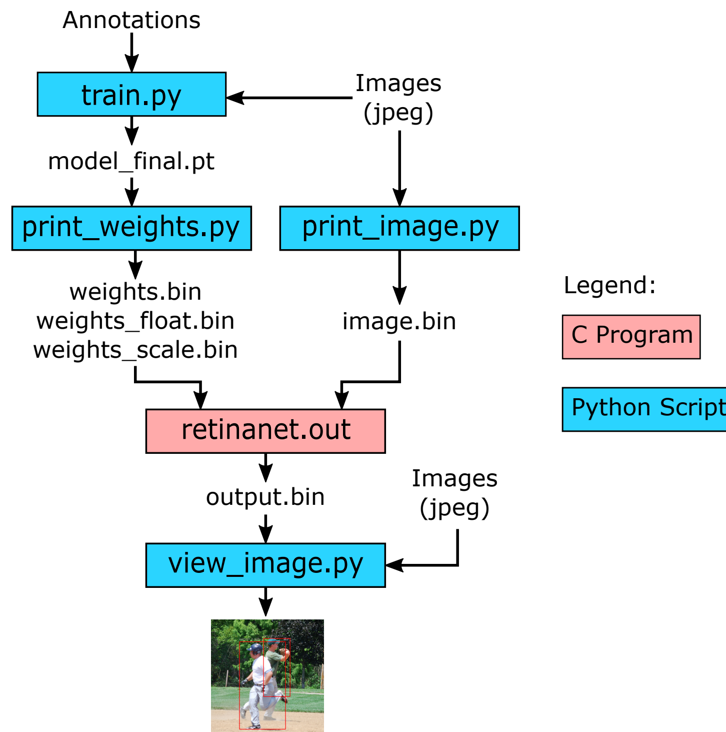


Figure 4.6: RetinaNet usage workflow.

Figure 4.6 shows the workflow used to make an inference using the C program. The training script produces a file called *model_final.pt* that stores all the trained model parameters including the weights and bias values. The *model_final.pt* is parsed by the python script *print_weights.py*, whose purpose is to produce three binary files:

- *weights.bin* that stores all the quantized weights in 4 bit fixed point format and all quantized bias in 8 bit fixed point format;
- *weights_float.bin* that stores all the weights and bias that are not quantized, such as the ones used in the last convolution layers of each stage of the classification and regression sub-networks;
- *weights_scale.bin* that stores the scale of each set of quantized weights and bias, so that is known the fixed point format used in each set.

Since the bias are not quantized using quantization aware training they are quantized during the conversion to fixed point in *print_weights.py* script.

The image to be subjected to inference has to be decoded from the jpeg format and normalized first, this is made using the python script *print_image.py*. The output product of this script is *image.bin*, this

is a binary file containing, each image pixel value as three floating point numbers one for each primary color.

The binary files are then used as input of the C program. The RetinaNet C program outputs a binary file (*output.bin*) with the results from the regression and classification sub-networks, that are the final part of RetinaNet, concatenated with each other. In this way the binary files are used as interface between the C program and the python scripts. The *view_image.py* script interprets the *output.bin* to produce a visual proof of the object detection by overlaying the bonding boxes with labels stating the correspondent classification over the original jpeg image.

4.2.1 Convolution layer implementation

In this implementation of convolution layers the operations use 32 bits floating point values and when needed the values are quantized after the operations, as explained in section 4.1. Three different functions implementing convolution layers were made:

- *conv_quant()* — This implementation uses 8 bits quantized values for activations, weighs and bias. The output is quantized before returning;
- *conv_float()* — This implementation does not use quantized values: all activations, weighs and bias are represented using floating point;
- *conv()* — This implementation is a hybrid, the weights are 8 bits quantized values, but all the activations and bias are represented using floating point and the output value is not quantized.

The *conv_quant()* implementation is the one that is normally used throughout the quantized RetinaNet implementations in C. However in specific situations different implementation are used, for instance the *conv_float()* is used for the layers that were not quantized and the *conv()* is used when the convolution layer is followed by a batch normalization layer and a ReLU layer. In the latter case, the hybrid implementation is adopted to reduce the number of type conversions to be made, since the batch normalization layers are not quantized. The batch normalization is followed by a ReLU layer that quantizes the result.

Figure 4.7 shows the quantized convolution layer definition, the following arguments are used:

- C, A, B, bias — Are pointers to the memory regions containing the output activations, input activations, weighs and bias respectively;
- C_x, C_y — Are the x and y dimensions of the output matrix respectively;
- A_x, A_y, B_x, B_y — Are the x and y dimensions of the input activations and weights respectively;
- N_in, N_out — Are the number of input and output channels;
- stride, padding — Are the stride and padding values;
- output_scale, input_scale, weight_scale, bias_scale - Are the scale factors used in the quantization of output activations, input activations, weights and bias respectively.

```c
1  void conv_quant(int8_t* C,
2                  int8_t* A,
3                  int8_t* B,
4                  int8_t* bias,
5                  int* C_x,
6                  int* C_y,
7                  int A_x,
8                  int A_y,
9                  int B_x,
10                 int B_y,
11                 int N_in,
12                 int N_out,
13                 int stride,
14                 int padding,
15                 float output_scale,
16                 float input_scale,
17                 float weight_scale,
18                 float bias_scale){
```

Figure 4.7: Quantized convolution layer function definition in C.

Figure 4.8 shows the *conv_quant()* implementation. A set of nested loops are used to implement the convolution as in the algorithm shown in figure 3.1. The 3 outer loops in lines 1, 2 and 3 iterate over the output activation matrix, while the 3 inner loops in lines 6, 7 and 8 iterate over the weight filter. The variable *c* indexes the position of the output matrix, being incremented each time a new pixel is calculated, in line 4, this is every time the iteration variables of the 3 outer loops change values. The variable *val* is used as accumulator for the pixel calculation, and its value is reset, in line 5, this is each time a new pixel is indexed.

The if statement between line 10 and 14 selects if the input activation value being iterated corresponds to a padding value, or is an activation from the input activation matrix. The input activation matrix is multiplied by its scale factor before being multiplied with the weight and added to the accumulator as shown in line 13. The weight is also multiplied by its scale factor before being multiplied in line 17.

Since weight values are stored in pairs in int8 variables, they need to be indexed using the function *nibble()* shown in figure 4.9. This function selects between the 4 most or least significant bits of the 8 bit value in the input argument *value*. To access the most significant bits the value is shifted right 4 bits, on the other hand to access the least significant bits the value is shifted left 4 bits and shifted again to the original position in order to cut the 4 most significant bits and deal with the signal bit correctly.

After the 3 inner loops between line 6 to 20 of figure 4.8 finish executing the bias value is added to the accumulator in line 22. The bias value is quantized in a fixed point format, so it is multiplied by its scaling factor to be converted in the floating point representation. The output pixel value is now ready to be quantized between lines 23 and 29. The quantization is made dividing the pixel value by its scale factor, and then saturating its output to the dynamic range of an 8 bit variable. Finally, the value is rounded and stored in the output activation matrix in line 31.

```c
for(int n=0; n<N_out; n++){
    for(int x=0; x+B_x<=A_x+2*padding; x+=stride){
        for(int y=0; y+B_y<=A_y+2*padding; y+=stride){
            c++;
            val=0;
            for(int z=0; z<N_in; z++){
                for(int i=0; i<B_x; i++){
                    for(int j=0; j<B_y; j++){

                        if(x+i<padding || x+i>=A_x+padding || y+j<padding
                            || y+j>=A_y+padding){
                            a=0;
                        }else{
                            a=((float)A[z*A_x*A_y+(x+i-padding)*A_y+y+j-
                                padding])*input_scale;
                        }

                        indice = n*N_in*B_x*B_y+z*B_y*B_x+i*B_y+j;
                        val += a * ((float)nibble(B[indice/2],indice%2))*
                            weight_scale;
                    }
                }
            }
            if(bias!=NULL)
                val += ((float)bias[n])*bias_scale;
            out=val/output_scale;

            if(out>127){
                out=127;
            }else if(out<-127){
                out=-127;
            }

            C[c] = (int8_t)(round(out));
        }
    }
}
```

Figure 4.8: Quantized convolution layer implementation in C.

```c
int8_t nibble(int8_t value, int hi){
    if(hi){
        value = value>>4;
    }else{
        value = value<<4;
        value = value>>4;
    }
    return value;
}
```

Figure 4.9: 4 bits reading function implementation in C.

4.2.2 Overall implementation of RetinaNet

The RetinaNet was first implemented to run on a PC, in a GNU/Linux environment. Functions were created for convolution, batch normalization, ReLU, maxpool, sigmoid, upsample and matrix sum layers. Big chunks of static allocated memory are used for weight, activations and bias as shown in figure 4.10. This big chunks of data emulate the self-managed memory space available in an embedded device. Each array of memory has its propose:

- *memory_weights* — Stores the 4 bits weights and 8 bits bias of all convolutional layers;
- *memory_weights_float* — Stores the weights from the convolutional layer that were not quantized and all the weights from batch normalization layers;
- *memory_image* — Stores the input activations of RetinaNet in floating point format;
- *memory_data* — Stores activations produced in between layers in floating point format;
- *memory_scale* — Stores the scales factors of all sets of quantized values in RetinaNet;
- *memory_data_int* — Stores activations produced in between layers in 8 bits quantized values.

```
1 static int8_t memory_weights[WEIGHTS_TOTAL];
2 static float memory_weights_float[WEIGHTS_FLOAT_TOTAL];
3 static float memory_image[IMAGE_TOTAL];
4 static float memory_data[90000000];
5 static float memory_scale[SCALE_TOTAL];
6 static int8_t memory_data_int[20000000];
```

Figure 4.10: Static allocation of memory for the C implementation of RetinaNet.

Pointers to memory arrays are created to show where the weights, bias and activations for a specific layer are located. The function *define_memory_regions()* in figure 4.11 assign memory addresses for each set of weights, bias and activations. Those addresses are calculated based on the memory used by values stored for the previous layers, as shown by the incremental offset in lines 10, 13, 18 and 21. Each weight only accounts for half byte since they are 4 bits values stored in pairs in 8 bits data type. The weights from different layers are stored contiguous because they are read all at once from the input file.

Figure 4.12 shows the main function of RetinaNet implementation in C. The program starts by reading the files containing the weights, bias, scale factors and the input activation matrix. Then the *define_memory_regions()* function is executed in line 5 in order to create pointers for each matrix of weights, bias and activations. Those pointers are used as arguments of the functions that implement each layer as shown between line 9 and 12. The DNN layers are executed sequentially, and their output are stored in memory regions that are used as input in the next layer, as shown by the *data* pointer that is used as output in the convolution layer in line 9 and then used as input in the batch normalization layer in line 10.

The *regression()* and *classification()* functions between line 17 to 24 and line 28 to 35 respectively, are used to implement a stage of the regression and classification sub-networks. The output of each

```
1  void define_memory_regions(){
2
3      int offset = 0;
4      int offset_float = 0;
5
6      //***ResNet-18***
7      //Stage0
8      //Conv1 Weight 64*3*7*7=9408
9      conv1_weight = memory_weights+offset;
10     offset+=4704;
11     //Batchnorm Weight+Bias+Mean+Var 4*64=256
12     batchnorm_weight = memory_weights_float+offset_float;
13     offset_float+=256;
14
15     //Stage1
16     //Stage1 Block1 Conv1 Weight 64*64*3*3=36864
17     l1_b1_conv1_weight = memory_weights+offset;
18     offset+=18432;
19     //Stage1 Block1 Batchnorm1 Weight+Bias+Mean+Var 4*64=256
20     l1_b1_bn1_weight = memory_weights_float+offset_float;
21     offset_float+=256;
22
23     //....
24 }
```

Figure 4.11: Function defining the memory arrangement of weights, bias and activations in the memory arrays.

stage is arranged as 2 dimensional matrix. *out_x* and *out_y* in line 17 are the dimension x and y of the output matrix. The output of each stage is concatenated with the output of the previous stage using the pointers *out* and *out2* as shown in line 18 where the output activations of that stage are placed in the memory region contiguous to the last activation value of the last stage. In line 19 the x dimension of the output activation matrix of the last stage is summed to the *out_x* variable in order to offset the pointer, that assigns the memory region where the activations of the next stage are stored.

In the end of the program between line 39 and 42 the activations from the regression and convolution sub-networks are written to the *output.bin* file. Since the input image of the RetinaNet has a variable dimension, the output dimension is also variable, that is why the first value written to file, in line 39, is the x dimension of the output activation matrices.

4.2.3 Time profiling results for baseline RetinaNet

The C implementation of RetinaNet was run on a PC using an Intel I7-5700HQ CPU, in order to make a time profiling of the DNN, surveying which part has the most impact in the execution time of the DNN. The time profiling results, in table 4.2, show that regression and classification sub-networks account for 27.3% and 48.9% of the DNN execution time respectively. The stage 2 of regression (Regression S2) is responsible for 77.2% of the execution time of the regression sub-network and similarly, the stage 2 of classification (Classification S2) is responsible for 75.6% of the execution time of the classification subnetwork.

Stage	Intel I7-5700HQ		
	Execution time		Partial execution time
	s	%	%
ResNet S0	10.4	-	6.0
ResNet S1	44.1	-	25.4
ResNet S2	39.3	-	22.7
ResNet S3	40.4	-	23.3
ResNet S4	39.5	-	22.7
Total ResNet	**173.7**	**16.6**	**100.0**
FPN S5	1.4	-	1.9
FPN S6	0.2	-	0.2
FPN S4	3.5	-	4.7
FPN S3	13.6	-	18.1
FPN S2	56.4	-	75.1
Total FPN	**75.1**	**7.2**	**100.0**
Regression S2	219.9	-	77.2
Regression S3	50.0	-	17.5
Regression S4	11.6	-	4.1
Regression S5	2.8	-	1.0
Regression S6	0.7	-	0.2
Total Regression	**285.0**	**27.3**	**100.0**
Classification S2	386.3	-	75.6
Classification S3	97.0	-	19.0
Classification S4	21.6	-	4.2
Classification S5	5.1	-	1.0
Classification S6	1.3	-	0.3
Total Classification	**511.3**	**48.9**	**100.0**
Total	**1045.0**	**100.0**	**-**

Table 4.2: Time profiling of RetinaNet DNN.

```c
int main(int argc, char **argv){

    //Read weighs.bin, weights_float.bin, weights_scale.bin and image.bin files

    define_memory_regions();

    // ***ResNet-18***
    //Stage 0
    conv(data,memory_image,conv1_weight,NULL,&data_x,&data_y,image_y,image_z,7,7,3,64,2,3,memory_scale[0]);
    batchnorm(data,data,batchnorm_weight,&data_x,&data_y,data_x,data_y,64,0.00001);
    relu(data,data,&data_x,&data_y,data_x,data_y,64,0.03125);
    maxpool(res,data,&res_x,&res_y,data_x,data_y,3,3,64,2,1);

    //...

    // ***Regression***
    regression(out,res,&out_x,&out_y,res_x,res_y,0.03125,0.25);
    regression(out+(out_x*out_y),res2,&aux_x,&aux_y,res2_x,res2_y,0.03125,0.25);
    out_x+=aux_x;
    regression(out+(out_x*out_y),res3,&aux_x,&aux_y,res3_x,res3_y,0.03125,0.125);
    out_x+=aux_x;
    regression(out+(out_x*out_y),res4,&aux_x,&aux_y,res4_x,res4_y,0.03125,0.125);
    out_x+=aux_x;
    regression(out+(out_x*out_y),res5,&aux_x,&aux_y,res5_x,res5_y,0.03125,0.125);
    out_x+=aux_x;

    // ***Classification***
    classification(out2,res,&out2_x,&out2_y,res_x,res_y,0.001953125,0.25);
    classification(out2+(out2_x*out2_y),res2,&aux_x,&aux_y,res2_x,res2_y,0.001953125,0.25);
    out2_x+=aux_x;
    classification(out2+(out2_x*out2_y),res3,&aux_x,&aux_y,res3_x,res3_y,0.001953125,0.125);
    out2_x+=aux_x;
    classification(out2+(out2_x*out2_y),res4,&aux_x,&aux_y,res4_x,res4_y,0.001953125,0.125);
    out2_x+=aux_x;
    classification(out2+(out2_x*out2_y),res5,&aux_x,&aux_y,res5_x,res5_y,0.001953125,0.125);
    out2_x+=aux_x;

    //write output.bin file
    fwrite(&out_x, 4, 1, fout);
    fwrite(out, 4, out_x*out_y, fout);
    fwrite(out2, 4, out2_x*out2_y, fout);
    fclose(fout);
}
```

Figure 4.12: Main function of quantized RetinaNet implementation in C.

Region name	Size (bytes)	Base address	End address
mem_weights	2,361,344	0x00100000	0x003407FF
mem_weights_float	6,970,320	0x00340800	0x009E63CF
mem_scale	64	0x009E63D0	0x009E640F
mem_data	80,000,000	0x01000000	0x05C4B3FF
mem_data_int	20,000,000	0x05C4B400	0x06F5E0FF
mem_input	200,000,000	0x07000000	0x12EBC1FF
mem_output	80,000,000	0x13000000	0x17C4B3FF
Total	389,331,728		

Table 4.3: Memory map of the DDR embedded system memory.

Stage 2 of classification and regression have as input an activation matrix with 80×80×256 in dimension, this is four times more activations than stage 3. While the stage number increases, the number of activations decrease four times in comparison with the previous stage as seen in appendix A. Since the execution time of a given stage is approximately proportional with the number of input activations it is understandable that Regression S2 and Classification S2 stages, are the ones that takes more time to execute. It can also be seen in table 4.2 that has the stage number of classification and regression increases, the execution time it takes is approximately decreases four times.

In RetinaNet-18 DNN 76.2 % of the execution time is due to the regression and classification sub-networks, this is the primary reason why this is the chosen part to be implemented in hardware. In addition, the repetitive pattern of this part of RetinaNet would also help to reduce the flexibility that the hardware architecture has to accomplish.

4.3 Embedded implementation of RetinaNet in C

The regression and convolution sub-networks are the components to be accelerated in the hardware, and are deployed to the embedded system. For development, only these parts are executed in the embedded platform in order to reduce the overall execution times. In this way the input of the embedded implementation is the output from the FPN.

The main difference between the implementation in the GNU/Linux environment, and the embedded environment, is that the memory is managed by the developer. Table 4.3 shows the memory regions size and their base addresses. The regions are the same as the ones explained in section 4.2.2, a part from two new regions:

- *mem_input* — Stores the input values of the RetinaNet part that is implemented in the embedded device;
- *mem_output* — Stores the output of RetinaNet.

The memory was carefully arranged so that no memory regions are overlaid as seen on table 4.3.

Figure 4.13 shows the changes made to *define_memory_regions()* function so the memory mapping in table 4.3 is implemented. Between lines 12 and 16 the pointers to the memory regions are initialized with their base addresses.

In the embedded environment the memory is initialized with the weights, bias and input activations, so that all the values are already available when the embedded system starts running. In this way no file reading is required. The values between line 22 and 31 are initialized with the dimensions of the input feature maps. Between line 33 and 37 pointers are initialized to the input activation. The pointer to the region where the output of the RetinaNet is stored is initialized with the base address of that region plus one, so that enough space is left for an integer containing the size of the output.

```
1  #define MEM_WEIGHTS 0x100000
2  #define MEM_WEIGHTS_FLOAT 0x340800
3  #define MEM_SCALE 0x9E63D0
4  #define MEM_DATA 0x1000000
5  #define MEM_DATA_INT 0x5C4B400
6
7  #define MEM_INPUT 0x7000000
8  #define MEM_OUTPUT 0x13000000
9
10 void define_memory_regions(){
11
12     memory_weights = (int8_t*) MEM_WEIGHTS;
13     memory_weights_float = (float*) MEM_WEIGHTS_FLOAT;
14     memory_scale = (float*) MEM_SCALE;
15     memory_data = (float*) MEM_DATA;
16     memory_data_int = (int8_t*) MEM_DATA_INT;
17
18     int * aux_pointer;
19
20     aux_pointer=(int*)MEM_INPUT;
21
22     res_x=aux_pointer[0];
23     res_y=aux_pointer[1];
24     res2_x=aux_pointer[2];
25     res2_y=aux_pointer[3];
26     res3_x=aux_pointer[4];
27     res3_y=aux_pointer[5];
28     res4_x=aux_pointer[6];
29     res4_y=aux_pointer[7];
30     res5_x=aux_pointer[8];
31     res5_y=aux_pointer[9];
32
33     res=(float*)MEM_INPUT+10;
34     res2=&res[256*res_x*res_y];
35     res3=&res2[256*res2_x*res2_y];
36     res4=&res3[256*res3_x*res3_y];
37     res5=&res4[256*res4_x*res4_y];
38
39     out=(float*)MEM_OUTPUT+1;
40
41     //...
42 }
```

Figure 4.13: Function defining the memory mapping in the embedded system implementation.

Stage	Xilinx Zynq-7020 SW only (ARM Cortex A9)		
	Execution time		Partial execution time
	s	%	%
Regression S2	**764**	-	31.7
Regression S3	**157**	-	6.5
Regression S4	**39**	-	1.6
Regression S5	**9**	-	0.4
Regression S6	**2**	-	0.1
Total Regression	**971**	**40.3**	**100.0**
Classification S2	**1,118**	-	46.4
Classification S3	**244**	-	10.1
Classification S4	**58**	-	2.4
Classification S5	**14**	-	0.6
Classification S6	**3**	-	0.1
Total Classification	**1,436**	**59.7**	**100.0**
Total	**2,407**	**100.0**	**-**

Table 4.4: Time profiling of RetinaNet software implementation on embedded device.

4.3.1 Time profiling results for embedded software RetinaNet

Table 4.4 shows the time profiling results of the software implementation of RetinaNet in the embedded device Xilinx Zynq-7020. In total the embedded systems takes about 40 minutes to execute the regression and convolution sub-networks of RetinaNet. In comparison, the implementation made in the GNU/Linux environment takes 13.3 minutes to execute the same part of the DNN as shown in table 4.2. The regression sub-network is responsible for about 40 % of the total execution time, the remaining 60 % of the time being spent with the classification sub-network. Appendix B presents a detailed time profiling of RetinaNet implementation, which shows that a single convolution layer with 80×80×256 dimensions takes about 186 s to execute.

4.4 Conclusions

In this chapter a study of the trade-offs of different quantization was made, and it was chosen to quantize all weights using 4 bits values. This quantization was selected since the accuracy result is better than the result of the object detections DNN with similar size, and there is not a visual perceptible difference from the results of quantizations with bigger bit-widths. Using 4 bits values for the weights reduces in half the memory usage comparing to quantizations with 8 bits, allowing for the use of lower cost devices for the implementation of the DNN.

The results from the baseline implementation of RetinaNet showed that the majority of the execution time was spent in the regression and classification sub-networks, and that the convolution hardware accelerator should be targeted to accelerate the convolution layers belonging to those parts of RetinaNet. The execution time targeted to beat is the 186 s that a single 80×80×256 takes to execute in the embedded software implementation of RetinaNet.

Chapter 5

Convolution accelerator hardware architecture

To accelerate RetinaNet a convolution hardware accelerator is developed, since the convolution layers are responsible for most of the execution time of an object detection inference.

In section 5.1 the architecture is presented in a top-down approach. Some cross-cutting issues over the overall architecture are approached in subsection 5.1.1 and 5.1.2 such as the way data is formatted and the scheduling of operations. In section 5.2, 5.3 and 5.4 a more in depth description is made of the several components of the convolution accelerator. Section 5.5 shows how the components described in the previous section integrate into a final IP solution. The results achieved by the convolution accelerator are presented in section 5.6.

5.1 Architecture Overview

The hardware accelerator was designed specifically to accelerate convolution layers, since these layers take most of the execution time. The hardware accelerator was developed using Xilinx Vivado 2019.2 High Level Synthesis (HLS). This hardware architecture was targeted to be implemented in a FPGA Xilinx Artix-7, this is the programmable logic available in the chosen device for this work, Xilinx Zynq-7020.

The designed hardware architecture was made to be flexible about the x and y dimensions of the input and output feature maps and is also compatible with values quantized with different scale factors. On the other hand it works with a fixed padding of 1, a fixed stride of 1, a fixed kernel size of 3×3 and a fixed output and input channel dimension of 256. These characteristics give enough flexibility for the accelerator to compute all the quantized convolution layers in the classification and regressions sub-networks without overcomplicating the architecture, since all the layers in those sub-networks share the same padding, stride, kernel size and channel dimensions, being different only in the x and y input and output dimensions and using different values with different scale factors.

The connection between the FPGA and the DDR external memory has a high latency compared

to on-chip FPGA memory. Each activation, weight and bias is accessed more than once in a single convolution, in order to reduce the number of times that each of these data need to be individually transferred between the FPGA and the DDR memory, and therefore internal FPGA memory resources were used to cache each one of these types of data. In this way the data is transferred from the external memory in big contiguous chunks of data, making use of data pipeline, reducing the impact of the latency of accessing the external DDR memory. The designed architecture achieved total reuse of data, which means that each weight, bias and activation is transferred only once from the external memory.

Figure 5.1 presents an overall overview of the developed hardware architecture. The main data input of the hardware architecture is an AXI-Stream port, that is connected to activation, weight and bias memories. These three memories are used to feed the processing elements (PE) with the data to be computed. The output of each PE is multiplex, which makes it possible to use only one AXI-Stream output port for all the PE's.

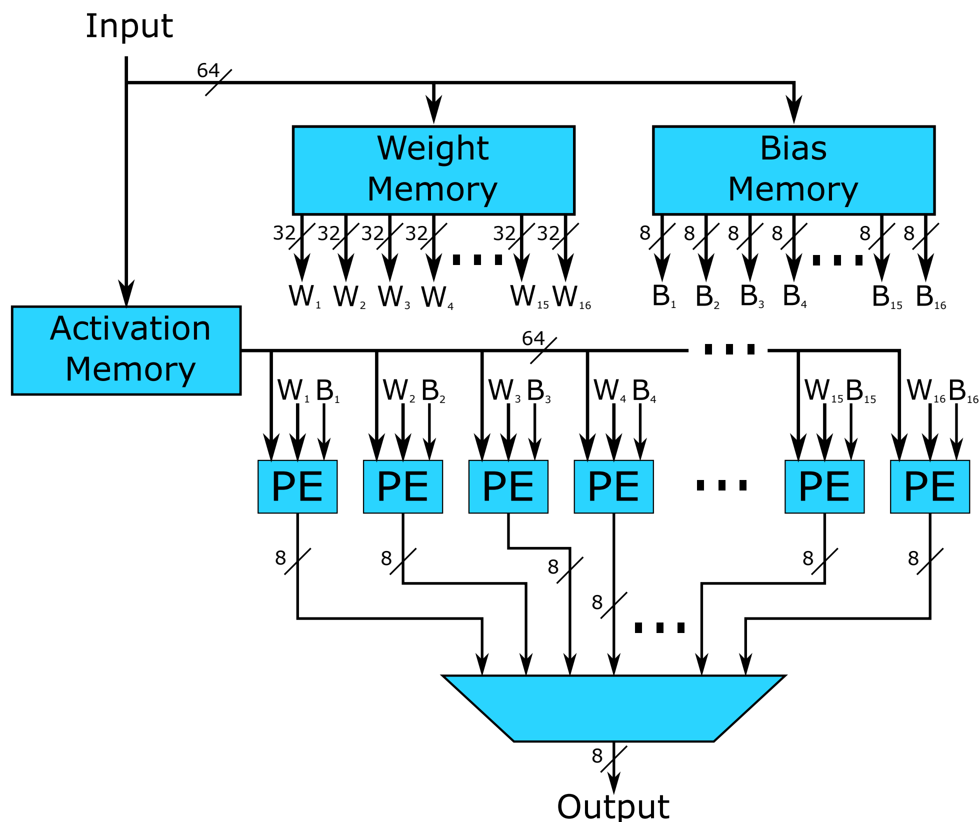


Figure 5.1: General overview of the accelerator hardware architecture.

The interface specification is shown in figure 5.2, the *in* and *out* ports of the *conv3d* IP are AXI-Stream I/O channels (as defined by the first two *HLS INTERFACE* pragmas). The other *conv3d* inputs are implemented using a single AXI-Lite interface, *BUS_A* as defined by the corresponding *HLS INTERFACE* pragmas. AXI-Lite inputs are used for IP configuration, specifying characteristics of the convolution layer to be executed:

- *a_x* - Specifies the x dimension of the input activations;
- *a_y* - Specifies the y dimension of the input activations;

- *output_scale* - Specifies the number of fractional bits wanted in the quantization of the output activations;
- *input_scale* - Specifies the number of fractional bits used in the quantization of the input activations;
- *weight_scale* - Specifies the number of fractional bits used in the quantization of the weights;
- *bias_scale* - Specifies the number of fractional bits used in the quantization of the bias.

```
1  void conv3d(hls::stream< ap_int<64> > &in,
2              hls::stream< ap_axis > &out,
3              int a_x,
4              int a_y,
5              int output_scale,
6              int input_scale,
7              int weight_scale,
8              int bias_scale){
9
10     #pragma HLS INTERFACE axis port=in
11     #pragma HLS INTERFACE axis port=out
12     #pragma HLS INTERFACE s_axilite port=a_x bundle=BUS_A
13     #pragma HLS INTERFACE s_axilite port=a_y bundle=BUS_A
14     #pragma HLS INTERFACE s_axilite port=output_scale bundle=BUS_A
15     #pragma HLS INTERFACE s_axilite port=input_scale bundle=BUS_A
16     #pragma HLS INTERFACE s_axilite port=weight_scale bundle=BUS_A
17     #pragma HLS INTERFACE s_axilite port=bias_scale bundle=BUS_A
```

Figure 5.2: HLS code that implements the interface of the hardware architecture.

5.1.1 Data storage format

The order in which the activations values are stored in both external and FPGA memories, follow ZXY coordinate order. This means that the values are stored first by their channel number (z dimension) and only then through their column number (x dimension) and row number (y dimension). In this way values with the same x and y coordinates but with sucessive z coordinates are consecutive to each other.

Memory resources inside the FPGA are limited, not all the activations values of a given layer can fit simultaneously, so it is better to store only the values that are going to be used right away. Since the values have to be transferred in parts, XYZ storage format avoids accessing the external memory randomly, by sequencing the values more closely to the order they are going to be used. This is true because to compute a single output pixel, all span over the z dimension is needed, however that does not apply with the x and y dimensions.

The weight values follow NZXY coordinate order in the internal FPGA memories, N being the channel number of a filter. The memory is iterated first by the channel number because of the way the weight memory is arranged as explained in section 5.4. The z coordinate is iterated before x and y coordinates in order to follow the format used in activations, since the weights are going to be multiplied by the activations and both values are going to be iterated simultaneously.

The weight values are stored in the DDR memory following ZXYN coordinate order and adapted to the NZXY coordinate order of the local memory, during the transfer process by the designed hardware architecture. All the weights of a given convolution layer are fully copied to the local weight memory. This change in storage format is efficient because the transfer is still made in big chunks, each chunk is processed upon receiving and stored at the right memory address.

5.1.2 Operations Scheduling

Each processing element is scheduled to compute a different output feature map in a given time, this means that every PE is doing a different output activation in the z dimension at the same time. However, all PEs are processing pixels with the same x and y dimensions in a given time as seen in figure 5.3. With this parallelization strategy the same activation values may be broadcast to every PE as seen in figure 5.1, since the same set of input activations are used for output pixels with the same x and y coordinates. The broadcast of activations reduces the bandwidth requirements of the activation memory and the complexity of the architecture.

The weight and bias used by each PE in a given time are different, creating the need for the weight and bias memories to have as many output ports has the number of PE's. The developed architecture uses 16 PEs that is why there are 16 ports for the bias and weight memories.

An output feature map pixel is processed entirely in a single PE, so that partial results don't need to be transferred between PEs. In addition, all the operations of a pixel are made contiguous in time, so that no partial results need to be stored outside the PEs temporarily. In this way a PE is responsible for determining the value of a pixel, and only changes to the next pixel when the final output value of that pixel is determined.

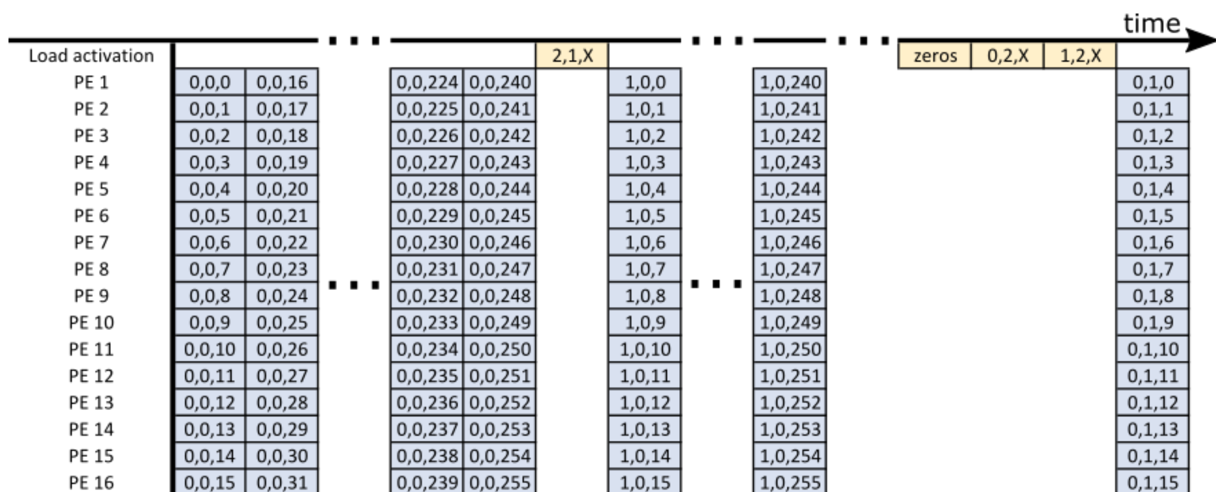


Figure 5.3: Timeline of PE's operations, showing the coordinates of output pixel being determined at given time in a given PE, using the x,y,z notation. The coordinates of the input activations being loaded from the external DDR memory are also shown in yellow, following x,y,z notation where X means that all values in that dimension are being loaded.

Each PE is doing the output pixel with coordinates $C(n) = (x_n = x_{n-1}, y_n = y_{n-1}, z_n = z_{n-1} + 1)$ being n the ID number of the PE, thus $C(n-1) = (x_{n-1}, y_{n-1}, z_{n-1})$ being the coordinates of the previous

PE. The PE's output are selected sequentially to be sent to the DDR memory, sending the values in ZXY format. It is ideal to send the values in this order, in this way they can be stored sequentially, and they will sustain the format wanted to be used as input in the next convolution layer.

5.2 Processing Elements

The processing elements (PE) are composed of 8 multiply and accumulate operators (MAC) that make the multiplications between activations and weights, as described in more detail in subsection 5.2.1. Those 8 MACs are then connected to a sum tree to obtain the final result for the output pixel calculated by the PE, as described in more detail in subsection 5.2.2. The PE output is quantized to a bit width of 8, as described in more detail in subsection 5.2.3. All the components and connections that compose the PE are shown in figure 5.4.

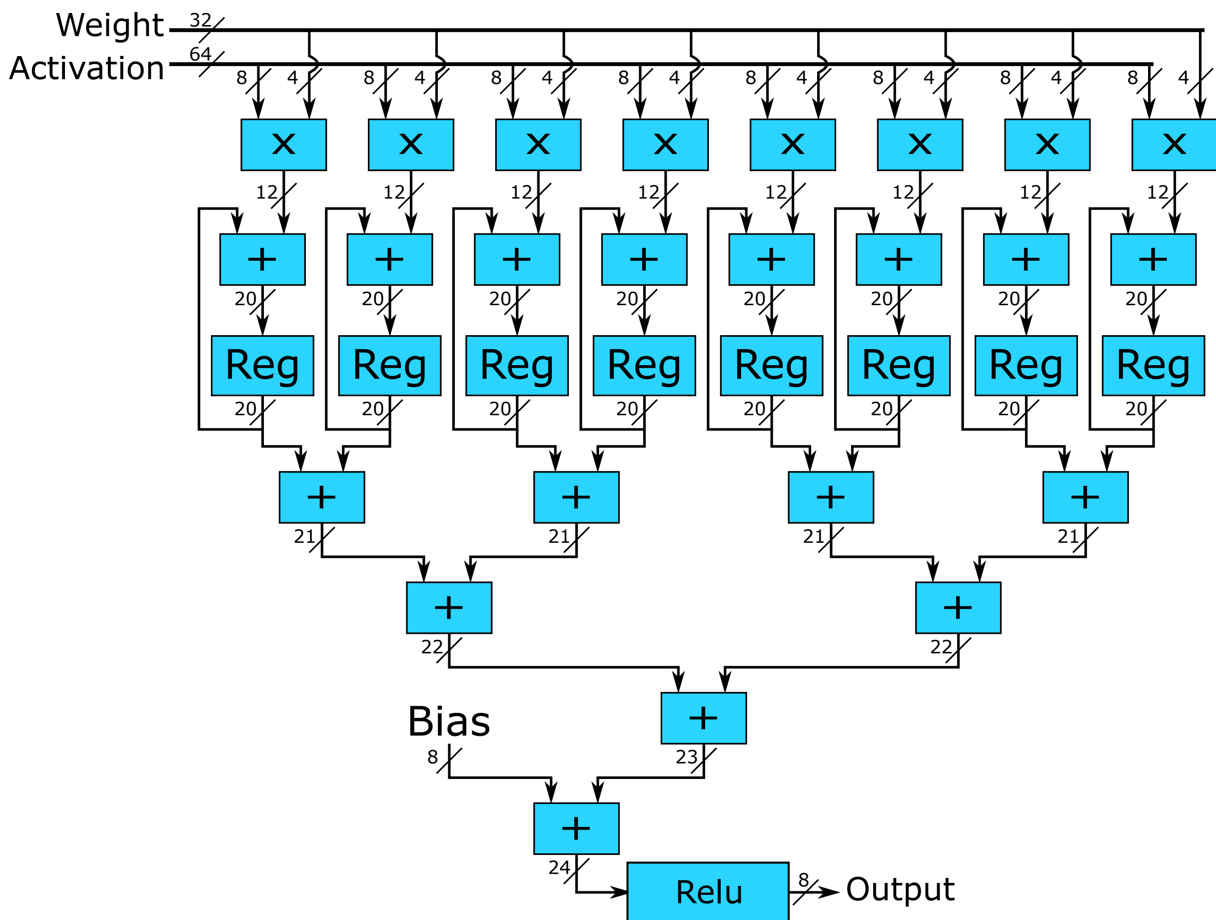


Figure 5.4: PE hardware architecture.

5.2.1 Multiply and accumulate operators

Each output pixel of a convolution is a sum of multiplications of weights with activations as shown in equation 2.3. In this way, multiply and accumulate operator (MAC) is ideal for this application. The MAC

is composed by a multiplier, an adder and a register. The adder adds the result from the multiplier with the result of the register, this arrangement makes the set adder and register work as an accumulator.

The processing element is composed by 8 multiply and accumulate operators, working in a divide and conquer strategy. Each MAC, inside a PE, is responsible for processing different input activation values. The input activation xyz coordinate that each MAC is responsible at any given time is shown in figure 5.5. At the same time, different MAC are processing input activations with the same x and y coordinates, but with different z coordinates.

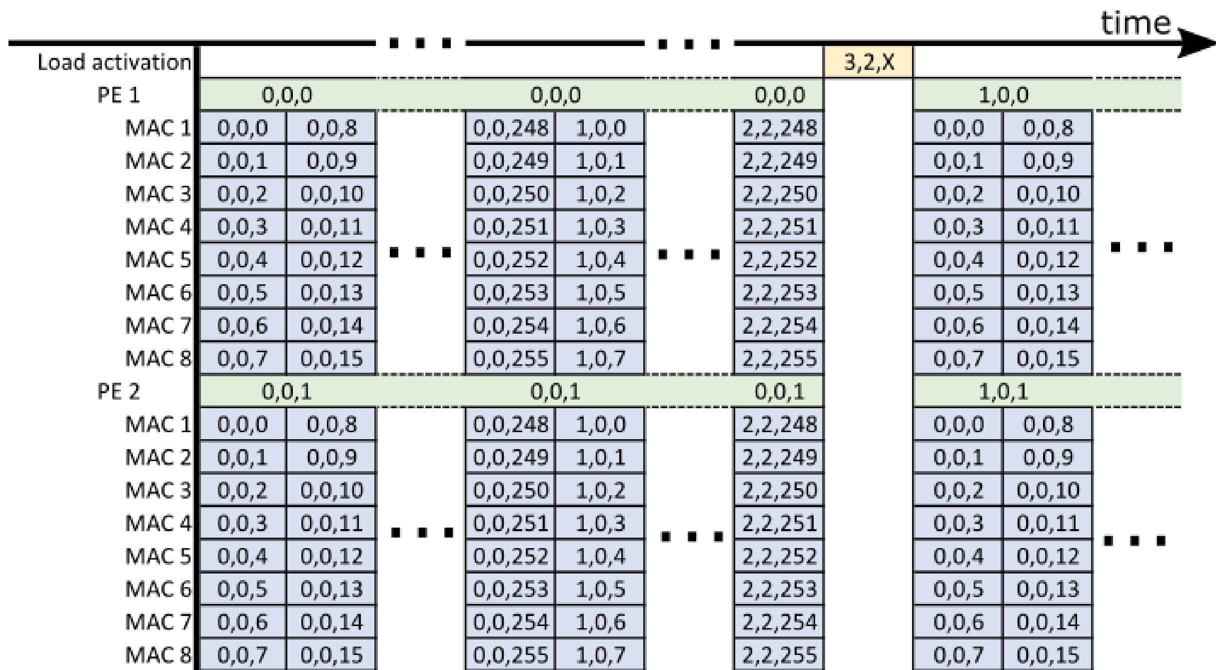


Figure 5.5: Timeline of MAC's operations, showing the coordinates of the input pixel processed in each MAC/PE set at any given time in blue, using x,y,z notation. The output pixel being determined at given time in a given PE, is showed in green using the x,y,z notation. The coordinates of the input activations being loaded from the external DDR memory are also shown in yellow, following x,y,z notation where X means that all values in that dimension are being loaded.

The activation and weight memories are arranged in a ZXY format and since every MAC is processing contiguous values in the z dimension, it is possible to receive a chunk of data from the activation memory containing 8 activations and a chunk of data from the weight memory containing 8 weights. Figure 5.4 shows the input activation bus with a bit width of 64 bits, because it contains 8 activations, these 8 activations are then separated for the different MACs. Similarly, the input weight bus has a bit width of 32 bits, because it contains 8 weights that are also separated for each one of the MACs.

The HLS code used to implement the MACs can be seen in figure 5.6: between line 10 and 17 are implemented the bus splitter for the weight values explained above, and between line 19 and 26 are implemented the bus splitter for the activation values. The *c.range(a,b)* method gets the value contained between *a* and *b* bits in the bus *c*. The values of the weights and activations are then multiplied and summed to the accumulator between lines 28 and 35.

The code in figure 5.6 implements all the PEs using a loop followed by a pragma of *HLS unroll*. The loop instantiates each one of the PEs, assigning a different value of *o* for each PE. The variable *o* is

```
1  for(int o=0; o<PE; o++){
2  #pragma HLS unroll
3      if(o==0)
4          a = activation[l*WEIGHT_X*N_IN8+m*N_IN8+n];
5
6      ap_int<32> w = weight[l*WEIGHT_X+m][n*N_OUT+(k+o)];
7      ap_int<4> aux_w[8];
8      ap_int<8> aux_a[8];
9
10     aux_w[0]=w.range(3,0);
11     aux_w[1]=w.range(7,4);
12     aux_w[2]=w.range(11,8);
13     aux_w[3]=w.range(15,12);
14     aux_w[4]=w.range(19,16);
15     aux_w[5]=w.range(23,20);
16     aux_w[6]=w.range(27,24);
17     aux_w[7]=w.range(31,28);
18
19     aux_a[0]=a.range(7,0);
20     aux_a[1]=a.range(15,8);
21     aux_a[2]=a.range(23,16);
22     aux_a[3]=a.range(31,24);
23     aux_a[4]=a.range(39,32);
24     aux_a[5]=a.range(47,40);
25     aux_a[6]=a.range(55,48);
26     aux_a[7]=a.range(63,56);
27
28     accum[o*8] += aux_w[0]*aux_a[0];
29     accum[o*8+1] += aux_w[1]*aux_a[1];
30     accum[o*8+2] += aux_w[2]*aux_a[2];
31     accum[o*8+3] += aux_w[3]*aux_a[3];
32     accum[o*8+4] += aux_w[4]*aux_a[4];
33     accum[o*8+5] += aux_w[5]*aux_a[5];
34     accum[o*8+6] += aux_w[6]*aux_a[6];
35     accum[o*8+7] += aux_w[7]*aux_a[7];
36 }
```

Figure 5.6: HLS code that implements the MACs in the PEs.

then used along the code to find the right input values for that specific PE and also to use the right accumulator registers, so that each PE use a different set of MACs. The pragma *HLS unroll* allows the loops iterations to occur in parallel, making each one of the iterations to use similar but different hardware resources.

Lines 3 and 4 of the code in figure 5.6 show the activation input *a*, that is only updated in the first PE iteration, because the same value is used for all the PEs as seen in figure 5.4 and described in section 5.1.2. Both the activation input in line 4 and the weight input in line 6 are accessed through indexes calculated using iteration variables that come from external loops of the PE. These iteration variables schedule the whole system operations spanning across the input feature map and filters. When the HLS code is synthesized, the schedule of all operations will be implemented as a control unit, in this way the iteration variables will not be directly mapped to hardware but instead help to defining the control signals and memory addresses used along an execution cycle of the convolution accelerator.

The size of the multiplier, adder and register of the MAC is calculated in order to make all the operations without overflows or any loss of resolution. In this way taking into account that the activation values have a bit width of 8, and that the weight values have a bit width of 4, the multiplier must have a bit width of $8 + 4 = 12$. In the worst case scenario, the MAC would make 288 consecutive operations without resetting the accumulator, because there are $256 * 3 * 3 = 2304$ values in a filter with a kernel size of 3×3 and 256 channels and all those values will be processed in the same PE that in turn has 8 MAC, giving a total of $2304/8 = 288$ operations per MAC. The maximum signed value that the multiplier can output is $-128 * -8 = 1024$. If all the 288 multiplications have a result of 1024 then the value that ends up being accumulated is $1024 * 288 = 294912$. To store this value 19 bits plus a sign bit are needed, this is why both adder and register of the MACs have a bit width of 20 bits.

5.2.2 Sum tree

After all the multiplications between weights and activations are concluded for a given output pixel, all the MAC results are added in a sum tree, as shown in 5.4. The sum tree result is then added with the bias used in that output feature map, this adder result is then used as input in a quantizer.

The implementation of the sum tree is shown in figure 5.7. The first two lines implement, using a loop and a pragma, all the PE's sum trees in the same way that it is done for the MACs. From line 3 to 6 the adders of the sum tree are declared, taking into account the required bit width, so that there is no overflow. Lines 9 to 12 implement the first stage of adders of the sum tree, lines 14 and 15 implement the second stage of adders, and line 17 implements the last stage.

```
1  for(int o=0; o<PE; o++){
2  #pragma HLS unroll
3      ap_int<21> sum1[4]={0,0,0,0};
4      ap_int<22> sum2[2]={0,0};
5      ap_int<23> sum3=0;
6      ap_int<24> sum4=0;
7      ap_int<23> bias_aux;
8
9      sum1[0] = accum[o*8] + accum[o*8+1];
10     sum1[1] = accum[o*8+2] + accum[o*8+3];
11     sum1[2] = accum[o*8+4] + accum[o*8+5];
12     sum1[3] = accum[o*8+6] + accum[o*8+7];
13
14     sum2[0] = sum1[0] + sum1[1];
15     sum2[1] = sum1[2] + sum1[3];
16
17     sum3 = sum2[0] + sum2[1];
18
19     bias_aux = bias[k+o];
20     bias_aux = (ap_int<23>)(bias_aux>>(bias_scale-(weight_scale+input_scale
           )));
21
22     sum4 = sum3 + bias_aux;
23 }
```

Figure 5.7: HLS code that implements the sum tree in the PEs.

The bias is accessed in line 19 according to the output feature map being produced in that specific PE, that is why the bias indexing depends on the PE iteration variable *o*. In conjunction with the variable *o*, an output channel iteration variable *k* is also used to index the bias array, this variable is external to the sum tree. To sum the bias with the result value of the sum tree, the decimal point of the bias is aligned with the sum tree result. The alignment is made using a barrel-shifter, the number of shifted bits and the shifting direction is determined by finding the difference between the number of fractional bits of the bias values and the number of fractional bits of the sum tree result, that in turn is the sum of the fractional bits of the weights and activations fed to this convolution layer. The aligned bias defined by the code in line 20 is then added to the sum tree result in line 22.

5.2.3 Quantizer and ReLU

The quantizer quantizes the lossless result of all the operations made inside the PE to an 8 bit width value. This is done so that the output value occupies less memory and, in accordance with the quantization strategy selected, so that it can be used as input of the next convolution layer.

Figure 3.3 shows the HLS code used to implement the quantizer. Line 1 calculates how many bits the quantizer input value have to be shifted in order to comply with the number of fractional bits wanted for that output. This is determined by subtracting the number of fractional bits of the input (*input_scale+weight_scale*) with the number of factional bits desired for the output (*output_scale*). In line 11 the variable *sum4*, that is holding the input of the quantizer, is shifted the amount of bits calculated in line 1. In line 12 the quantized result is sent to the output using only the 8 least significant bits of the shifted input. The if statement in lines 6 to 13 is responsible for saturating the output value by comparing in line 8 the input value of the quantizer with the maximum possible output value. The maximum output value is calculated in lines 4 and aligned with the input value.

```
1  int lower_bit = input_scale+weight_scale-output_scale;
2  ap_int<24> min=-1, max=1;
3
4  max = (max << (lower_bit+7))-1;
5
6  if(sum4<0){
7      result.data = 0;
8  }else if(sum4>max){
9      result.data = 0x7F;
10 }else{
11     sum4=sum4>>lower_bit;
12     result.data = sum4.range(7,0);
13 }
```

Figure 5.8: HLS code that implements the PE's quantizer.

The ReLU operation is performed by zeroing the output value of the PE if the input value is lower than zero. This is done by the same if statement used for the quantization saturation, in lines 6 and 7.

5.3 Activation memory

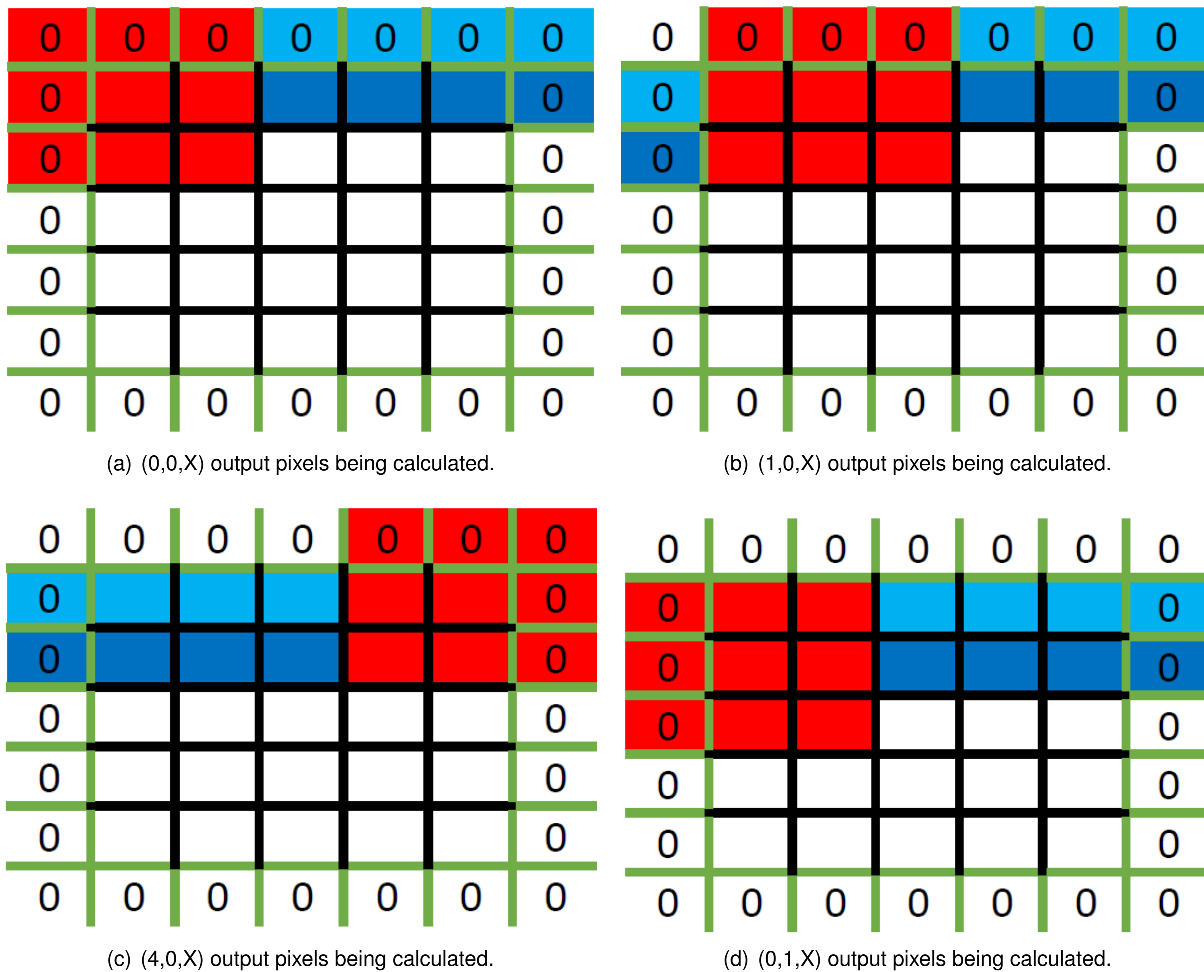
The number of input activations of the largest convolution layer to be processed in the hardware accelerator is $80 * 80 * 256 = 1638400$. Each activation value has a bit width of 8, in this way the biggest input activation matrix occupies 13.1 Mb. The target device Xilinx Zynq-7020 has a total of 4.9 Mb of block RAM, this is not enough to store the whole input activation matrix. The adopted strategy was to store in local memory only enough activations so that each activation is only transferred once from the external memory. It was decided to store all the activations across the z axis together because to determine a single output pixel all the activations across that axis are needed. In that way in figure 5.9 are shown two-dimensional matrices of input activations, a single position on those matrices represent all the activations across the Z axis with that specific x and y coordinates. The matrices positions are colored in:

- red to show which activations are stored in the local memory and ready to be read to determine the output pixel being produced at that given time;
- blue to show which activations are stored in local memory waiting to be used again;
- white to show which activations are not stored in local memory.

Figure 5.9 (a) shows the content of the activation memory when the output pixels with coordinates $x = 3$ and $y = 2$ and across all z axis ($(x, y, z) = (0, 0, X)$) are being determined. After that pixel was determined the input activations with coordinates $(x, y, z) = (3, 2, X)$ are loaded to the local activation memory. The input activations $(x, y, z) = (0, 0, X)$ previously loaded are then erased from the local memory because they are not going to be used again, since the kernel is moving along the x and then y axis, as show in figure 5.9 (b). Figure 5.9 (b) also shows that input activations with coordinates $(x, y, z) = (0, 1, X)$ and $(x, y, z) = (0, 2, X)$ are still in local memory waiting to be used later, for example when the output pixels $(x, y, z) = (0, 1, X)$ are being determined.

The local storage strategy used for the input activations makes it possible to reuse the input activations completely using the minimum space possible. The activation values were stored in the different types of memory shown in figure 5.10. The input activations are stored in shift-registers accordingly to their coordinates if they are part of the calculations of the output pixel being determined at that moment. In case the activation values are in local memory waiting to be used later they are stored in FIFOs. The way the FIFOs and shift-registers are connected with each other, as shown in figure 5.10, is related to the way that activations flow from being actively used in a calculation of an output pixel and are out on hold to be used later. A multiplexer is used to select which one of the shift registers would output its value to be used in the PEs, as shown in figure 5.10.

The shift registers were implemented using Flip-Flops (FF), as for the FIFOs they were implemented using BRAMs. The activations were divided in different types of memories because the implementation of FIFOs in BRAMs does not allow for random access of the activations, which is necessary to feed the PEs. On the other hand, using only FF for all the activations would generate a high usage of FF, that would make the system as an all not able to fit in the Zynq-7020.



(a) (0,0,X) output pixels being calculated.

(b) (1,0,X) output pixels being calculated.

(c) (4,0,X) output pixels being calculated.

(d) (0,1,X) output pixels being calculated.

Figure 5.9: Matrices representing in 2D the 3D input activations (each position representing all the elements in the z axis), showing with colors the composition of the activation memory, when different output pixels are being calculated. In red are the activations stored in local memory and being used to calculate the output pixel. In blue are the activations stored in local memory and waiting to be used again. The zeros along the borders represent the padding of the matrices

To implement the FIFOs in HLS a library was included as shown in line 1 of the code in figure 5.11. The two FIFOs were declared as different objects in line 4 and 5. The shift-registers were declared as an array in line 2, and it was used a *HLS_PARTITION* pragma with the option complete, to guarantee random accessibility to their values.

Every time a new output pixel needs to be calculated new activations must be loaded to the activation memory as discussed above and showed in figure 5.9. The input activations enter the hardware architecture using a 64 bit data stream, this is sets of 8 activation values, however each time we load a new set of activations we need activations in a specific x and y coordinates but spanning the whole z dimension which means that 256 activations must be loaded. A loop is needed to fetch N_IN8 $= 256/8 = 32$ times the input stream as shown in line 1 of figure 5.12. The if statement between line 4 and 8 selects if the new activations values are padding or are fetched from the stream, being in line 7 where the stream is actually fetched. The function *shift_activation(ap_int<64>in)* in line 10 shows a function specifically made for this work, that receives as argument the new value to load into the activation memory, and shifts the activations values already in memory following the flow specified in figure 5.10. The *HLS*

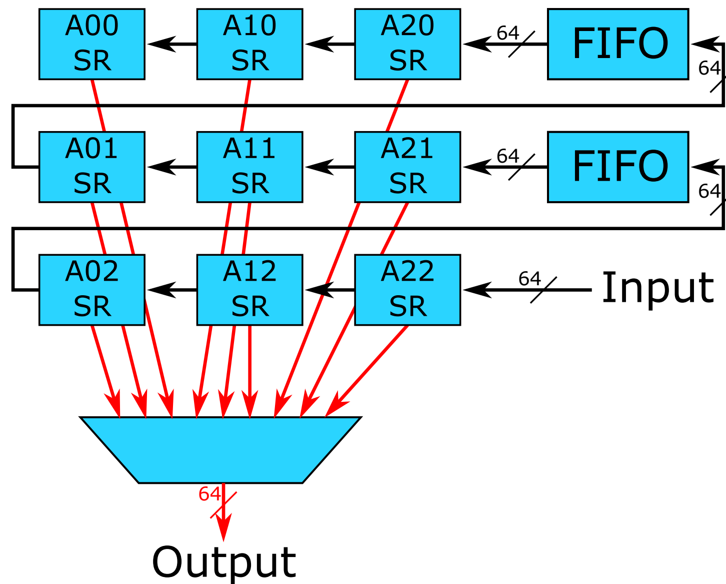


Figure 5.10: Activation memory hardware architecture. Arrows in black show the loading path of activations. Arrows in red show the reading path of activations

```
1  #include "hls_stream.h"
2  ap_int<64> activation[A_MATRIX];
3
4  hls::stream< ap_int<64> > line1;
5  hls::stream< ap_int<64> > line2;
6
7  #pragma HLS ARRAY_PARTITION variable=activation complete dim=1
```

Figure 5.11: HLS code that declares the activation memory.

PIPELINE pragma showed in line 2 of figure 5.12, is used to make the HLS synthesizer make this loop iteration with a throughput of 1, by creating all the pipeline stages necessary.

When the output pixel being calculated change lines as shown between figure 5.9 (c) and 5.9 (d), three new x/y position must be loaded, which means that the code showed in figure 5.12 must be repeated three times.

```
1  for(int index1=0; index1<N_IN8; index1++){
2  #pragma HLS PIPELINE
3      ap_int<64> aux;
4      if(j+WEIGHT_X>=a_x || i+WEIGHT_Y>a_y){
5          aux = 0;
6      }else{
7          aux = in.read();
8      }
9
10     shift_activation(aux);
11 }
```

Figure 5.12: HLS code that declares a load of a new set of values to the activation memory.

5.4 Weight memory

All the weights from a single convolution layer fit in the local memory of the FPGA, provided that the weight memory is implemented using BRAMs. Since weights are stored in 32 bits words and each convolution layer uses $3 * 3 * 256 * 256 * 4 = 2.359$ Mb of space, a memory space with a depth of $2359296/32 = 73728$ positions is needed. This makes the address space for the weight memory not a power of two, which makes the HLS synthesis tool to generate a memory with an unnecessary excessive number of BRAMs (128 instead of 72).

To solve this issue the weight memory array in HLS code is divided in 9 smaller memories with an address space that is a power of two. Each one of those smaller memories correspond to a different kernel position.

The weight memory is organized with 8 dual-port BRAMs, to provide 16 independent ports and therefore allow simultaneous read accesses to each of the 16 PEs. These 8 banks of memory are represented, in figure 5.13, where each bank of memories is composed by the 9 memories referred above.

Figure 5.14, line 1, shows the weight memory being specified as a bi-dimensional array, such that the first dimension is fully partitioned in 9 independent memories corresponding to the 9 kernel elements using *HLS ARRAY_PARTITION* pragma with *complete* option, shown in line 3.

Data in the weight memory must be arranged in way that weight values accessed simultaneously are displaced evenly between memories, to take advantage of all the ports provided. Each PE is doing a different output channel at the same time, in this way the values being accessed concurrently are contiguous in N dimension, being N the filter number. By storing the weight in NZXY format and using the *HLS ARRAY_PARTITION* pragma with *cyclic* option and *factor=8* option, shown in line 2, contiguous weight values are stored in different memories, only repeating the same memory after 8 values, which also means that all the values with the same N coordinate are in the same memory and every memory will only contain two different filters.

5.5 Convolution IP

The general structure of the HLS code, shown in figure 5.15, is composed by nested loops that iterate over the output feature maps positions, the input activation and the weight filters, similar to the structure used in a general algorithm for convolution layers like the one shown in figure 3.1.

The first step of the developed algorithm is loading all the weights and bias from the external memory to the on FPGA memory, and load the first activations until it fills the activation memory. After loading the local memories, the scheduling of operations is done by the nested loops.

The 3 outer loops control which pixels of the output feature map are being made. These loops are represented in lines 5, 4 and 3 and they iterate over the z, x, y axis of the output activations respectively. The loop in line 5 is incremented in each iteration by the number of PE's, because in each iteration a number of pixels equal to the number of PE are processed at the same time, since each PE is respon-

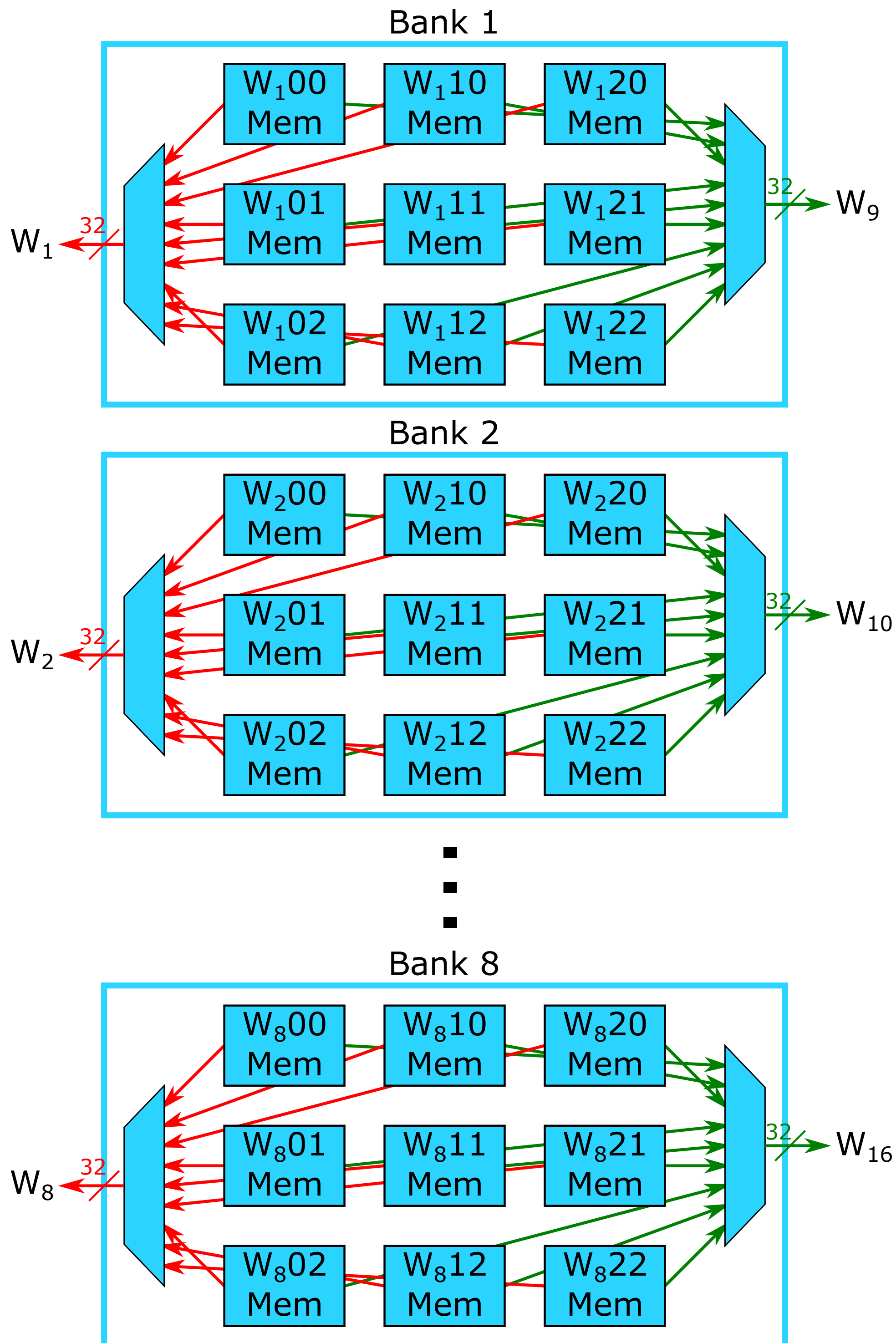


Figure 5.13: Weight memory hardware architecture. Arrows in different colors correspond to different ports of the dual port memories.

```
1  ap_int<32> weight[KERNEL_SIZE][WEIGHT_SIZE];
2  #pragma HLS ARRAY_PARTITION variable=weight cyclic factor=8 dim=2
3  #pragma HLS ARRAY_PARTITION variable=weight complete dim=1
```

Figure 5.14: HLS code that declares the weight memory.

sible for a different pixel in the z axis. Every time the 3 outer loops change the pixels that are being processed, the accumulators of the MACs are reset.

The 3 inner loops, in line 11, 10 and 9 iterate over the weight filter in the z, x and y axis respectively. Inside the three inner loops are PE's multiply and accumulate code that is shown in figure 5.6. A *HLS PIPELINE* pragma is used to specify a throughput of one for the MACs.

Once all MAC operations are completed for a given set of output pixels, the results are summed using a tree of adders. This summation is executed once per output pixel as shown in line 20.

```
1  //Load weights, bias and activations
2
3  for(int i=-1; i+(WEIGHT_Y-1)<=a_y; i++){
4      for(int j=-1; j+(WEIGHT_X-1)<=a_x; j++){
5          for(int k=0; k<N_OUT; k+=PE){
6
7              //Reset accumulator
8
9              for(int l=0; l<WEIGHT_Y; l++){
10                 for(int m=0; m<WEIGHT_X; m++){
11                     for(int n=0; n<N_IN8; n++){
12 #pragma HLS PIPELINE
13
14                         //PEs multiply and accumulate
15
16                     }
17                 }
18             }
19
20             //PEs sum trees
21
22         }
23
24         //Load activation from external memory
25
26     }
27
28     //Load activation from external memory x2
29
30 }
```

Figure 5.15: General structure of the HLS code of convolution accelerator.

A new set of input activations is loaded each time the x or y coordinates of the output pixel to be determined change. If the x coordinate is incremented, it means the movement is along the same line of the input activation matrix, like the one represented between figure 5.9 (a) and (b). The code that makes this activation load is placed in line 24 of HLS code structure shown in figure 5.15. On the other hand if the y coordinate is incremented, it means there is a change in the line of the input activation matrix like the one represented between figure 5.9 (c) and (d). This load activation activity requires three times more the number of activation values than the ones when the movement is across the same line. In this way an additional load activation code is placed in line 28. These two load activations are shown on the timelines in figure 5.3 and figure 5.5.

Loop name	Number of iterations		Iteration latency (clock cycles)		Initiation latency (clock cycles)	Loop latency (clock cycles)	
	Min	Max	Min	Max		Min	Max
Weight load	36864	36864	2	2	0	73728	73728
Bias load	32	32	1	1	0	32	32
Initial activation load	544	5344	1	1	4	548	5348
Output pixel Y	5	80	3454	468950	0	17270	37516000
Output pixel X	5	80	677	5861	0	3385	468880
Output pixel Z	16	16	40	364	0	640	5824
Filter Y & X	9	9	2	38	0	18	342
Filter Z	32	32	1	1	2	34	34
Activation load	32	32	1	1	1	33	33
Activation load	64	64	1	1	1	65	65
Total						**91578**	**37595108**

Table 5.1: Time profiling across the different loops of convolution accelerator.

5.6 Convolution accelerator results

The timing results of the convolution accelerator are presented in table 5.1. The timing profiling shows the number of clock cycles scheduled by HLS for each loop. The first column shows the number of iterations of each loop. For some loops such as the *Initial activation load*, *Output pixel Y* and *Output pixel X* the number of iterations depends on the size of the input activation matrix. The iteration latency column shows the maximum and minimum number of clock cycles each iteration takes. For some loops that are pipelined there is an extra latency value to be accounted that is the amount of clock cycles taken to fill the pipeline, which is shown in the initialization latency column. The total latency of each loop can be calculated by multiplying the number of iterations with the iteration latency and adding the initiation latency overhead. This gives a range of values since the number of iterations and iteration latencies can both be variable.

There are 2.359 Mb of weights to be loaded through a stream channel of 64 bits. In each iteration of the weight load loop 64 bits of weights are stored in local memory, in this way it takes 36864 loops to store all the weights as shown in table 5.1.

For a single convolution layer there are 256 bias values of 8 bits which gives a total of 2048 Mb to load. In each iteration 64 bits of bias are loaded, giving a total of 32 iterations to load all the bias.

The number of activations to be initially loaded (N) depend on the line size of the input activation matrix ($input_X$). As $N = ((input_X + 2) * 2 + 3) * 256$, the number of iterations of the initial activation load loop is variable. In each iteration 64 bits of activations are loaded and since each activation has a bit width of 8, it takes $I = ((input_X + 2) * 2 + 3) * 32$ iterations to load all the initial activations. In this way, when the input activation matrix has a dimension of 80×80×256, the biggest used in the accelerated part of Retinanet, then 5344 iterations are necessary to load the initial activations as shown in table 5.1. If the smallest activation matrix is considered, having a dimension of 5×5×256, then only 544 iterations are needed. The initial activation load loop was pipelined to achieve a throughput of one reading of the input stream per clock cycle.

The output pixel Y, output pixel X and output pixel Z loops represent the iteration over the output activation matrix. The number of iterations of output pixel Y and output pixel X loops depends only on

the dimension of the input activation matrix that is equal to the dimension of the output activation matrix. The number of iterations of output pixel Z is 16 because there are a total of 256 filters and there are 16 PEs each one doing a different filter at the same time. Since each iteration computes 16 pixels across the z axis, it takes 16 iterations to do all the 256 filters.

Filter Y & X loop, represents two loops, one for each dimension, that are combined by the HLS synthesizer in one loop because there is no code between the declaration of those two loops. To optimize the architecture each time it is detected that the operation being made during that iteration is with a padding value, the loop jumps to the next operation. This avoids doing operations with padding values since, these values are zeros and their multiplication with a weight would not contribute to the accumulator value. This is why there is a difference between the maximum and minimum iteration latency of this loop as shown in table 5.1. The number of iterations of filter Y & X loop is the number of values is a 3x3 kernel.

The filter Z loop, which iterates over the Z axis of the filter, was pipelined in order to achieve an iteration latency of one, which means a multiply and accumulate operation per clock cycle. This optimization has a big impact because the latency of this loop is successively multiplied by the number of iterations of the outer loops. Filter Z loop has 32 iterations because each PE has 8 MACs processing 8 weights across the z axis in each iteration. Since there are 256 weights across the z axis it takes 32 iterations to complete all the operations across that axis.

As described in section 5.5, when a new output pixel in the x axis is iterated, 256 values of activations have to be loaded. These values are loaded in loop activation load, and they take 32 iterations to be loaded, since 64 bits of activations are loaded in each iteration.

When a new output pixel is iterated in y axis it needs more two sets of activations, which means 512 activations. The 512 activations are loaded in 64 iterations of the activation load loop.

The convolution accelerator was targeted to achieve a clock cycle of 10 ns and it achieved 9.7 ns before implementing (not accounting for route delays). By multiplying the maximum total latency of the convolution accelerator in clock cycles shown in table 5.1 by the target clock cycle we have an estimate for the time it takes to process a convolution layer with an input activation matrix with $80\times80\times256$ dimension, giving 376 ms. This value gives a speedup of 495 comparing to 186 s that it takes to compute the convolution layer in software only as shown in section 4.3.1.

The FPGA resources used by the convolution accelerator are shown in table 5.2, the used percentage is taking into account the available resources in the selected targeted device the Zynq-7020.

The 128 DSP are totally used by the PE's MACs, since each PE has 8 MACs and there are 16 PEs. The BRAM usage of 89 is shared between the bias, weight and activation memories. The weigh memory uses 72 BRAMs, each FIFO of the activation memory uses 7.5 BRAMS giving a total of 15 BRAM for the activation memory and the bias memory uses 2 BRAMs. There are 18432 flip-flops used by the activation memory for the shift registers.

Resource	Units used	Used percentage
LUT	44940	84.5%
FF	42695	40.1%
LUTRAM	0	0.0%
BRAM	89	40.1%
DSP	128	63.6%

Table 5.2: Resource usage of convolution accelerator.

5.7 Conclusion

In this chapter a convolution hardware accelerator with 16 PEs was designed. Each PE is composed by 8 MACs, enabling the accelerator to make 128 MAC operations simultaneously. To reduce as much as possible the idle time of this MACs, on-chip FPGA memory was created to store the data being fed to the MACs. The local memories created achieved total reuse of data, reducing the number of transactions with the DDR memory thus reducing the time it takes to access data. This chapter introduced several techniques used in HLS to optimize memory usage and displace data in convenient ways, to achieve high bandwidths.

The developed convolution accelerator used a 10 ns clock cycle, and is able to execute a convolution layer with an input activation matrix with $80 \times 80 \times 256$ in 376 ms. The resulted execution time achieves a speedup of 495 comparing to the execution of a convolution in an ARM processor. The hardware accelerator occupies 84.5 % of the total number of LUTs available in the target device.

Chapter 6

RetinaNet hardware/software implementation

In this chapter the convolution hardware accelerator is integrated into the embedded system using the hardware architecture described in section 6.1.1. Changes to the embedded software are made to accommodate the newly added convolution hardware accelerator component, these changes are described in section 6.1.2. The experimental results of the HW/SW RetinaNet implementation are showed in section 6.2.

6.1 Hardware/software system

The parts of RetinaNet selected for the final implementation were the regression and classification subnetworks. The hardware/software architecture run some layers in the embedded ARM processor and others in the convolution hardware accelerator. All the convolution layers in the implemented part of RetinaNet, apart from the ones that were not quantized, are processed in the convolution hardware accelerator, which accounts for 40 layers of the total of 50 layers. The sigmoid layers, the input quantization layers and the data conversions and arrangements functions are processed in the ARM processor.

The maximum input activation matrix that the convolution hardware accelerator can compute is $80\times80\times256$ dimension, this restricts the input image of the complete RetinaNet DNN to a maximum image of 640×640. This was the chosen size for the images used to test the system.

6.1.1 Hardware architecture

The hardware architecture is composed by two parts: the programmable logic (PL), this is all the components that are implemented using the FPGA resources; and the processing system (PS), this is the ARM processor and some related peripherals. The block diagram of the developed architecture is shown in figures 6.1 and figure 6.2, the latter is less detailed, only showing the data interfaces, and does not include the processor system reset.

The PS (*processing_system7_0*) is configured with a CPU clock frequency of 667 MHz, and DDR memory has a clock frequency of 533 MHz. The PS also generates a clock frequency for the PL, this clock frequency was configured at 71 MHz. The PS uses an AXI-Lite interface to connect to an AXI interconnect IP (*ps7_0_axi_periph*), this IP splits the AXI-Lite interface to the convolution hardware accelerator (*conv3d_0*) and the DMA IP (*axi_dma_0*), enabling one single interface to communicate with to different hardware components. The PS sends through the AXI-Lite interface to the convolution accelerator, control data, specifying several characteristics of the convolution layer being executed in the accelerator, as explained in section 5.1. The PS also uses the same AXI-Lite interface to control the DMA, controlling the content that is being transferred to and from the convolution accelerator at any given time.

The DMA responsibility is to manage the memory transferences between the DDR and the convolution accelerator. The DMA is connected to the convolution accelerator through two AXI-Stream interfaces, one that sends input activations, weights and bias to the accelerator and the other that receives the output activations from the accelerator. An AXI interconnect IP (*axi_mem_intercon*) is used to connect the DMA IP to the PS through an AXI HP port so that it can access the DDR memory.

The processor system reset component is showed in figure 6.1 and is used to generate system resets, being added automatically to the design by the development tools.

The ARM processor controls the convolution accelerator, by sending control data to the accelerator through the AXI-Lite connection and also by issuing memory transactions to the DMA.

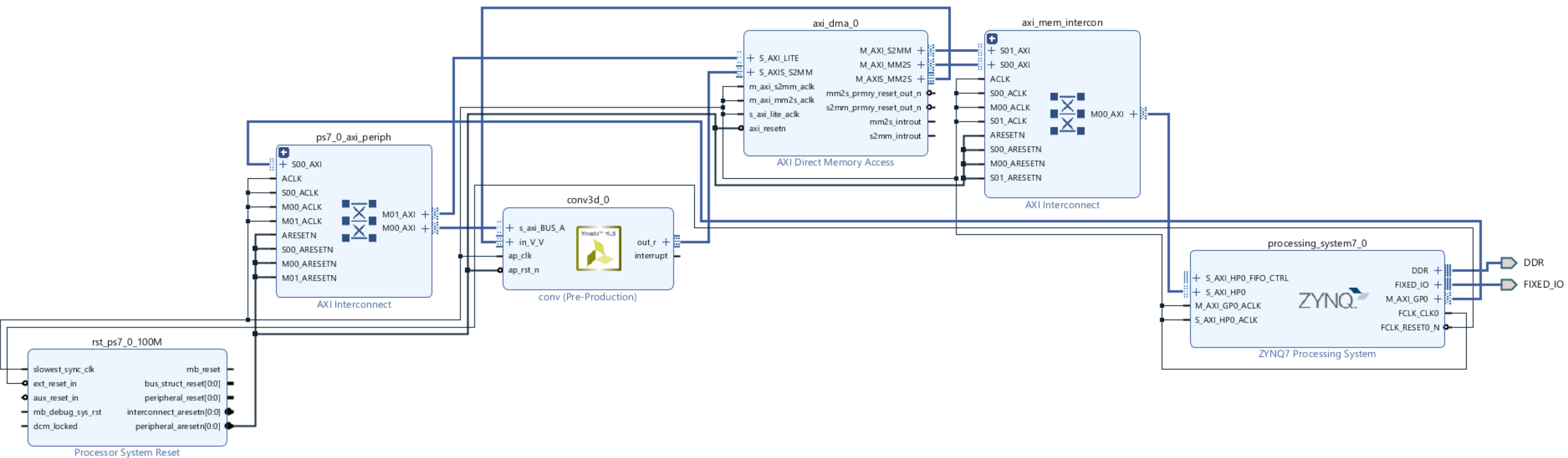


Figure 6.1: Hardware architecture of the RetinaNet hw/sw implementation.

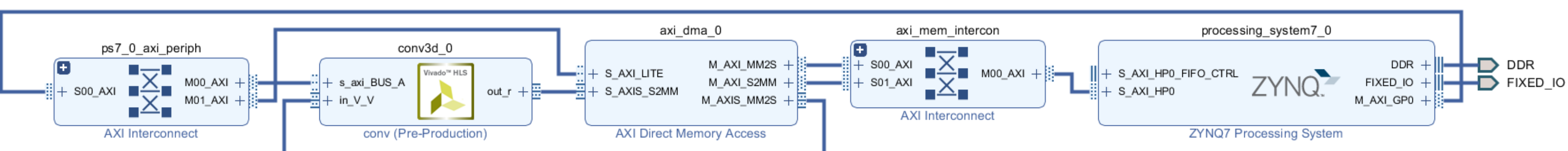


Figure 6.2: Simplified hardware architecture of the RetinaNet hw/sw implementation.

6.1.2 Embedded software

To issue the execution of a layer in the convolution hardware accelerator, the function in figure 6.3 was made. This function receives as arguments all data that is needed to execute the convolution layer and sends it either through the AXI-Lite connection to the convolution hardware accelerator, or issues transactions to the DMA IP, so that the weights, bias and activations at the given addresses are transferred to the accelerator.

To send control data to the accelerator, a set of values are written to specific AXI-Lite interface registers, as shown in lines 6 to 12. The *input_control* points to a register, that signals the state of the AXI-Lite interface. To transfer all the control values to the accelerator the memory region pointed by *input_control* is set to 1 after all the control values are written to their registers.

In line 14 an order is issued to the DMA to transfer the weights, pointed by pointer *B*, from the DDR to the convolution accelerator. The size of the weights is also part of the issued command, so that the DDR knows when to stop the transaction. In line 15 a function is used to signal when the transaction was completed, blocking the code execution until the end of the transaction. Lines 17 and 18 shows the transference of the bias from the DDR to the convolution accelerator. In line 20 the DMA is set up to send the output activations sent by the convolution accelerator to the memory region pointed by *C*. This is done before transferring the input activation to the convolution accelerator, to prevent blocking the accelerator, that would be waiting for the DMA to accept its output. In line 22 and 23 the input activations are transferred to the accelerator. In line 25 the function *conv_hw()* is prevented from returning until all the output activations of that convolution layer are sent to the DDR.

Figure 6.4 shows the implementation of a classification stage using the *conv_hw()* function to execute the convolution layers in the accelerator.

In line 3 the input activations are quantized and then changed to the ZXY data storage format in line 5. Before using the hardware accelerator, since it accesses directly the DDR memory, the cache contents are flushed into the DDR memory. This prevents the hardware accelerator from accessing outdated values. Between lines 8 and 11 four convolution layers are issued to be executed in the accelerator. In line 13 the cache is invalidated to prevent the ARM processor from using outdated values, since the DDR memory has been updated directly by the convolution hardware accelerator. The activation values produced by the hardware accelerator are converted again to the XYZ data storage format in line 14 and converted in floating point numbers line 16. The last convolution layer of the stage is executed in the ARM processor in line 18. All the activations are then passed through a sigmoid function in line 20 that is also executed in the ARM processor. Finally, the output activations are arranged in a ZXY format again in line 22 to be concatenated with the activations from the other stages and stored in the right format to be processed by the python script *view_image.py*, that was described in section 4.2.

The regression stages are implemented similarly to the classification stages, being the most significant difference the absence of the sigmoid layer. All the regression and classifications stages are issued in a code similar to the one used for the baseline implementation in figure 4.12. The *define_memory_regions()* function defined in figure 4.11 is used in this implementation to establish the memory location of each weight, bias and activation.

```
1  void conv_hw(int8_t* C, int8_t* A, int8_t* B, int8_t* bias, int* C_x, int*
       C_y, int A_x, int A_y, int B_x, int B_y, int N_in, int N_out, int
       output_scale, int input_scale, int weight_scale, int bias_scale){
2
3       *C_x = A_x;
4       *C_y = A_y;
5
6       *input_a_x = A_x;
7       *input_a_y = A_y;
8       *input_output_scale = output_scale;
9       *input_input_scale = input_scale;
10      *input_weight_scale = weight_scale;
11      *input_bias_scale = bias_scale;
12      *input_control = 1;
13
14      DMA_ddr2stream((UINTPTR)B, (int)((B_x*B_y*N_in*N_out)/2));
15      DMA_wait_ddr2stream();
16
17      DMA_ddr2stream((UINTPTR)bias, (int)(N_out));
18      DMA_wait_ddr2stream();
19
20      DMA_stream2ddr((UINTPTR)C, (int)(A_X*A_Y*N_out));
21
22      DMA_ddr2stream((UINTPTR)A, (int)(A_x*A_y*N_in));
23      DMA_wait_ddr2stream();
24
25      DMA_wait_stream2ddr();
26 }
```

Figure 6.3: Functions that issues the execution of the convolution hardware accelerator.

6.2 Experimental results

Table 6.1 shows the resource usage of the RetinaNet hardware/software implementation. Most of the resources are used by the convolution accelerator. All the DSPs used are in this component, and 89 of the 92 BRAMs used are in the accelerator. The DMA is the second component to use more resources, however it only accounts for 2.9 % of LUT usage and 2.1 % of Flip-flop usage. The DMA also uses 3 BRAM for buffering proposes. The other components have a negligible resource usage.

The high LUT usage of the system makes it harder to optimize the signal paths ending up with long paths between registers. This issue make it difficult to achieve high clock frequencies. A clock period of

Component	LUT		FF		BRAM		DSP	
	Units used	Used percentage	Units used	Used percentage	Units used	Used percentage	Units used	Used percentage
Convolution Accelerator	44,940	84.5 %	42,695	40.1 %	89	40.1 %	128	63.6 %
DMA	1,565	2.9 %	2,231	2.1 %	3	2.1 %	0	0.0 %
axi_mem_intercon	541	1.0 %	648	0.6 %	0	0.0 %	0	0.0 %
ps7_0_axi_periph	503	0.9 %	657	0.6 %	0	0.0 %	0	0.0 %
Processor System Reset	16	0.0 %	33	0.0 %	0	0.0 %	0	0.0 %
Total	**47,565**	**89.4 %**	**46,264**	**43.4 %**	**92**	**42.2 %**	**128**	**63.6 %**

Table 6.1: Resource usage of RetinaNet hardware/software implementation.

```
1  void classification(float* C, float* A, int* C_x, int* C_y, int A_x, int
       A_y, float input_scale, int input_scale_int){
2
3      quant(int_data2,A,A_x*A_y*256,input_scale);
4
5      xyz_to_zxy(int_data, int_data2, &int_data_x, &int_data_y, A_x, A_y,
           256);
6      Xil_DCacheFlushRange((INTPTR)int_data, (u32)(int_data_x*int_data_y*256*
           sizeof(int8_t)));
7
8      conv_hw(int_data2,int_data,classification_conv1_weight,
           classification_conv1_bias,&int_data2_x,&int_data2_y,int_data_x,
           int_data_y,3,3,256,256,4,input_scale_int,6,8);
9      conv_hw(int_data,int_data2,classification_conv2_weight,
           classification_conv2_bias,&int_data_x,&int_data_y,int_data2_x,
           int_data2_y,3,3,256,256,4,4,6,8);
10     conv_hw(int_data2,int_data,classification_conv3_weight,
           classification_conv3_bias,&int_data2_x,&int_data2_y,int_data_x,
           int_data_y,3,3,256,256,4,4,6,8);
11     conv_hw(int_data,int_data2,classification_conv4_weight,
           classification_conv4_bias,&int_data_x,&int_data_y,int_data2_x,
           int_data2_y,3,3,256,256,4,4,6,8);
12
13     Xil_ICacheInvalidateRange((INTPTR)int_data, (u32)(int_data_x*int_data_y
           *256*sizeof(int8_t)));
14     zxy_to_xyz(int_data2, int_data, &int_data2_x, &int_data2_y, int_data_x,
            int_data_y, 256);
15
16     dequant(data,int_data2,256*int_data2_x*int_data2_y,0.0625);
17
18     conv_float(data2,data,classification_conv5_weight,
           classification_conv5_bias,&data2_x,&data2_y,int_data2_x,int_data2_y
           ,3,3,256,720,1,1);
19
20   sigmoid(data2,data2,&data2_x,&data2_y,data2_x,data2_y,720,0.00195313);
21
22   arrange_tensor(C,data2,C_x,C_y,data2_x,data2_y,720,80);
23 }
```

Figure 6.4: Functions that implements a convolution stage using the convolution hardware accelerator.

10 ns was attempted, however the big route delays made it unsuccessful.

The achieved clock period was 14 ns, having a worst setup slack of 0.038 ns. To achieve this results, several synthesis and implementation Vivado strategies were tried. The set of Vivado strategies that enabled the best results were the synthesis strategy *Flow_PerfOptimized_high* together with the implementation strategy *Performance_ExplorePostRoutePhysOpt*.

The critical path is in the data path of the convolution accelerator between a flip-flop and a DSP register. The route delay accounts for 89 % of the total delay of this path. Figure 6.5 shows the critical path in the floor plan of the targeted device. As shown, the path is very long, crossing almost through the entire length of the FPGA.

Table 6.2 shows the time profiling of the hardware/software implementation together with the software only implementation. A more detailed layer-by-layer comparison between both implementations can be

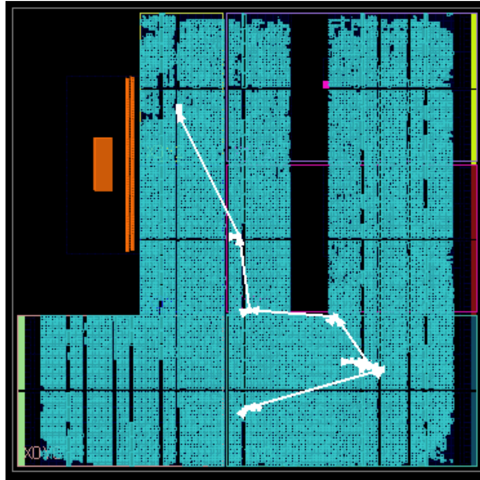


Figure 6.5: Critical path of RetinaNet hardware/software implementation (in white) shown in the floor plan of the targeted device.

seen in appendix B.

As seen on appendix B the layers with an input activation matrix of 80×80×256 dimension, these are the biggest convolution layers implemented in hardware, take about 519 ms to execute. Comparing the execution of a single layer between both software only and hardware/software implementation a speedup of 359 is achieved. When looking to the overall speedup of a single convolution or regression stage the speedups achieved are much lower, since one of the layers in each stage in not executed in the convolution accelerator. In the HW/SW implementation the time taken by the convolution that is processed in software accounts for 87 % of time taken by the regression stage where it belongs. This number is even higher for the convolution stages where the software processed convolution accounts for 98 % of the total time of the stage.

The speedup achieved in overall by the regressions stages is 35, much higher than the 2.9 speedup achieved by the classification stages. This is due to the fact that the software processed convolution layer in the classification stages is bigger than the software processed convolution layer in the regression stages.

The total speedup achieved by the HW/SW implementation is 4.6. There is a great potential to achieve higher speedups by accelerating more layers.

6.3 Conclusion

In this chapter the HW/SW implementation was developed and detailed, both the hardware architecture and the embedded software were shown. A new function was made to issue transfer orders to the DMA and to load control data into the convolution accelerator. Special care was taken to ensure caches were cleaned or flushed to maintain the memory integrity between the ARM processor and the hardware accelerator.

The resource usage and timing results of the HW/SW implementation were also shown in this chapter. The high usage of LUTs made it impossible to achieve lower clock periods than 14 ns, since the

Stage	Xilinx Zynq-7020 SW only (ARM Cortex A9)			Xilinx Zynq-7020 HW/SW (ARM Cortex A9)			Speedup
	Execution time		Partial execution time	Execution time		Partial execution time	
	s	%	%	s	%	%	
Regression S2	764	-	31.7	21.2	-	76.3	36.0
Regression S3	157	-	6.5	5.1	-	18.4	30.7
Regression S4	39	-	1.6	1.1	-	4.1	34.6
Regression S5	9	-	0.4	0.3	-	1.0	32.9
Regression S6	2	-	0.1	0.1	-	0.2	35.8
Total Regression	**971**	**40.3**	**100.0**	**27.8**	**5.3**	**100.0**	**35.0**
Classification S2	1,118	-	46.4	376.1	-	76.3	3.0
Classification S3	243	-	10.1	90.9	-	18.4	2.7
Classification S4	58	-	2.4	20.2	-	4.1	2.9
Classification S5	14	-	0.6	5.0	-	1.0	2.8
Classification S6	3	-	0.1	1.0	-	0.2	3.0
Total Classification	**1,436**	**59.7**	**100.0**	**493.2**	**94.7**	**100.0**	**2.9**
Total	**2,407**	**100.0**	-	**521.0**	**100.0**	**-**	**4.6**

Table 6.2: Time profiling of RetinaNet hardware/software implementation and comparison with software only implementation.

route delays were high.

The speedup between the HW/SW implementation and the software only implementation were obtained. The HW/SW implementation achieves a speedup of 359 in a single layer, however some layers were not processed using the hardware convolution accelerator which made the overall speedup to be 4.6.

Chapter 7

Conclusion

The chosen object detection DNN to be accelerated in this work was RetinaNet-18. This network is composed by a total of 174 layers of which 78 are convolution layers. This DNN is known for high-end accuracy results, however due to its large size it has above average inference latencies. The objective of this work was to reduce the inference latencies in a resource limited embedded environment, with a minimal loss of accuracy of the network.

Weights and activations from 68 of the 78 convolution layers in RetinaNet were quantized, using quantization aware training. A study with different quantizations was made to determine the more adequate trade-off between accuracy and memory usage. The chosen quantization of 4 bits weights and 8 bits activations and bias achieved better results than competitive DNNs with similar sizes, and occupies 8 times less memory for the storage of the weights than the non quantized RetinaNet.

The entire RetinaNet was implemented in C, two versions were created, one using floating point data and the other using the quantization chosen in the quantization study. It was determined that 76.2 % of the inference time of the DNN was derived from the two last parts of RetinaNet, the regression and convolution sub-networks. These two parts of the object detection DNN account together for 50 convolution layers that were implemented in a Xilinx Zynq-7020 embedded system. The timing results from this implementation were registered to be used as baseline for the comparison with the accelerated system. The results showed that convolution layers were the main responsible for the inference latency time in DNNs. 40 (of the 50) convolution layers were quantized and therefore suitable to be accelerated in hardware.

A convolution hardware accelerator was created, with 16 PEs, each one capable of doing 8 MAC operations in parallel, which can to achieve 128 MAC operations per clock cycle. Three specially designed memory arrangements were made to accommodate the activations, weights and bias in internal FPGA memory, reducing the latency issues of accessing an external memory randomly. In spite of the limited size of internal FPGA memory, the total reuse of data was possible, due to a carefully optimized scheduling of operations and the designed internal memory arrangements.

The convolution hardware accelerator was integrated in a co-designed hardware software implementation. This system achieved a speedup of 4.6 compared to the baseline implementation. This results

also showed a speedup of 359 comparing a hardware processed layer with a software processed layer. The final results are very promising, introducing a proof of concept for the expansion of the convolution layer acceleration to more layers.

7.1 Future Work

Two different strategies can be adopted to improve the work. The first one is adapting the system to enable more convolution layers to be processed by the convolution layer accelerator, as shown in section 7.1.1. The other strategy is optimizing the hardware architecture to use less resources, reducing route delays and enabling higher clock frequencies, as showed in section 7.1.2.

7.1.1 Expand convolution layer acceleration to more layers

The big difference between the speedup achieved by a single convolution layer and the entire system speedup, shows big room for improvements by just processing more convolution layers in the convolution hardware accelerator. There are two reasons why more layer are not yet being processed by the accelerator:

- There are layers that could not be quantized without affecting the capacity for the system to detect objects.
- Some layers have different properties from the ones that the convolution accelerator supports.

To solve the problem of having convolution layers that are not quantized, an in depth study should be taken to understand the reasons behind it. The quantization of bias, that was not tried in this work, could make it possible to quantize these layers. In addition, trying different quantization, like 16 bit-width activations and weights in a quantization aware framework could be the key to enable the quantization of these convolution layers. In terms of memory capacity theoretically the used convolution accelerator architecture can be scaled to accommodate 16 bit-width activations, in the worst case it would be necessary to reduce to half the number of PEs. The reduction in the number of PEs would have a negative impact on the inference latency of a single convolution layer, however the time saved by accelerating all layers would compensate for that loss.

Measures can be taken to make the convolution accelerator compatible with a bigger diversity of convolution layers properties. These are the current fixed characteristics that do not support all convolution layers in RetinaNet:

- Fixed kernel size of 3×3, not supported by the first convolution layer that has a 7×7 kernel, downsample convolution layers and some FPN convolutions layers that use 1×1 kernel;
- Fixed stride and padding, not supported by several layers across the network;
- 256 input and output number of channels, not supported by several layers across the network;

- Maximum x and y dimension of 80×80 for the input activation matrices, not supported by some convolution layers in the stage 2 of ResNet and all convolution layers in the stage 0 and 1 of ResNet;

To make the accelerator compatible with 7×7 kernel size, more memory resources must be used to store more weights and activations. However, to make it flexible to accept smaller kernels is much easier, as changes have only to be made to the schedule of operations, so that the size of the kernel loop iterations are dependent on the kernel size.

The stride and padding flexibility can be dealt in the same way of the kernel size, and the only changes to be made are in the scheduling of operations.

If the number of input and output channels are less than 256, changes have to be made to the scheduler and some shortcuts in the shift registers of the activation memory have to be added. If the number of input or output channels are bigger than 256, the best solution might be dividing the input matrices in smaller dimensions and process the same layer several times in the convolution accelerator as if it were different layers. The results of the smaller matrices have to be concatenated or summed together at the end depending on the case.

To add larger input and output activations in the x and y dimensions compatibly, the same strategy of divide and conquer of the solution for larger input and ouput channels can be used. The input matrices are divided in smaller ones, that are processed individually and then concatenated using the ARM processor.

7.1.2 Optimize hardware accelerator

One way to optimize the hardware accelerator and reduce the resource usage, is sharing one sum tree with all PEs. The sum tree can be pipelined and have a throughput of 1 activation per clock cycle. This change is not going to have a significant impact in performance because the output stream is only able to transmit 1 activation per clock cycle. In addition, the PEs already have to wait for the new activations to be loaded into the internal FPGA memory, so there is no advantage in having a fast output data flow.

The nested loops used for the convolution algorithm in the HLS code are not ideal to be mapped to hardware. The HLS code only allows to pipeline the inner loop. Nested loops make it difficult for the developer to have fine control over the architectural details. A single loop approach where a single iteration variable control a set of if statements is a better approach to control the schedule of operations.

Bibliography

[1] J. Walsh, N. O' Mahony, S. Campbell, A. Carvalho, L. Krpalkova, G. Velasco-Hernandez, S. Harapanahalli, and D. Riordan. Deep learning vs. traditional computer vision. 04 2019. ISBN 978-981-13-6209-5. doi: 10.1007/978-3-030-17795-9_10.

[2] M. Aryal. Object detection, classification, and tracking for autonomous vehicle. Master's thesis, Grand Valley State University, 12 2018.

[3] A. Raghunandan, M. Mohana, R. Pakala, and R. V. Object detection algorithms for video surveillance applications. 04 2018. doi: 10.1109/ICCSP.2018.8524461.

[4] S. Kamate and N. Yilmazer. Application of object detection and tracking techniques for unmanned aerial vehicles. *Procedia Computer Science*, 61:436 – 441, 2015. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2015.09.183. URL http://www.sciencedirect.com/science/article/pii/S1877050915030136. Complex Adaptive Systems San Jose, CA November 2-4, 2015.

[5] L. Hu and Q. Ni. Iot-driven automated object detection algorithm for urban surveillance systems in smart cities. *IEEE Internet of Things Journal*, 5(2):747–754, April 2018. doi: 10.1109/JIOT.2017.2705560.

[6] S. Benhimane, H. Najafi, M. Grundmann, Y. Genc, N. Navab, and E. Malis. Real-time object detection and tracking for industrial applications. In *VISAPP*, 2008.

[7] Z. Li, M. Dong, S. Wen, X. Hu, P. Zhou, and Z. Zeng. Clu-cnns: Object detection for medical images. *Neurocomputing*, 350:53 – 59, 2019. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2019.04.028. URL http://www.sciencedirect.com/science/article/pii/S0925231219305521.

[8] X. Zhao, E. Delleandrea, and L. Chen. A people counting system based on face detection and tracking in a video. In *2009 Sixth IEEE International Conference on Advanced Video and Signal Based Surveillance*, pages 67–72, Sep. 2009. doi: 10.1109/AVSS.2009.45.

[9] M. A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[10] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes challenge: A retrospective. *International Journal of Computer Vision*, 111(1):98–136, Jan. 2015.

[11] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998. ISSN 1558-2256. doi: 10.1109/5.726791.

[12] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár. Microsoft coco: Common objects in context, 2015.

[13] V. Sze, Y. Chen, T. Yang, and J. S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, Dec 2017. ISSN 1558-2256. doi: 10.1109/JPROC.2017.2761740.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

[15] X. Han, Y. Zhong, C. Liqin, and L. Zhang. Pre-trained alexnet architecture with pyramid pooling and supervision for high spatial resolution remote sensing image scene classification. *Remote Sensing*, 9:848, 08 2017. doi: 10.3390/rs9080848.

[16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL `http://arxiv.org/abs/1512.03385`.

[17] F. Ramzan, M. U. G. Khan, A. Rehmat, S. Iqbal, T. Saba, A. Rehman, and Z. Mehmood. A deep learning approach for automated diagnosis and multi-class classification of alzheimer's disease stages using resting-state fmri and residual neural networks. *Journal of Medical Systems*, 44:37, 12 2019. doi: 10.1007/s10916-019-1475-2.

[18] Z.-Q. Zhao, P. Zheng, S. tao Xu, and X. Wu. Object detection with deep learning: A review, 2018.

[19] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation, 2013.

[20] T. Evgeniou and M. Pontil. Support vector machines: Theory and applications. volume 2049, pages 249–257, 01 2001. doi: 10.1007/3-540-44673-7_12.

[21] R. Girshick. Fast r-cnn, 2015.

[22] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks, 2015.

[23] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection, 2015.

[24] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár. Focal loss for dense object detection, 2017.

[25] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection, 2016.

[26] X. Yang, M. Gao, J. Pu, A. Nayak, Q. Liu, S. E. Bell, J. O. Setter, K. Cao, H. Ha, C. Kozyrakis, and M. Horowitz. Dnn dataflow choice is overrated, 2018.

[27] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry. Accelerating cnn inference on fpgas: A survey, 2018.

[28] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, 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*, page 26–35. ACM, 2016.

[29] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, Jan 2017. ISSN 1558-173X. doi: 10.1109/JSSC.2016.2616357.

[30] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference, 2017.

[31] J. Su. *Artificial Neural Networks Acceleration on Field-Programmable Gate Arrays Considering Model Redundancy*. PhD thesis, Imperial College London, 2018.

[32] J. Wang, Q. Lou, X. Zhang, C. Zhu, Y. Lin, and D. Chen. Design flow of accelerating hybrid extremely low bit-width neural network in embedded fpga, 2018.

[33] A. Pappalardo. Xilinx/brevitas. URL `https://doi.org/10.5281/zenodo.3333552`.

[34] Y. Henon. pytorch-retinanet. URL `https://github.com/yhenon/pytorch-retinanet`. Visited on 28-05-2021.

Appendix A

RetinaNet specifications

Layer		Input				Weight					Bias
Name	Type	X	Y	Z	Total	Nout	Nin	X	Y	Total	Total
Resnet S0 Conv	Conv	640	640	3	1 228 800	64	3	7	7	9 408	0
Resnet S0 Batchnorm	BN	320	320	64	6 553 600	1	1	4	64	256	0
Resnet S0 ReLu	ReLu	320	320	64	6 553 600	0	0	0	0	0	0
Resnet S0 Maxpool	Mpool	320	320	64	6 553 600	0	0	0	0	0	0
Resnet S1 B1 Conv1	Conv	160	160	64	1 638 400	64	64	3	3	36 864	0
Resnet S1 B1 Batchnorm1	BN	160	160	64	1 638 400	1	1	4	64	256	0
Resnet S1 B1 ReLu1	ReLu	160	160	64	1 638 400	0	0	0	0	0	0
Resnet S1 B1 Conv2	Conv	160	160	64	1 638 400	64	64	3	3	36 864	0
Resnet S1 B1 Batchnorm2	Bn	160	160	64	1 638 400	1	1	4	64	256	0
Resnet S1 B1 Sum3D	Sum	160	160	64	1 638 400	0	0	0	0	0	0
Resnet S1 B1 ReLu2	ReLu	160	160	64	1 638 400	0	0	0	0	0	0
Resnet S1 B2 Conv1	Conv	160	160	64	1 638 400	64	64	3	3	36 864	0
Resnet S1 B2 Batchnorm1	BN	160	160	64	1 638 400	1	1	4	64	256	0
Resnet S1 B2 ReLu1	ReLu	160	160	64	1 638 400	0	0	0	0	0	0
Resnet S1 B2 Conv2	Conv	160	160	64	1 638 400	64	64	3	3	36 864	0
Resnet S1 B2 Batchnorm2	BN	160	160	64	1 638 400	1	1	4	64	256	0
Resnet S1 B2 Sum3D	Sum	160	160	64	1 638 400	0	0	0	0	0	0
Resnet S1 B2 ReLu2	ReLu	160	160	64	1 638 400	0	0	0	0	0	0
Resnet S2 B1 Conv1	Conv	160	160	64	1 638 400	128	64	3	3	73 728	0
Resnet S2 B1 Batchnorm1	BN	80	80	128	819 200	1	1	4	128	512	0
Resnet S2 B1 ReLu1	ReLu	80	80	128	819 200	0	0	0	0	0	0
Resnet S2 B1 Conv2	Conv	80	80	128	819 200	128	128	3	3	147 456	0
Resnet S2 B1 Batchnorm2	BN	80	80	128	819 200	1	1	4	128	512	0
Resnet S2 B1 Conv D	Conv	160	160	64	1 638 400	128	64	1	1	8 192	0
Resnet S2 B1 Batchnorm D	BN	80	80	128	819 200	1	1	4	128	512	0
Resnet S2 B1 Sum3D	Sum	80	80	128	819 200	0	0	0	0	0	0
Resnet S2 B1 ReLu2	ReLu	80	80	128	819 200	0	0	0	0	0	0
Resnet S2 B2 Conv1	Conv	80	80	128	819 200	128	128	3	3	147 456	0
Resnet S2 B2 Batchnorm1	BN	80	80	128	819 200	1	1	4	128	512	0
Resnet S2 B2 ReLu1	ReLu	80	80	128	819 200	0	0	0	0	0	0
Resnet S2 B2 Conv2	Conv	80	80	128	819 200	128	128	3	3	147 456	0
Resnet S2 B2 Batchnorm2	BN	80	80	128	819 200	1	1	4	128	512	0
Resnet S2 B2 Sum3D	Sum	80	80	128	819 200	0	0	0	0	0	0
Resnet S2 B2 ReLu2	ReLu	80	80	128	819 200	0	0	0	0	0	0

Table A.1: RetinaNet layer by layer specification part 1.

Layer		Input				Weight					Bias
Name	Type	X	Y	Z	Total	Nout	Nin	X	Y	Total	Total
Resnet S3 B1 Conv1	Conv	80	80	128	819 200	256	128	3	3	294 912	0
Resnet S3 B1 Batchnorm1	BN	40	40	256	409 600	1	1	4	256	1 024	0
Resnet S3 B1 ReLu1	ReLu	40	40	256	409 600	0	0	0	0	0	0
Resnet S3 B1 Conv2	Conv	40	40	256	409 600	256	256	3	3	589 824	0
Resnet S3 B1 Batchnorm2	BN	40	40	256	409 600	1	1	4	256	1 024	0
Resnet S3 B1 Conv D	Conv	80	80	128	819 200	256	128	1	1	32 768	0
Resnet S3 B1 Batchnorm D	BN	40	40	256	409 600	1	1	4	256	1 024	0
Resnet S3 B1 Sum3D	Sum	40	40	256	409 600	0	0	0	0	0	0
Resnet S3 B1 ReLu2	ReLu	40	40	256	409 600	0	0	0	0	0	0
Resnet S3 B2 Conv1	Conv	40	40	256	409 600	256	256	3	3	589 824	0
Resnet S3 B2 Batchnorm1	BN	40	40	256	409 600	1	1	4	256	1 024	0
Resnet S3 B2 ReLu1	ReLu	40	40	256	409 600	0	0	0	0	0	0
Resnet S3 B2 Conv2	Conv	40	40	256	409 600	256	256	3	3	589 824	0
Resnet S3 B2 Batchnorm2	BN	40	40	256	409 600	1	1	4	256	1 024	0
Resnet S3 B2 Sum3D	Sum	40	40	256	409 600	0	0	0	0	0	0
Resnet S3 B2 ReLu2	ReLu	40	40	256	409 600	0	0	0	0	0	0
Resnet S4 B1 Conv1	Conv	40	40	256	409 600	512	256	3	3	1 179 648	0
Resnet S4 B1 Batchnorm1	BN	20	20	512	204 800	1	1	4	512	2 048	0
Resnet S4 B1 ReLu1	ReLu	20	20	512	204 800	0	0	0	0	0	0
Resnet S4 B1 Conv2	Conv	20	20	512	204 800	512	512	3	3	2 359 296	0
Resnet S4 B1 Batchnorm2	BN	20	20	512	204 800	1	1	4	512	2 048	0
Resnet S4 B1 Conv D	Conv	40	40	256	409 600	512	256	1	1	131 072	0
Resnet S4 B1 Batchnorm D	BN	20	20	512	204 800	1	1	4	512	2 048	0
Resnet S4 B1 Sum3D	Sum	20	20	512	204 800	0	0	0	0	0	0
Resnet S4 B1 ReLu2	ReLu	20	20	512	204 800	0	0	0	0	0	0
Resnet S4 B2 Conv1	Conv	20	20	512	204 800	512	512	3	3	2 359 296	0
Resnet S4 B2 Batchnorm1	BN	20	20	512	204 800	1	1	4	512	2 048	0
Resnet S4 B2 ReLu1	ReLu	20	20	512	204 800	0	0	0	0	0	0
Resnet S4 B2 Conv2	Conv	20	20	512	204 800	512	512	3	3	2 359 296	0
Resnet S4 B2 Batchnorm2	BN	20	20	512	204 800	1	1	4	512	2 048	0
Resnet S4 B2 Sum3D	Sum	20	20	512	204 800	0	0	0	0	0	0
Resnet S4 B2 ReLu2	ReLu	20	20	512	204 800	0	0	0	0	0	0
FPN S5 P5 Conv	Conv	20	20	512	204 800	256	512	3	3	1 179 648	256
FPN S6 P6 ReLu	ReLu	10	10	256	25 600	0	0	0	0	0	0
FPN S6 P6 Conv	Conv	10	10	256	25 600	256	256	3	3	589 824	256
FPN S4 M4 Conv	Conv	20	20	512	204 800	256	512	1	1	131 072	256
FPN S4 Upsample	Ups	20	20	256	102 400	0	0	0	0	0	0
FPN S4 P4 Conv	Conv	20	20	256	102 400	256	256	3	3	589 824	256
FPN S3 M3 Conv	Conv	40	40	256	409 600	256	256	1	1	65 536	256
FPN S3 Sum3D	Sum	40	40	256	409 600	0	0	0	0	0	0
FPN S3 Upsample	Ups	40	40	256	409 600	0	0	0	0	0	0
FPN S3 P3 Conv	Conv	40	40	256	409 600	256	256	3	3	589 824	256
FPN S2 M2 Conv	Conv	80	80	128	819 200	256	128	1	1	32 768	256
FPN S2 Sum3D	Sum	80	80	256	1 638 400	0	0	0	0	0	0
FPN S2 P2 Conv	Conv	80	80	256	1 638 400	256	256	3	3	589 824	256

Table A.2: RetinaNet layer by layer specification part 2.

Layer		Input				Weight					Bias
Name	Type	X	Y	Z	Total	Nout	Nin	X	Y	Total	Total
Regression 2 Conv1	Conv	80	80	256	1 638 400	256	256	3	3	589 824	256
Regression 2 ReLu1	ReLu	80	80	256	1 638 400	0	0	0	0	0	0
Regression 2 Conv2	Conv	80	80	256	1 638 400	256	256	3	3	589 824	256
Regression 2 ReLu2	ReLu	80	80	256	1 638 400	0	0	0	0	0	0
Regression 2 Conv3	Conv	80	80	256	1 638 400	256	256	3	3	589 824	256
Regression 2 ReLu3	ReLu	80	80	256	1 638 400	0	0	0	0	0	0
Regression 2 Conv4	Conv	80	80	256	1 638 400	256	256	3	3	589 824	256
Regression 2 ReLu4	ReLu	80	80	256	1 638 400	0	0	0	0	0	0
Regression 2 Conv5	Conv	80	80	256	1 638 400	36	256	3	3	82 944	36
Regression 3 Conv1	Conv	40	40	256	409 600	256	256	3	3	589 824	256
Regression 3 ReLu1	ReLu	40	40	256	409 600	0	0	0	0	0	0
Regression 3 Conv2	Conv	40	40	256	409 600	256	256	3	3	589 824	256
Regression 3 ReLu2	ReLu	40	40	256	409 600	0	0	0	0	0	0
Regression 3 Conv3	Conv	40	40	256	409 600	256	256	3	3	589 824	256
Regression 3 ReLu3	ReLu	40	40	256	409 600	0	0	0	0	0	0
Regression 3 Conv4	Conv	40	40	256	409 600	256	256	3	3	589 824	256
Regression 3 ReLu4	ReLu	40	40	256	409 600	0	0	0	0	0	0
Regression 3 Conv5	Conv	40	40	256	409 600	36	256	3	3	82 944	36
Regression 4 Conv1	Conv	20	20	256	102 400	256	256	3	3	589 824	256
Regression 4 ReLu1	ReLu	20	20	256	102 400	0	0	0	0	0	0
Regression 4 Conv2	Conv	20	20	256	102 400	256	256	3	3	589 824	256
Regression 4 ReLu2	ReLu	20	20	256	102 400	0	0	0	0	0	0
Regression 4 Conv3	Conv	20	20	256	102 400	256	256	3	3	589 824	256
Regression 4 ReLu3	ReLu	20	20	256	102 400	0	0	0	0	0	0
Regression 4 Conv4	Conv	20	20	256	102 400	256	256	3	3	589 824	256
Regression 4 ReLu4	ReLu	20	20	256	102 400	0	0	0	0	0	0
Regression 4 Conv5	Conv	20	20	256	102 400	36	256	3	3	82 944	36
Regression 5 Conv1	Conv	10	10	256	25 600	256	256	3	3	589 824	256
Regression 5 ReLu1	ReLu	10	10	256	25 600	0	0	0	0	0	0
Regression 5 Conv2	Conv	10	10	256	25 600	256	256	3	3	589 824	256
Regression 5 ReLu2	ReLu	10	10	256	25 600	0	0	0	0	0	0
Regression 5 Conv3	Conv	10	10	256	25 600	256	256	3	3	589 824	256
Regression 5 ReLu3	ReLu	10	10	256	25 600	0	0	0	0	0	0
Regression 5 Conv4	Conv	10	10	256	25 600	256	256	3	3	589 824	256
Regression 5 ReLu4	ReLu	10	10	256	25 600	0	0	0	0	0	0
Regression 5 Conv5	Conv	10	10	256	25 600	36	256	3	3	82 944	36
Regression 6 Conv1	Conv	5	5	256	6 400	256	256	3	3	589 824	256
Regression 6 ReLu1	ReLu	5	5	256	6 400	0	0	0	0	0	0
Regression 6 Conv2	Conv	5	5	256	6 400	256	256	3	3	589 824	256
Regression 6 ReLu2	ReLu	5	5	256	6 400	0	0	0	0	0	0
Regression 6 Conv3	Conv	5	5	256	6 400	256	256	3	3	589 824	256
Regression 6 ReLu3	ReLu	5	5	256	6 400	0	0	0	0	0	0
Regression 6 Conv4	Conv	5	5	256	6 400	256	256	3	3	589 824	256
Regression 6 ReLu4	ReLu	5	5	256	6 400	0	0	0	0	0	0
Regression 6 Conv5	Conv	5	5	256	6 400	36	256	3	3	82 944	36

Table A.3: RetinaNet layer by layer specification part 3.

Layer		Input				Weight					Bias
Name	Type	X	Y	Z	Total	Nout	Nin	X	Y	Total	Total
Classification 2 Conv1	Conv	80	80	256	1 638 400	256	256	3	3	589 824	256
Classification 2 ReLu1	ReLu	80	80	256	1 638 400	0	0	0	0	0	0
Classification 2 Conv2	Conv	80	80	256	1 638 400	256	256	3	3	589 824	256
Classification 2 ReLu2	ReLu	80	80	256	1 638 400	0	0	0	0	0	0
Classification 2 Conv3	Conv	80	80	256	1 638 400	256	256	3	3	589 824	256
Classification 2 ReLu3	ReLu	80	80	256	1 638 400	0	0	0	0	0	0
Classification 2 Conv4	Conv	80	80	256	1 638 400	256	256	3	3	589 824	256
Classification 2 ReLu4	ReLu	80	80	256	1 638 400	0	0	0	0	0	0
Classification 2 Conv5	Conv	80	80	256	1 638 400	720	256	3	3	1 658 880	720
Classification 2 Sigmoid	Sig	80	80	720	4 608 000	0	0	0	0	0	0
Classification 3 Conv1	Conv	40	40	256	409 600	256	256	3	3	589 824	256
Classification 3 ReLu1	ReLu	40	40	256	409 600	0	0	0	0	0	0
Classification 3 Conv2	Conv	40	40	256	409 600	256	256	3	3	589 824	256
Classification 3 ReLu2	ReLu	40	40	256	409 600	0	0	0	0	0	0
Classification 3 Conv3	Conv	40	40	256	409 600	256	256	3	3	589 824	256
Classification 3 ReLu3	ReLu	40	40	256	409 600	0	0	0	0	0	0
Classification 3 Conv4	Conv	40	40	256	409 600	256	256	3	3	589 824	256
Classification 3 ReLu4	ReLu	40	40	256	409 600	0	0	0	0	0	0
Classification 3 Conv5	Conv	40	40	256	409 600	720	256	3	3	1 658 880	720
Classification 3 Sigmoid	Sig	40	40	720	1 152 000	0	0	0	0	0	0
Classification 4 Conv1	Conv	20	20	256	102 400	256	256	3	3	589 824	256
Classification 4 ReLu1	ReLu	20	20	256	102 400	0	0	0	0	0	0
Classification 4 Conv2	Conv	20	20	256	102 400	256	256	3	3	589 824	256
Classification 4 ReLu2	ReLu	20	20	256	102 400	0	0	0	0	0	0
Classification 4 Conv3	Conv	20	20	256	102 400	256	256	3	3	589 824	256
Classification 4 ReLu3	ReLu	20	20	256	102 400	0	0	0	0	0	0
Classification 4 Conv4	Conv	20	20	256	102 400	256	256	3	3	589 824	256
Classification 4 ReLu4	ReLu	20	20	256	102 400	0	0	0	0	0	0
Classification 4 Conv5	Conv	20	20	256	102 400	720	256	3	3	1 658 880	720
Classification 4 Sigmoid	Sig	20	20	720	288 000	0	0	0	0	0	0
Classification 5 Conv1	Conv	10	10	256	25 600	256	256	3	3	589 824	256
Classification 5 ReLu1	ReLu	10	10	256	25 600	0	0	0	0	0	0
Classification 5 Conv2	Conv	10	10	256	25 600	256	256	3	3	589 824	256
Classification 5 ReLu2	ReLu	10	10	256	25 600	0	0	0	0	0	0
Classification 5 Conv3	Conv	10	10	256	25 600	256	256	3	3	589 824	256
Classification 5 ReLu3	ReLu	10	10	256	25 600	0	0	0	0	0	0
Classification 5 Conv4	Conv	10	10	256	25 600	256	256	3	3	589 824	256
Classification 5 ReLu4	ReLu	10	10	256	25 600	0	0	0	0	0	0
Classification 5 Conv5	Conv	10	10	256	25 600	720	256	3	3	1 658 880	720
Classification 5 Sigmoid	Sig	10	10	720	72 000	0	0	0	0	0	0
Classification 6 Conv1	Conv	5	5	256	6 400	256	256	3	3	589 824	256
Classification 6 ReLu1	ReLu	5	5	256	6 400	0	0	0	0	0	0
Classification 6 Conv2	Conv	5	5	256	6 400	256	256	3	3	589 824	256
Classification 6 ReLu2	ReLu	5	5	256	6 400	0	0	0	0	0	0
Classification 6 Conv3	Conv	5	5	256	6 400	256	256	3	3	589 824	256
Classification 6 ReLu3	ReLu	5	5	256	6 400	0	0	0	0	0	0
Classification 6 Conv4	Conv	5	5	256	6 400	256	256	3	3	589 824	256
Classification 6 ReLu4	ReLu	5	5	256	6 400	0	0	0	0	0	0
Classification 6 Conv5	Conv	5	5	256	6 400	720	256	3	3	1 658 880	720
Classification 6 Sigmoid	Sig	5	5	720	18 000	0	0	0	0	0	0

Table A.4: RetinaNet layer by layer specification part 4.

Appendix B

RetinaNet detailed time profiling

Stage	Xilinx Zynq-7020 SW only (ARM Cortex A9)			Xilinx Zynq-7020 HW/SW (ARM Cortex A9)			Speedup
	Execution time		Partial execution time	Execution time		Partial execution time	
	s	%	%	s	%	%	
Extra	0.055	-	0.0	0.199	-	0.9	0.28
Regression S2 Conv1	186.270	-	24.4	0.518	-	2.4	359.59
Regression S2 Conv2	186.264	-	24.4	0.519	-	2.5	358.89
Regression S2 Conv3	186.002	-	24.4	0.519	-	2.5	358.39
Regression S2 Conv4	186.274	-	24.4	0.518	-	2.4	359.60
Regression S2 Conv5	18.761	-	2.5	18.897	-	89.3	0.99
Total Regression S2	**763.626**	**31.7**	**100.0**	**21.170**	**4.1**	**100.0**	**36.07**
Extra	0.023	-	0.0	0.033	-	0.6	0.70
Regression S3 Conv1	38.066	-	24.2	0.128	-	2.5	297.39
Regression S3 Conv2	38.123	-	24.3	0.129	-	2.5	295.53
Regression S3 Conv3	38.125	-	24.3	0.128	-	2.5	297.85
Regression S3 Conv4	38.124	-	24.3	0.129	-	2.5	295.53
Regression S3 Conv5	4.541	-	2.9	4.567	-	89.3	0.99
Total Regression S3	**157.002**	**6.5**	**100.0**	**5.114**	**1.0**	**100.0**	**30.70**
Extra	0.005	-	0.0	0.007	-	0.6	0.71
Regression S4 Conv1	9.518	-	24.4	0.032	-	2.8	297.44
Regression S4 Conv2	9.518	-	24.4	0.032	-	2.8	297.44
Regression S4 Conv3	9.519	-	24.4	0.032	-	2.8	297.47
Regression S4 Conv4	9.519	-	24.4	0.032	-	2.8	297.47
Regression S4 Conv5	0.977	-	2.5	0.995	-	88.1	0.98
Total Regression S4	**39.056**	**1.6**	**100.0**	**1.130**	**0.2**	**100.0**	**34.56**
Extra	0.001	-	0.0	0.001	-	0.4	1.00
Regression S5 Conv1	2.259	-	24.3	0.009	-	3.2	251.00
Regression S5 Conv2	2.259	-	24.3	0.008	-	2.8	282.38
Regression S5 Conv3	2.259	-	24.3	0.008	-	2.8	282.38
Regression S5 Conv4	2.259	-	24.3	0.009	-	3.2	251.00
Regression S5 Conv5	0.243	-	2.6	0.247	-	87.6	0.98
Total Regression S5	**9.280**	**0.4**	**100.0**	**0.282**	**0.1**	**100.0**	**32.91**

Table B.1: RetinaNet detailed time profiling part 1.

Stage	Xilinx Zynq-7020 SW only (ARM Cortex A9)			Xilinx Zynq-7020 HW/SW (ARM Cortex A9)			Speedup
	Execution time		Partial execution time	Execution time		Partial execution time	
	s	%	%	s	%	%	
Extra	0.000	-	0.0	0.001	-	1.6	0.00
Regression S6 Conv1	0.543	-	24.4	0.002	-	3.2	271.50
Regression S6 Conv2	0.542	-	24.4	0.003	-	4.8	180.67
Regression S6 Conv3	0.542	-	24.4	0.003	-	4.8	180.67
Regression S6 Conv4	0.544	-	24.5	0.002	-	3.2	272.00
Regression S6 Conv5	0.050	-	2.3	0.051	-	82.3	0.98
Total Regression S6	**2.221**	**0.1**	**100.0**	**0.062**	**0.0**	**100.0**	**35.82**
Extra	0.056	**-**	0.0	0.198	-	0.1	0.28
Classification S2 Conv1	186.046	-	16.6	0.519	-	0.1	358.47
Classification S2 Conv2	186.022	-	16.6	0.519	-	0.1	358.42
Classification S2 Conv3	185.999	-	16.6	0.518	-	0.1	359.07
Classification S2 Conv4	185.999	-	16.6	0.519	-	0.1	358.38
Classification S2 Conv5	373.609	-	33.4	373.834	-	99.4	1.00
Total Classification S2	**1,117.731**	**46.4**	**100.0**	**376.107**	**72.2**	**100.0**	**2.97**
Extra	1.117	-	0.5	1.127	-	1.2	0.99
Classification S3 Conv1	38.066	-	15.6	0.128	-	0.1	297.39
Classification S3 Conv2	38.122	-	15.7	0.129	-	0.1	295.52
Classification S3 Conv3	38.124	-	15.7	0.128	-	0.1	297.84
Classification S3 Conv4	38.125	-	15.7	0.129	-	0.1	295.54
Classification S3 Conv5	89.985	-	36.9	89.297	-	98.2	1.01
Total Classification S3	**243.539**	**10.1**	**100.0**	**90.938**	**17.5**	**100.0**	**2.68**
Extra	0.272	-	0.5	0.273	-	1.4	1.00
Classification S4 Conv1	9.518	-	16.4	0.032	-	0.2	297.44
Classification S4 Conv2	9.519	-	16.4	0.032	-	0.2	297.47
Classification S4 Conv3	9.518	-	16.4	0.032	-	0.2	297.44
Classification S4 Conv4	9.519	-	16.4	0.032	-	0.2	297.47
Classification S4 Conv5	19.528	-	33.7	19.780	-	98.0	0.99
Total Classification S4	**57.874**	**2.4**	**100.0**	**20.181**	**3.9**	**100.0**	**2.87**
Extra	0.065	-	0.5	0.066	-	1.3	0.98
Classification S5 Conv1	2.259	-	16.2	0.008	-	0.2	282.38
Classification S5 Conv2	2.259	-	16.2	0.008	-	0.2	282.38
Classification S5 Conv3	2.259	-	16.2	0.009	-	0.2	251.00
Classification S5 Conv4	2.259	-	16.2	0.008	-	0.2	282.38
Classification S5 Conv5	4.876	-	34.9	4.901	-	98.0	0.99
Total Classification S5	**13.977**	**0.6**	**100.0**	**5.000**	**1.0**	**100.0**	**2.80**
Extra	0.016	-	0.5	0.017	-	1.6	0.94
Classification S6 Conv1	0.544	-	17.1	0.002	-	0.2	272.00
Classification S6 Conv2	0.543	-	17.0	0.003	-	0.3	181.00
Classification S6 Conv3	0.544	-	17.1	0.003	-	0.3	181.33
Classification S6 Conv4	0.542	-	17.0	0.002	-	0.2	271.00
Classification S6 Conv5	0.997	-	31.3	1.027	-	97.4	0.97
Total Classification S6	**3.186**	**0.1**	**100.0**	**1.054**	**0.2**	**100.0**	**3.02**
Total	**2,407.492**	**100.0**	**-**	**521.038**	-	**100.0**	**4.62**

Table B.2: RetinaNet detailed time profiling part 2.