TÉCNICO LISBOA

Object Detection and Classification on the Versat Reconfigurable Processor

Daniel Garigali Pestana

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor(s): Prof. Horácio Cláudio De Campos Neto
Prof. José João Henriques Teixeira de Sousa

Examination Committee

Chairperson: Prof. Francisco André Corrêa Alegria
Supervisor: Prof. José João Henriques Teixeira de Sousa
Member of the Committee: Prof. Mário Pereira Vestias

January 2021

Declaration

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

Resumo

O objectivo principal deste trabalho é o desenvolvimento do VersatCNN, um módulo IP baseado na arquitectura de matriz reconfigurável de grão grosso Versat e estendido para a computação eficiente de redes neuronais convolucionais (CNNs). O VersatCNN é validado através da implementação de um detector de objectos de estado de arte. O VersatCNN é composto por unidades de multiplicação-acumulação (MAC) embebidas em unidades vectoriais e organizadas numa matriz para explorar reuso dos dados e paralelismo ao nível da convolução e mapa de características. Unidades paralelas de leitura e escrita comunicam com a memória externa através dum barramento amplo de controlo de memória. As unidades de computação reconfiguráveis formam diferentes caminhos de dados para acelerar várias camadas das CNNs e funções de ativação. O detector de objectos é o YOLOv3-Tiny, uma versão leve do YOLOv3 para sistemas embebidos. Neste trabalho, o código-fonte é convertido para vírgula fixa e optimizado para aceleração em hardware através da aproximação das funções de activação e quantização dinâmica pós-treino. A queda de precisão é de apenas 2,1 usando a métrica da precisão média (mAP), quando comparada ao modelo original de virgula flutuante. O software optimizado do YOLOv3-Tiny é executado num CPU RISC-V de baixo desempenho, acelerado pelo VersatCNN e prototipado numa UltraScale XCKU040, atingindo um desempenho de 32,4 imagens por segundo, funcionando a 143 MHz para imagens de tamanho 768x576 e um factor de paralelismo de 832 (número de MACs).

Palavras-chave: Matriz Reconfigurável de Grão Grosso, Redes Neuronais Convolucionais, YOLOv3-Tiny, Versat, CPU RISC-V

Abstract

The main goal of this work is the development of VersatCNN, an IP core based on the Versat Coarse-Grained Reconfigurable Array, extended to efficiently compute Convolutional Neural Networks (CNNs). VersatCNN is validated with the deployment of a state-of-art object detector. VersatCNN is composed of a large number of Multiply-Accumulate (MAC) units embedded in vector units, organized in a matrix structure to exploit parallelism at the convolution and feature map levels and to enhance data sharing. Parallel memory read and write units exchange data with the external memory over a wide memory controller bus. The reconfigurable computing units form different datapaths for accelerating different CNN layers and activation functions. The state-of-art object detector used is YOLOv3-Tiny, a lightweight version of the YOLOv3 detector targeting embedded systems, which has the best trade-off between accuracy and execution time. In this work, the source code is converted to fixed-point and optimized for hardware acceleration using approximated activation functions, batch-normalization folding and post-training dynamic quantization. The precision drop is only 2.1 using the Mean Average Precision (mAP) metric, when compared to the original floating-point model. The YOLOv3-Tiny detector, running the optimized software on a minimal and low performance RISC-V CPU and using the VersatCNN IP core for acceleration, is prototyped in a UltraScale XCKU040 FPGA and achieves a performance of 32.4 frames per second, running at 143 MHz for 768x576 sized images and a parallelism factor of 832 (number of MAC units).

Keywords: Coarse Grained Reconfigurable Array, Convolutional Neural Networks, Versat Reconfigurable Processor, YOLOv3-Tiny, RISC-V CPU

Contents

List of Tables

List of Figures

List of Acronyms

AGU Address Generator Unit

ALU Arithmetic Logic Unit

AP Average Precision

API Application Programming Interface

AXI Advanced eXtensible Interface

BRAM Block RAM

CGRA Coarse Grained Reconfigurable Array

CNN Convolutional Neural Network

COCO Common Objects In Context

CPI Cycles per Instruction

CPU Central Processing Unit

DDR Double Data Rate

DMA Direct Memory Access

DNN Deep Neural Network

DSSD Deconvolutional SSD

DSP Digital Signal Processing

FF Flip-Flop

FM Feature Map

FPGA Field Programmable Gate Array

FPN Feature Pyramid Network

FPS Frames Per Second

FU Functional Unit

GPU Graphics Processing Unit

IOb-SoC IObundle System-On-Chip

IoU Intersection over Union

IP Intellectual Property

LRU Least Recently Used

LUT Look-Up Table

MAC Multiply-Accumulate

MulAdd Multiplier and Accumulator

mAP Mean Average Precision

NMS Non-Maximum Suppression

R-CNN Region Based CNN

ReLU Rectified Linear Unit

R-FCN Region-based Fully Convolutional Network

RGB Red Green Blue

RAM Random-Access Memory

ROM Read Only Memory

SRAM Static RAM

SSD Single Shot Detector

PE Processing Element

UART Universal Asynchronous Receiver-Transmitter

VOC Visual Object Classes

YOLO You Only Look Once

WNS Worst Negative Slack

WHS Worst Hold Slack

Chapter 1

Introduction

Object detectors have a wide range of application fields such as security (e.g., face detection), transportation (e.g., autonomous driving), military (e.g., aircraft detection) and medical (e.g., computer aided diagnosis) [1]. Their task consists in classifying and locating multiple objects in an image from predefined categories. General-purpose object detectors focus on detecting a broad range of natural categories from highly structured objects (e.g., car, bicycle) to articulated objects (e.g., person, dog), rather than specific categories (e.g., faces) [2]. The deployment of a real-time general-purpose object detector for resource-constrained embedded systems is a major and current challenge.

This thesis presents a new extension of the Versat reconfigurable processor, which consists of a hardware architecture aimed for low-end devices and extended for the acceleration of the computation of CNNs, which are the basis of the state-of-art object detectors. The system is validated with the deployment of a state-of-art object detector that maximizes performance for a target frame rate of 30 Frames Per Second (FPS) with reduced hardware size and cost.

1.1 Motivation

Object detection has been under extensive research in both academia [1–3] and real world applications [4, 5]. Traditional approaches were based on handcrafted low-level features and shallow trainable architectures. Their performance were limited due to the difficulty of manually designing a robust feature extractor and combining low-level features with high-level context from classifiers [6].

Recent technological breakthroughs led to the fast evolution of object detectors. The main contributions include the development of Deep Neural Networks (DNNs) and the increase of the hardware computing power. State-of-art object detectors use DNNs with deeper architectures to learn more complex features without the need to design them manually.

The superior accuracy of DNNs comes at the cost of high computational complexity with tens of millions of parameters and billions of operations (i.e., additions and multiplications). Therefore, these networks require parallel computation, high data reusability and large memory bandwidth [7]. Graphics Processing Units (GPUs) have been the most common programmable accelerators for deploying DNNs

due to their high parallelization and high-speed floating point computing power. GPUs use large-size batch to perform parallelization, which requires the simultaneous input of several images. For real-time applications with the need to process images frame by frame, this strategy is not viable due to the considerable latency of each frame. Moreover, GPUs cannot be deployed in embedded systems as a result of their high power consumption.

Recent studies [8–11] have been using Field Programmable Gate Arrays (FPGAs) as a more energy-efficient alternative to GPUs for deploying DNNs. FPGAs present advantages in terms of high flexibility to design application-specific hardware, fixed-point calculation, parallel computing and low power consumption. The dataflow is a main concern when designing these programmable accelerators. Accelerators based on Coarse Grained Reconfigurable Arrays (CGRAs) for DNNs have also been further investigated [12, 13]. A CGRA is a programmable hardware circuit from the same family of the FPGAs but with a lighter configuration infrastructure, resulting in less silicon area and lower cost.

1.2 Objectives

The Deep Versat CGRA [14] is a configurable and customisable hardware accelerator developed for speeding-up loop-based applications. Although highly scalable, this reconfigurable processor presents limitations for the acceleration of CNNs. Thus, the main focus of this work is the development of a new hardware accelerator named VersatCNN, which is based on the former Deep Versat but suitable for the computation of CNNs. The improvements include the addition of a Direct Memory Access (DMA) module for fast data transfers, the deployment of vector Functional Units (FUs) for shared configurations between the same type of FUs, the implementation of automatic ping-pong memories and the introduction of heterogeneous stages.

The second main goal is to validate and demonstrate the VersatCNN Intellectual Property (IP) core by the deployment and acceleration of an object detector for an ambitious performance of at least 30 FPS. As a result, the source code of the original floating-point model of the detector is firstly reduced for its application on an resource-constrained embedded system and then simplified for hardware computation, which englobes post-training fixed-point quantization and approximation of activation functions. The software baseline is implemented on top of the IObundle System-On-Chip (IOb-SoC) platform, which is based on a RISC-V soft-processor. This thesis also shows how to configure the internal architecture of the IP core taking into account the characteristics of the CNN to be accelerated and the target performance.

This thesis was developed in cooperation with IObundle, a Lisbon-based computer architecture company, for further investigation and commercialization of the designed IP core. Given the foreseen magnitude of the development of a hardware accelerator and its configuration for the deployment of an object detector, this work was implemented alongside another thesis [15], which focused more on the integration of both software baseline and VersatCNN IP core in the IOb-SoC platform, besides of the software code for configuration of the IP core with the purpose of accelerating an object detection algorithm.

1.3 Outline

This document is organized as follows. Chapter 2 introduces DNNs and shows how they are implemented in FPGAs. In Chapter 3, the current state-of-art for object detection is analyzed. Based on that analysis, one object detector is selected and fully described. Chapter 4 introduces the concept of CGRA and explains the Deep Versat CGRA architecture, remarking its limitations for CNN acceleration. Chapter 5 describes the architecture of the IP core developed to accelerate CNNs, which is inspired on the Deep Versat CGRA. In Chapter 6, the software-only version of the object detector chosen is implemented in the RISC-V soft-processor to be used as baseline for the hardware acceleration and the IP core is parameterized and tested for accelerating the object detector. Chapter 7 presents the performance results of the final solution in terms of the resource consumption, the execution time of the detector and the comparison with other FPGA-based works. Finally, Chapter 8 concludes the work and highlights the major achievements and suggestions for future work.

Chapter 2

Deep Neural Networks

This chapter introduces the concept of DNN and proceeds to CNNs, a common form of DNN widely used in image processing applications such as object detection. The evolution of CNNs and the associated popular models are further discussed. FPGAs have recently been used to accelerate CNNs by developing dedicated hardware and memory systems. Thus, this chapter also focuses on the strategies implemented in previous works to accelerate CNNs in FPGAs.

2.1 Neural networks

The neuron is the main computational unit of neural networks and is implemented as the weighted sum of the inputs plus a constant, followed by an activation function, as represented in Figure 2.1.

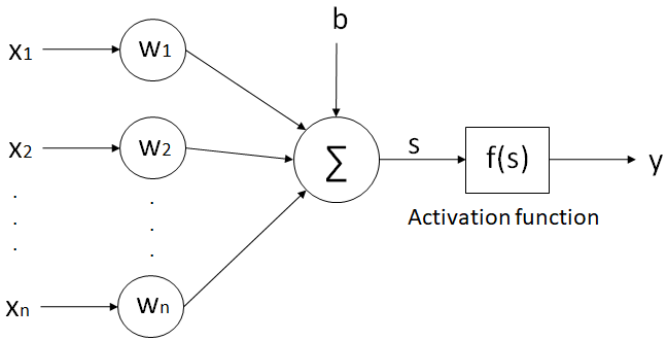

Figure 2.1: Implementation of a neuron.

The weight, w_i, expresses the influence of a given input, x_i, to the neuron's output and the bias, b, is an offset term used to shift the result of the activation function, f, towards the negative or positive side. Eq. 2.1 computes the output, y, of a neuron.

$$y = f\left(b + \sum_{i=1}^{n} x_i w_i\right) \tag{2.1}$$

Conventional activation functions include the identity function, the sigmoid (i.e., logistic function) and the hyperbolic tangent. Recently, the Rectified Linear Unit (ReLU) and some variations (e.g., Leaky

ReLU) have become popular due to their simplicity and fast convergence during training [9]. These activation functions are represented in Figure 2.2.

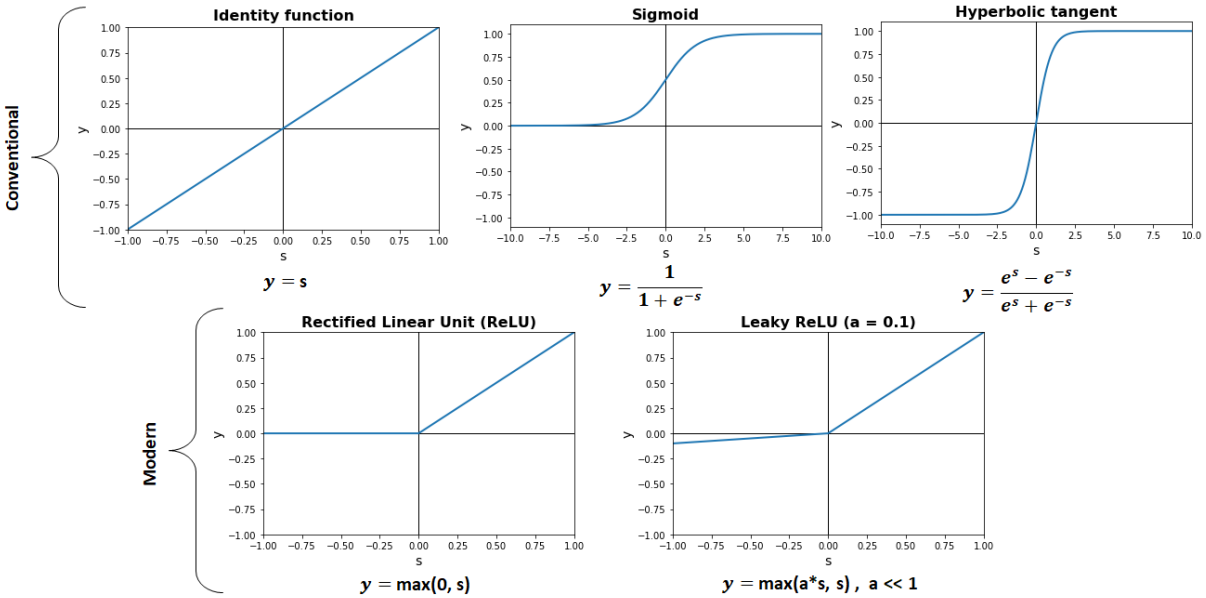

Figure 2.2: Plot and expression of the most common activation functions.

Neural networks are composed of neurons organized in layers: one input layer, one or more hidden layers and one output layer. A DNN is a neural network with more than one hidden layer. Figure 2.3 exemplifies a fully connected (i.e., each neuron from one layer is connected to every neuron of the next layer) DNN with 2 hidden layers.

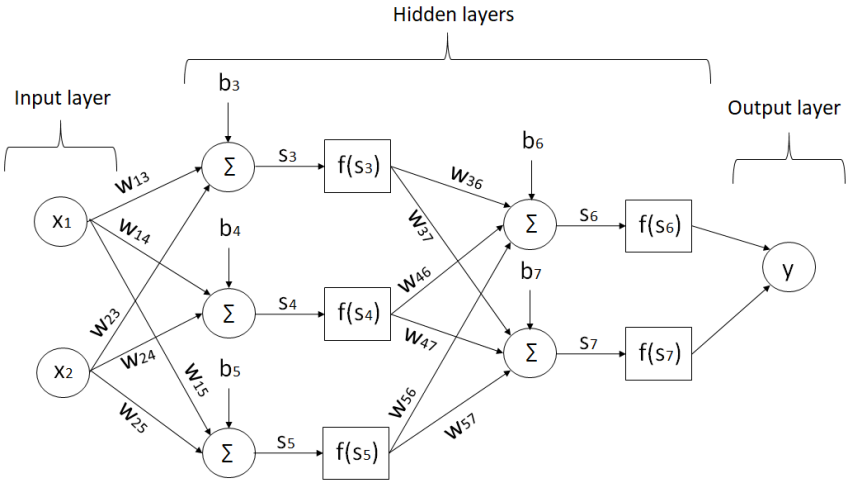

Figure 2.3: Example of a fully connected DNN.

By using more layers, DNNs can learn more complex high-level features. For instance, in a typical image processing application, the input layer receives the normalized pixels (i.e., scaled between 0 and 1) from an image which then go through the first hidden layer. This layer outputs several low-level features (e.g., lines and edges). In the next hidden layers, these features are combined to form higher level features (e.g., contours, shapes). Hence, each layer builds on the features detected in previous layers. The output layer predicts if all those features describe a certain object or scene [9]. This process

is an example of **inference**, which consists in making predictions over the input data by using a learned model. This work focuses on the inference for embedded systems, thus, a pre-trained DNN is used.

2.2 Convolutional Neural Networks

Fully connected DNNs are more difficult to train as the network gets deeper due to the increasing number of connections and weights. CNNs are DNNs characterized by the presence of convolutional layers. The neurons of a convolutional layer only connect to sub-regions of the previous layer, instead of being fully connected, allowing to build deeper networks and therefore achieve superior performance.

CNNs are implemented as a sequence of interconnected layers and consist in two stages: feature extraction and classification, as shown in Figure 2.4. The stages are based on convolutional, pooling and fully connected layers. For feature extraction, the network is built on repeated blocks, each composed of a convolutional layer, an optional batch-normalization layer, a non-linear layer (i.e., application of an activation function) and an optional pooling layer. For classification purposes, fully connected layers, optionally followed by a regression function, are typically applied after the last block of the feature extraction stage. Modern CNN models add other type of layers such as shortcut, route and upsample layers.

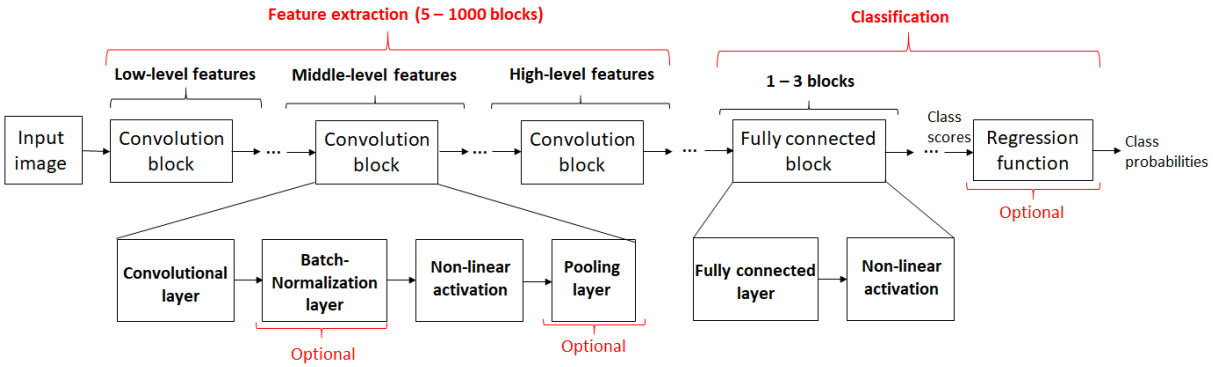

Figure 2.4: Constitution of a typical CNN.

2.2.1 Convolutional layer

Convolutional layers perform 3D convolutions, which can be seen as a set of 2D convolutions. In 2D convolutions, a 2D kernel is overlapped and shifted as a sliding window throughout the entire 2D input image (known as input feature map), generating a 2D output image (called output feature map). In each overlap, a MAC operation is performed. Padding, which consists in adding new elements around the edges of the input Feature Map (FM), allows the output to keep the same size as the input. Normally zero-padding is applied. Figure 2.5 exemplifies a 2D convolution between an 5x5 input feature map and 3x3 kernel with zero padding. Note how the weights that compose the kernel are shared during the process. In this example, the step size (also known as stride) used when shifting the kernel throughout the input feature map is 1.

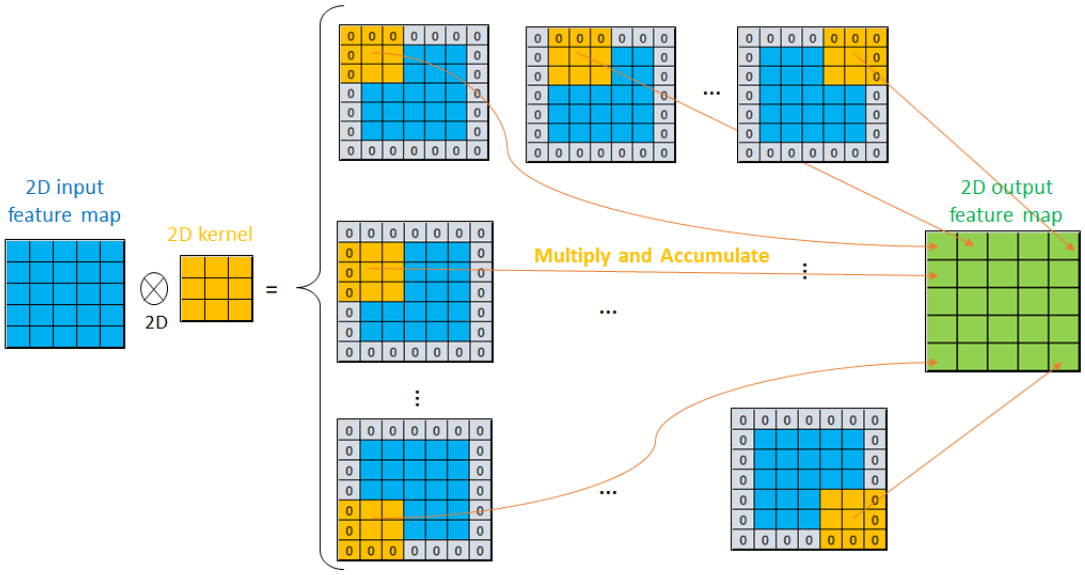

Figure 2.5: Example of an 5x5 input feature map and 3x3 kernel 2D convolution.

The input of convolutional layers is a set of 2D feature maps (each one is called a channel) and another set of 3D kernels, with each 3D kernel having the same number of 2D channels. For each 3D kernel, there is a 2D convolution between each channel of the input feature map and each channel of the given 3D kernel. The results of the convolutions are summed across all the channels. The output feature map is obtained after summing the former result with a shared bias associated to each 3D kernel. Therefore, one output feature map is created for each 3D kernel. Figure 2.6 exemplifies a 3D convolution between an 5x5 input feature map and two 3x3 kernels, all with 3 channels and zero-padding.

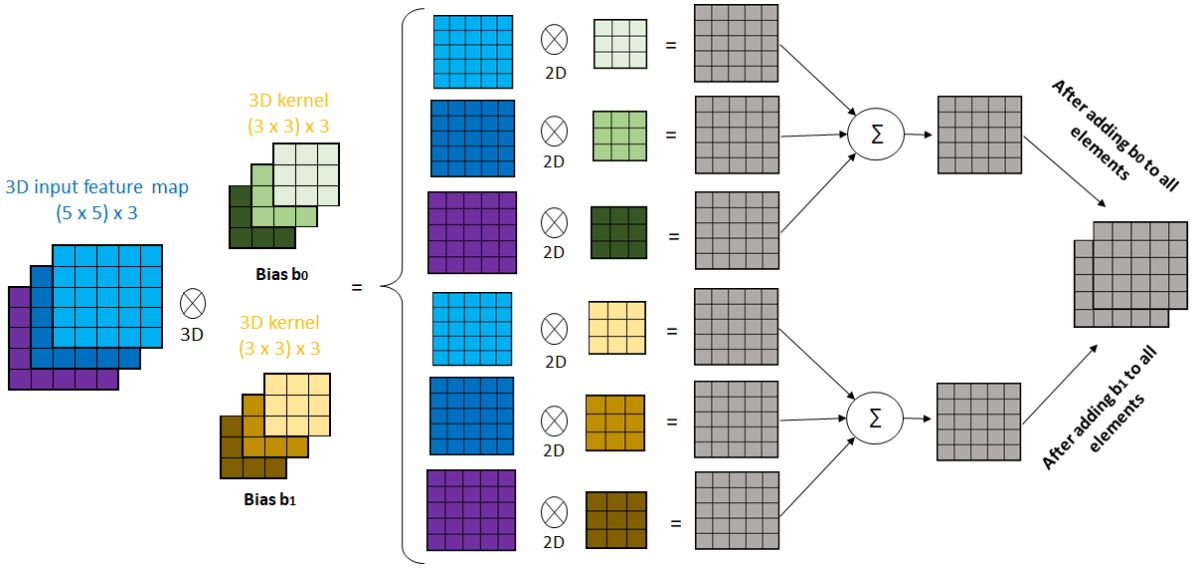

Figure 2.6: Example of an 5x5 input feature map and two 3x3 kernels 3D convolution (3 channels).

2.2.2 Batch-Normalization layer

The batch-normalization layer is used for speeding up the training by normalizing the input data (i.e., zero mean and unit standard deviation) [10]. Furthermore, the normalized value is scaled and shifted.

Eq. 2.2 expresses the computation performed by this layer for each input element, x, where the mean, μ, and the variance, σ^2, are statistics collected from training and the scale factor, γ, and the shift factor, β, are parameters learned during training. ϵ is a small constant that avoids dividing by zero. When using this layer, the bias can be included in the shift factor instead of being computed in the convolutional layer.

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta \tag{2.2}$$

In inference, the values of μ, σ^2, γ and β are known. Thus, Eq. 2.2 can be reformulated as one multiplication and one addition, as shown in Eq. 2.3, where γ_i and β_i are the new scale and shift factors.

$$y = x \times \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} + \left(-\frac{\mu \times \gamma}{\sqrt{\sigma^2 + \epsilon}} + \beta\right) = x \times \gamma_i + \beta_i \tag{2.3}$$

2.2.3 Pooling layer

The pooling layer downsamples the feature maps, leading to a reduction of the number of inputs in the next layers. Each 2D channel is divided into blocks, which are further replaced by the maximum (maxpooling) or the mean (average pooling) value of the block. The most common operation is a 2x2 maxpooling, as shown in Figure 2.7. Some CNNs, instead of using pooling layers, simply apply a stride of 2 in the convolutional layers. However, pooling layers are more robust as they turn the network invariant to small shifts and distortions when downsampling [9].

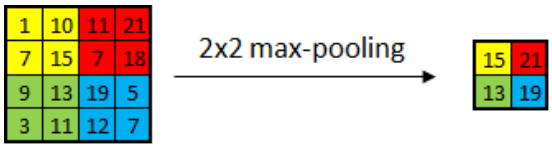

Figure 2.7: Example of 2x2 max-pooling.

2.2.4 Shortcut layer

The shortcut layer skips one or more layers by adding the output of a former layer to the input of the current layer. Figure 2.8 exemplifies a shortcut layer generated from adding the output from 2 layers before.

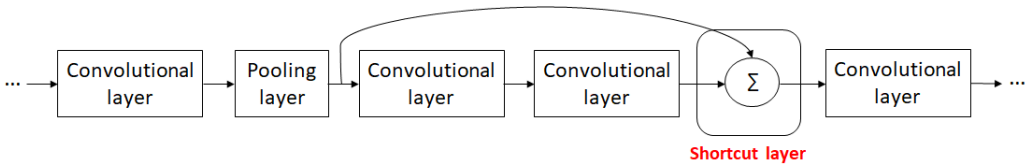

Figure 2.8: Example of a shortcut layer.

2.2.5 Route and upsample layers

Route and upsample layers were introduced for CNNs focused on object detection tasks [16]. The route layer concatenates the output from a former layer with the input of the current layer by stacking them into

different channels. For example, routing a (26x26)x256 feature map with a (26x26)x128 feature map results in a (26x26)x384 feature map. This allows the detection of fine grained features, improving the localization of small objects.

The upsample layer upsamples a feature map, typically by a factor of two, which allows to detect objects at different scales and obtain more meaningful semantic information from the features. Figure 2.9 exemplifies the simplest way to upsample a 2x2 feature map by a factor of two.

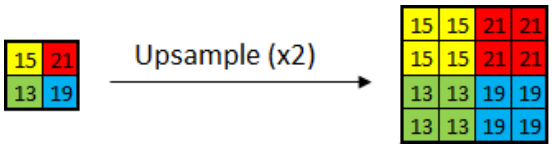

Figure 2.9: Example of upsampling by a factor of 2.

2.3 Training neural networks

The training of neural networks consists in finding the parameters (i.e., weights and bias) that maximize the score of the correct prediction and minimize the scores of the incorrect predictions. For that, there is a training dataset containing inputs of the network (e.g., an image) and the desired output (e.g., label). In order to obtain the parameters, the gradient descent method [9] is used. This iterative algorithm updates the parameters by computing the partial derivatives of the gradient, which indicates how the parameters should change to reduce the difference between the desired outputs and the actual predictions (also known as loss). To efficiently compute the partial derivatives, the loss is propagated backwards through the network, which is known as the backpropagation algorithm [9].

Often, the parameters are initialized randomly for the first iteration of the gradient descent, which could result in slow training. To speed-up the training and achieve better performance, fine-tuning can be applied. Fine-tuning consists in using previously-trained parameters as a starting point to adjust the data to a new dataset or a new constraint.

There are several frameworks for training and testing DNN models. The most popular are Caffe, Tensorflow, Torch [9] and Darknet [17]. The popular CNN models that will be mentioned in the next section were developed for image classification. The most popular datasets for this task are MNIST (digit classification), CIFAR and ImageNet [9].

2.4 Popular CNN models

AlexNet [9] was one of the first CNN-based models for image classification, followed by VGG-16 [9]. Both are based on the architecture presented in Figure 2.4. They mainly differ in the number of layers and in the number and size of the kernels. A more distinct model is GoogLeNet [9]. Different sized filters are convoluted in parallel for the same input feature map and the results are further concatenated. Therefore, the input is processed at multiple scales. The other difference is the use of 1x1 kernels to reduce the number of channels and, consequently, the number of weights.

ResNet [9] was the first CNN that exceeded the human-level accuracy for image classification by deploying a deeper network than the above-mentioned models. Those models suffered from the vanishing gradient problem, restraining them from getting deeper. When training, after several multiplications, the gradient becomes infinitely small during backpropagation, affecting the update of the weights in early layers for very deep networks. To avoid that, shortcut layers were added in the network.

Darknet-53 [16] is a more recent CNN model that also employs the shorcut layers first introduced by ResNet to allow a deeper network.

2.5 CNN acceleration with FPGAs

Several studies have been conducted for accelerating CNNs in FPGAs [8–11]. The main computation in CNNs is the MAC operation which mostly occurs in the convolutional layers, as shown in Table 2.1. Consequently, more than 90% of the inference execution time is typically spent in the computation of the convolutional layers [10]. Therefore, accelerators are focused on speeding-up these layers.

Table 2.1: Layer constitution of some popular DNN models (adapted from [10]).

Type of layer	Characteristic	AlexNet	VGG-16	GoogLeNet	ResNet-152
Convolutional	# Layers	5	13	57	155
	# MACs	666M	15.3G	1.58G	11.3G
	#Parameters	2.33M	14.7M	5.97M	58M
Pooling	# Layers	3	5	14	2
Fully Connected	# Layers	3	3	1	1
	# MACs	58.6M	124M	1.02M	2.05M
	#Parameters	58.6M	124M	1.02M	2.05M

The most common approaches for accelerating CNN inference in FPGAs in previous works are mainly focused on optimizing the accelerator by developing dedicated hardware and memory systems in order to exploit the parallelism of the MAC operations and to enhance data reuse. Approximating the model specifically for computation in FPGAs is another method that allows to reduce the amount of operations and storage requirements.

2.5.1 Accelerator optimization

One of the main advantages of accelerating CNNs in FPGAs, rather than in Central Processing Units (CPUs) or GPUs, is the flexibility to design customized hardware to exploit different sources of parallelism and dedicated caches to support data reuse. As shown in Table 2.1, as networks get deeper, the number of operations and the storage requirements increase. Thus, the use of external memory is required, whose access results in high latency and significant energy consumption. FPGAs present a density of hard-wired Digital Signal Processing (DSP) blocks and a collection of on-chip memories that can be used for performing the MAC operations and reducing the number of external memory accesses, respectively.

Typical FPGA-based CNN accelerators [18–20] introduce several levels of memory hierarchy, as shown in Figure 2.10. The system is composed of two on-chip input buffers, one for fetching the feature maps and the other for fetching the parameters from the external memory through the DMA. The data is streamed into configurable Processing Elements (PEs), which are responsible for computing the MAC

operations. Each PE has its own on-chip registers. The on-chip output buffer stores the intermediate results and output feature maps, which are transferred back, if needed, to the external memory. The CPU issues the workload to the controller, which in turn generates control signals to the other modules. The multipliers and adder trees present in each PE are usually pipelined in order to reduce the critical path of the circuit and increase the throughput.

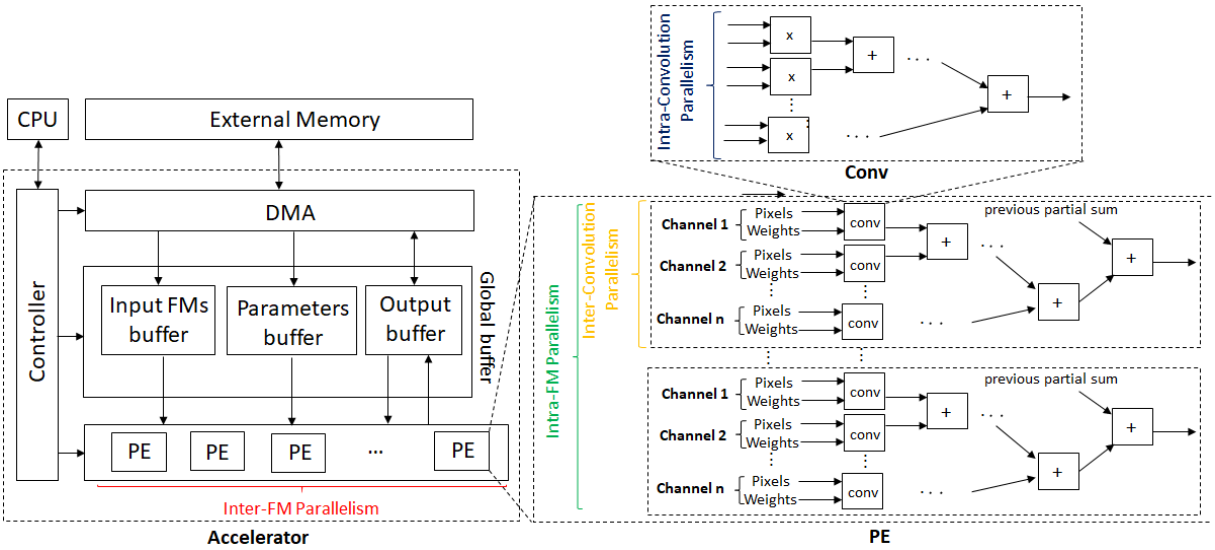

Figure 2.10: Typical FPGA-based CNN accelerator (adapted from [10]).

Figure 2.10 also shows four sources of concurrency [10] when computing each convolutional layer:

1) Intra-Convolution Parallelism: multiplications in 2D convolutions are implemented concurrently.
2) Inter-Convolution Parallelism: multiple 2D convolutions are computed concurrently.
3) Intra-FM Parallelism: multiple pixels of a single output FM are processed concurrently.
4) Inter-FM Parallelism: each output FM is processed separately in a different PE.

The computation of each convolutional layer can be seen as the application of four nested loops. Each loop is associated to a source of parallelism, as represented in Figure 2.11.

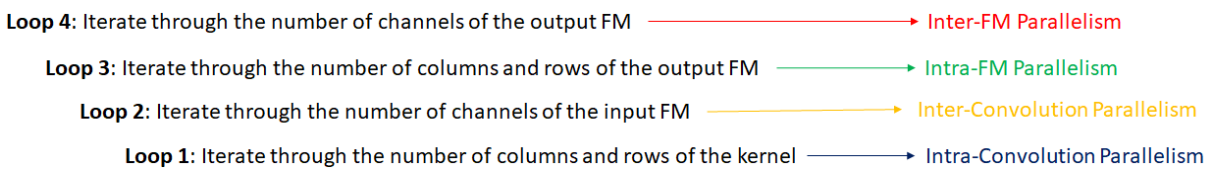

Figure 2.11: Association between loops and source of parallelism.

The sources of parallelism to be exploited depend on the characteristics of the convolutional layers (e.g., number of channels, input feature map size) and the FPGA resources. The architectural configuration of the PEs (i.e., number of MACs and registers) and the data temporal scheduling are defined by applying loop optimization techniques such as loop unrolling, loop tiling and loop interchange [10]. Examples of previous works that implement these techniques are analyzed in Section 2.5.3.

Loop unrolling consists in accelerating the execution of the loops at the expense of resource utilization. Each loop has an unroll factor that indicates how many times the respective loop is parallelized.

Taking into account the loop enumeration in Figure 2.11, the unroll factor for loop 4 determines the number of PEs while the unroll factors for the remaining loops determine the number of multipliers, adders and registers of each PE. The total number of multipliers is given by the product of the four unroll factors. The unroll factors must be carefully chosen, otherwise, they could lead to underutilization of the hardware. For instance, for any given loop, if the number of iterations is not divisible by the respective unrolling factor, then, the utilization ratio is less than 1 [11]. Figure 2.12 shows that the weights and the pixels can be reused by unrolling loops 3 and 4 respectively [8].

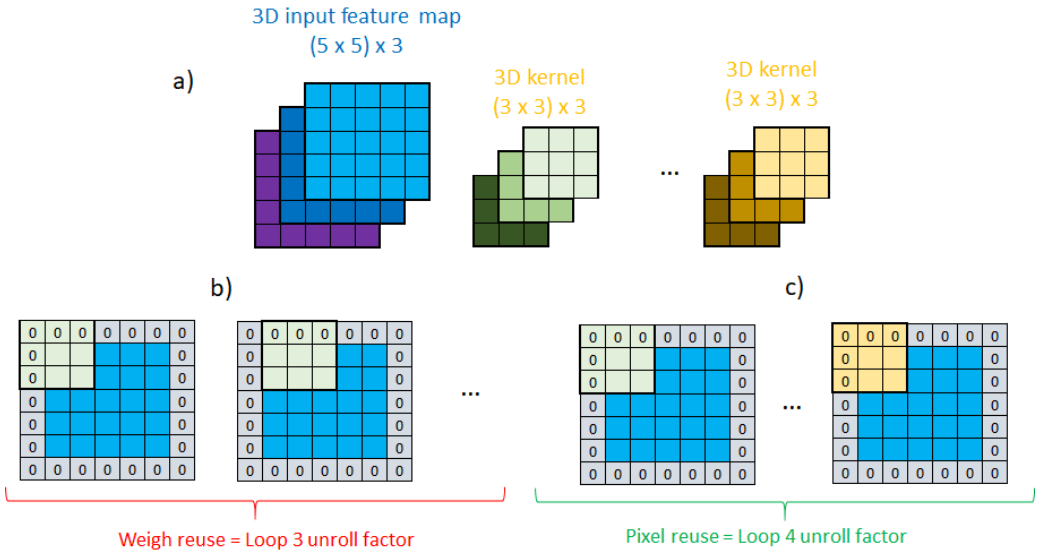

Figure 2.12: a) 3D convolution between one FM and several kernels with b) weight and c) pixel reuse.

Loop tiling is a higher level of loop unrolling that divides the data into multiple blocks that fit into the on-chip buffers, increasing the data locality. Each tiled loop is splitted into two loops: one for iterating inside each tile (intra-tiling) and the other for iterating over the tiles (inter-tiling). Each loop has a tiling factor that indicates how many iterations are performed inside the respective tile. The tiling factors determine the size of the input and output buffers. If the tiling factors can cover all pixels and weights for loops 1 and 2, the partial sums (between channels) are stored in the local registers [8]. Otherwise, the partial sums of one tile must be stored in the output buffer (intermediate result) until is used by the next tile [8]. Figure 2.13 exemplifies the division of the feature maps and kernels when tiling all loops.

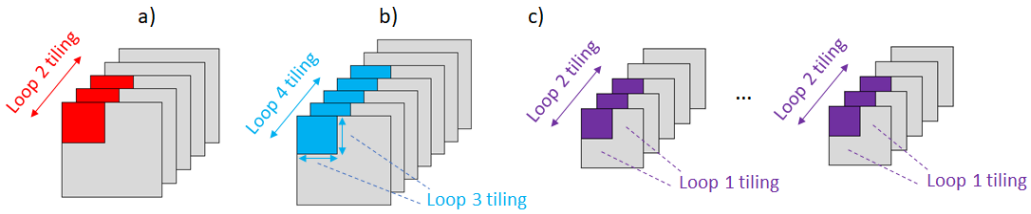

Figure 2.13: Example of loop tiling in the a) input FMs, b) output FMs and c) kernels.

The **loop interchange** strategy decides the execution order of the loops. For intra-tiling loops, the order determines the data being transferred from the on-chip buffers to the PEs. For inter-tiling loops, the order indicates the movement of data from the external memory to the on-chip buffers. The storage

required for intermediate results also depends on the order of the execution of the loops. For instance, the earlier loops 1 and 2 are computed, the fewer are the number of partial sums [8].

2.5.2 Model approximation

The CNN execution can be accelerated by approximating the computation at the cost of minimal accuracy drop. Two of the most common strategies are reducing the precision and the number of operations. During training, the data is typically in single-precision floating-point format (32 bits). For inference in FPGAs, the feature maps and kernels can be converted to fixed-point format with less precision (typically 8 or 16 bits), reducing the storage requirements, hardware utilization and power consumption [10].

The reduction of the FPGA resources consumption when using fixed-point representation is clear in Table 2.2, especially for lower precision data. The DSP consumption is hardly benefited when using narrower bit-width than 16 for fixed-point. Figure 2.14 exemplifies the conversion of a 32-bit floating-point value to fixed-point with 8 bits for the decimal part and 8 bits for the fractional part (Q8.8).

Table 2.2: Operation resource consumption for different operand sizes (adapted from [11]).

Data type	Multiplier		Adder		Multiplier and Adder		
	LUT	FF	LUT	FF	LUT	FF	DSP
floating-point (32 bits)	708	858	430	749	800	1284	2
floating-point (16 bits)	221	303	211	337	451	686	1
fixed-point (32 bits)	1112	1143	32	32	111	64	4
fixed-point (16 bits)	289	301	16	16	0	0	1
fixed-point (8 bits)	75	80	8	8	0	0	1
fixed-point (4 bits)	17	20	4	4	0	0	1

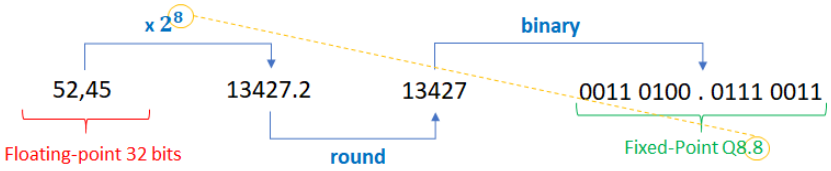

Figure 2.14: Example of conversion from 32 bits floating-point to Q8.8 fixed-point.

In general, the feature maps of deeper layers tend to present a larger numerical range than the ones from initial layers and the weights also tend to be much smaller than the pixels of feature maps [11]. To prevent overflow, the precision must be increased for intermediate results. Thus, different precision values are normally used for weights, intermediate results and output feature maps from different layers.

In order to reduce the number of operations, the most common approaches are weight pruning and low rank approximation [10]. These methods are often followed by a fine-tuning phase to counterbalance the accuracy drop. As this work is focused on the inference part, these methods are not further studied.

2.5.3 Comparison of FPGA-based CNN accelerators

To choose the unroll and tiling factors that maximize both computational throughput and resource utilization, besides minimizing the number of memory accesses, previous works performed a brute force exploration of the design space [10]. This design space exploration consists in testing several factors for different loops in order to exploit various parallelism and data reuse patterns.

[19] was one of the first works to employ loop optimization strategies to accelerate the execution of AlexNet in a FPGA. By unrolling loops 2 and 4, the accelerator achieved a performance of 61.62 GOPs, relying on 32-bit floating point arithmetic. This work also introduced the utilization of double buffers to perform ping-pong operations, allowing to simultaneously compute and transfer data.

Greater acceleration can be achieved when using loop optimization techniques alongside fixed-point arithmetic. For instance, [20] reached a performance of 187.24 GOPs for the same AlexNet network by unrolling loops 1, 2 and 4, relying on 16-bit fixed-point arithmetic. In general, unrolling loop 1 does not provide enough parallelism as the kernels are usually small (e.g. 3x3). Unrolling loop 1 also increases control complexity when layers present different kernel sizes.

The same unrolling scheme and fixed point arithmetic was followed by [18] for the VGG-16 network, achieving a performance of 136.97 GOPs. In all these approaches, the loops are unrolled in the same way they are tilled [10]. In [8], the tiling factors are set in a way all the data required to compute an element from the output feature map is fully buffered. As a result, intermediate results can be stored in the PE registers instead of in the output buffer. This accelerator outperforms all previous implementations by reusing pixels and weights when unrolling loops 3 and 4, reaching a total performance of 645.25 GOPs.

Table 2.3 compares the four FPGA-based accelerators in terms of resource consumption, optimization strategy and performance. All these approaches unroll loop 4 by employing several PEs. [19] has the major DSP consumption mainly due to using 32-bit floating point operands. [8] presents 3 times higher performance than [18] but consumes the double of resources in terms of DSPs and on-chip memory (BRAMs). The analysis of previous works regarding FPGA-based CNN acceleration allows to infer that it is essential to choose optimal unroll and tiling factors, which in turn depend on the characteristics of the convolutional layers and the available resources of the FPGA.

Table 2.3: Comparison between different FPGA-based accelerators.

Network	AlexNet [19]	AlexNet [20]	VGG-16 [18]	VGG-16 [8]
Device	Virtex VX485T	Stratix5 GSD8	Zynq XC7Z045	Arria-10 GX 1150
Frequency (MHz)	100	120	150	150
# Operations (GOP)	1.3	1.3	30.76	30.95
# Weights (M)	2.3	2.3	50.18	138.3
LUT (K)	186	138	183	161
BRAM	512 (36 kB)	—	486 (36 kB)	1900 (20 kB)
DSP	2240	635	780	1518
Performance (GOPs)	61.62	126.6	136.97	645.25
Unrolled loops	2,4	1,2,4	1,2,4	3,4
Precision	Float 32	Fixed 16	Fixed 16	Fixed 8-16

2.6 Final remarks

CNNs are composed of a sequence of interconnected layers, being the convolutional ones the most time consuming for inference execution. FPGAs, with dedicated hardware and cache memories, allows to exploit parallelism and data reuse. Previous works use fixed-point arithmetic and perform design space exploration to obtain the best loop optimization parameters. Similar strategies will be conducted to accelerate the convolutional layers of the object detection network chosen in the next chapter.

Chapter 3

Object Detection State-of-Art

This chapter studies the current CNN-based state-of-art object detectors, alongside the benchmarks and metrics used for their evaluation. Based on this study, one object detector is chosen and described.

The task of a general-purpose object detector is to locate and classify existing objects in an image from predefined categories. The most common way to label and output the coordinates of the located object is to draw a bounding box around it, as represented in Figure 3.1. State-of-art object detectors are DNN-based and their backbone network for feature extraction consists (or is inspired) in the networks for image classification mentioned in Section 2.4, excluding the last fully connected layers [1].

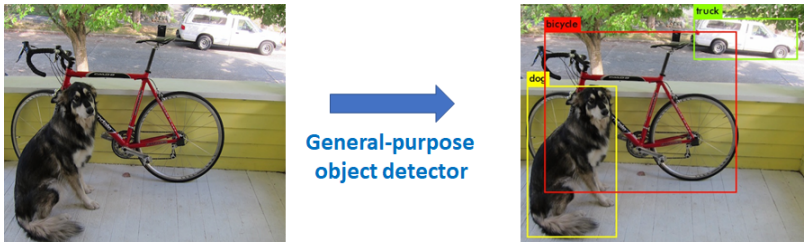

Figure 3.1: Example of bounding box usage to locate objects in an image (adapted from [21]).

3.1 Benchmarks and metrics

Two of the most common benchmarks for general-purpose object detection are PASCAL Visual Object Classes (VOC) 2007/2012 and Microsoft Common Objects In Context (COCO) [3]. The official metric for measuring the performance of detectors is based on the mAP with small variations between both benchmarks. This metric compares the predicted bounding boxes and labels with the ground truth data, which provides the true labels of each object in an image including the class and the coordinates of the true bounding box.

Object detection is simultaneously a regression (object bounding box) and a classification (object class) task. The process for calculating the mAP metric is summed up in Figure 3.2. Usually, the model outputs more boxes than actual objects, which indicates the presence of boxes with low confidence. Therefore, the first step is to label the predicted bounding boxes as true or false detections with respect to the ground truth bounding boxes. The Intersection over Union (IoU) is an evaluation metric that

measures the accuracy of the localization task by calculating the ratio of the area of overlap and the area of union between the predicted and the ground truth bounding boxes. The predicted bounding box is considered a true detection if its IoU score is above a given IoU threshold, otherwise, it is a false detection. Duplicated bounding boxes and wrong classifications are also false detections.

The Average Precision (AP) of each class is calculated based on the precision and recall metrics. The precision measures the ratio of the true detections and the total number of objects detected [4]. The recall measures the ratio of the true detections and the total number of objects in the dataset [4]. A high precision indicates that it is likely that a true detection is, in fact, a correct detection while a high recall means that the detector will positively detect all objects in the dataset.

Each bounding box has a score associated which indicates how likely that box contains an object. To calculate the AP of each class, the precision-recall curve is computed from the detections of the model by varying the score threshold. The AP corresponds to the area under that curve and is typically computed by numerical integration. After the AP of all classes is calculated, the mAP is determined as the average of all the APs, resulting in a value between 0 and 100%. Therefore, the mAP metric allows to evaluate both classification and localization and is designed to penalize the algorithms for missing object instances, for duplicate detections of one instance, for false positive detections and for specializing in some classes, resulting in worse performances in other classes [1].

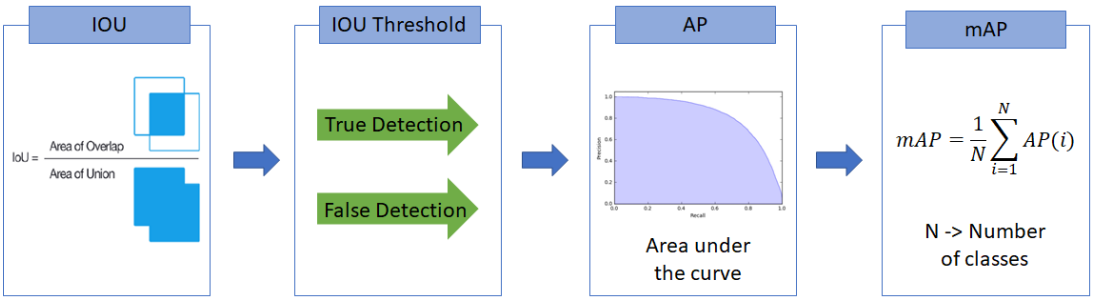

Figure 3.2: Steps for obtaining the mean average precision (adapted from [4]).

The PASCAL VOC datasets contain 20 object categories (e.g., person, bicycle, dog) spread over 11k images, from which over 27k object instances are labeled with bounding boxes [1]. This benchmark considers only one IoU threshold of 0.5 to obtain the mAP. On the other hand, the COCO dataset is composed of 300k images with an average of 7 object instances per image from a total of 80 categories [3]. This benchmark considers ten IoU thresholds (from 0.5 to 0.95 with an interval of 0.05) and the mAP is obtained by averaging the mAPs calculated for each IoU threshold. Considering several IoU thresholds tends to reward models that are better at precise localization and penalizes the algorithms with a high number of bounding boxes with wrong classifications.

3.2 Comparison between object detectors

Several studies have been conducted for comparing the performance of the state-of-art object detectors [1–3]. Object detectors can be divided into two categories: two-stage detectors (region proposal based) and one-stage detector (regression/classification based).

Two-stage detectors follow the traditional object detection pipeline by firstly scanning the whole scenario and then focusing on regions of interest. Thus, the first stage consists in generating region proposals (i.e., candidate bounding boxes). In the second stage, features are extracted from each candidate box in order to perform the classification and bounding box regression tasks. The most popular two-stage detectors are Region Based CNN (R-CNN), Fast R-CNN, Faster R-CNN, Mask R-CNN and Region-based Fully Convolutional Network (R-FCN) [1, 3, 7].

One-stage detectors treat object detection as a regression/classification problem by adopting a unified framework to obtain the labels and locations directly. These detectors map straightly from image pixels to bounding box coordinates and class probabilities by proposing predicted boxes directly from input images without the region proposal step. The most common one-stage detectors are You Only Look Once (YOLO) and its successors YOLOv2 and YOLOv3, Single Shot Detector (SSD) and its successor Deconvolutional SSD (DSSD) and RetinaNet [1, 3, 7].

The selection between one-stage and two-stage detectors resides on a choice between speed and accuracy. Two-stage detectors present higher localization and object recognition accuracy while one-stage detectors achieve higher inference speed. Table 3.1 summarizes the mAP metric for both PASCAL VOC 2007 and COCO datasets and the inference time of each of the above-mentioned object detectors. For the COCO dataset, besides the official mAP metric (i.e., obtained from ten IoU thresholds), the mAP with only one IoU of 0.5 is also shown.

Table 3.1: Comparison of the performance of several object detectors.

Type	Detector	PASCAL VOC07	COCO		Inference time		Backbone	Hardware
		mAP	mAP	mAP_{50}	ms	FPS		
Two-stage	R-CNN [7]	66	—	—	10000	0.1	AlexNet	Titax X GPU
	Fast R-CNN [1, 7]	70	19.7	35.9	2000	0.5	VGG16	
	Faster R-CNN [1, 7]	73.2	21.9	42.7	167	6	VGG16	
	R-FCN [1, 3]	83.6	29.9	51.9	170	5.9	ResNet-101	
	Mask R-CNN [1, 7]	—	39.8	62.3	303	3.3	ResNeXt-101	
One-stage	YOLO [7]	63.4	—	—	22	45	GoogLeNet	
	YOLOv2-544 [1, 3]	73.4	21.6	44	25	40	DarkNet-19	
	SSD-300 [1, 3]	74.3	23.2	41.2	21.7	46	VGG-16	
	SSD-512 [1, 3]	76.8	26.8	46.5	52.6	19		
	SSD-321 [1, 16]	—	28	45.4	61	16.4	ResNet-101	
	SSD-513 [1, 7, 16]	76.8	31.2	50.4	125	8		
	DSSD-321 [1, 3, 16]	78.6	28	46.1	85	11.8		
	DSSD-513 [1, 16]	—	33.2	53.3	156	6.4		
	YOLOv3-320 [16]	—	28.3	51.5	22	45.5	DarkNet-53	
	YOLOv3-416 [16]	—	31	55.3	29	34.5		
	YOLOv3-608 [16]	—	33	57.9	51	19.6		
	RetinaNet-500 [1, 16]	—	34.4	53.1	90	11.1	ResNet-101	M40 GPU
	RetinaNet-800 [1, 16]	—	37.8	57.5	198	5		

The best performance for both PASCAL VOC and COCO datasets are achieved by two-stage detectors, namely R-FCN and Mask R-CNN. Higher frame rates are achievable with one-stage detectors such as YOLO (and its successors) and one version of the SSD detector. There are several versions available for the same detectors which mainly differ in the size of the input feature maps of the first layer. However, the topology of the network is the same for any version. For instance, YOLOv3 has three versions with input feature maps of 320x320, 416x416 or 608x608. Bigger input feature maps tend to lead to higher accuracy but lower speed. In the case of the SSD detector, some versions use VGG-16 which results in accelerated performance and others use ResNet-101 for higher accuracy. Within these

detectors, YOLOv3 is the one that presents the best trade-off between accuracy and execution time (for the 320 and 416 versions). This is the object detector chosen for this work.

3.3 YOLOv3 detector

Figure 3.3 exemplifies the process flow of the YOLOv3 detector for an input feature map of 416x416. The input image is resized at the beginning of the process as the detector allows different input resolutions. The YOLOv3 network block is responsible for extracting features using the Darknet-53 backbone and for returning candidate bounding boxes from those features for three different scales (52x52, 26x26 and 13x13). Candidate bounding boxes are then filtered based on their objectness score and the score of each class. Finally, non-maximum suppression is used to remove multiple detections of the same object and the final detections (bounding boxes and class labels) are drawn over the original input image.

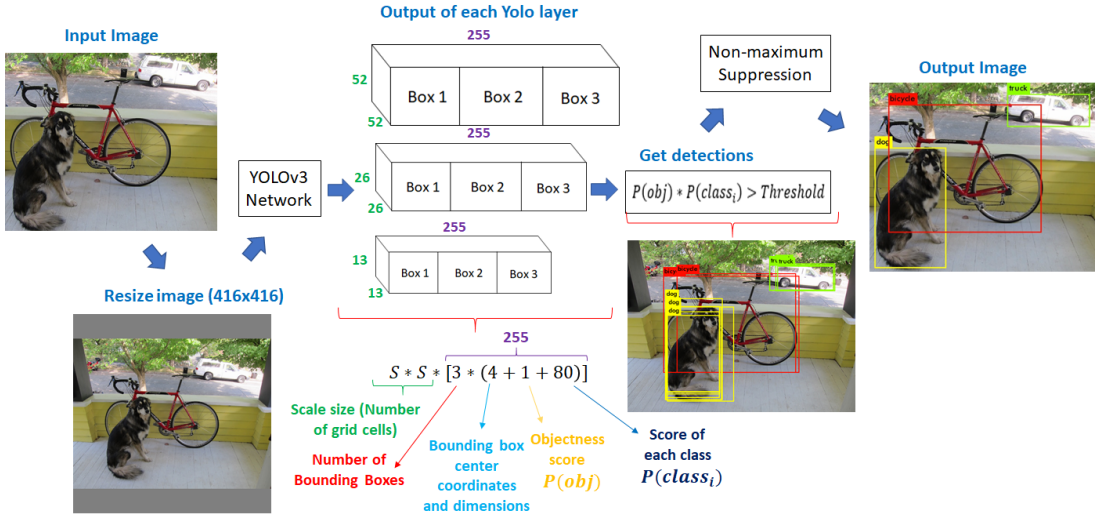

Figure 3.3: YOLOv3 process flow.

3.3.1 Image resizing (pre-CNN)

This method scales the input image to the size of the first layer of the YOLO network by maintaining its aspect ratio and adding padding until meeting the desired dimensions. To keep the aspect ratio, the maximum dimension (width or height) of the input image is scaled to the desired dimension of the resized image, while the other dimension is scaled proportionally. For instance, for an 768x576 input image, the width (768) is resized to 416 and the height (576) is resized to 312 (576 * 416/768). The remaining space (416-312=104) is padded with grey as shown in Figure 3.4.

The resize method is based on a bilinear interpolation. For each unknown pixel of the resized image, this method considers the closest 2x2 neighbourhood of known pixels from the input image surrounding the unknown pixel and interpolates its final value by calculating the weighted average of the 4 known pixels. The procedure is expressed in Algorithm 1 and can be divided in 3 steps. First, the pre-processing stage determines the indexes and factors that are related to each possible pixel of the resized image. In other words, the positions and weights of the closest 2x2 pixels are determined for each unknown pixel based on the dimensions of both input and resized images.

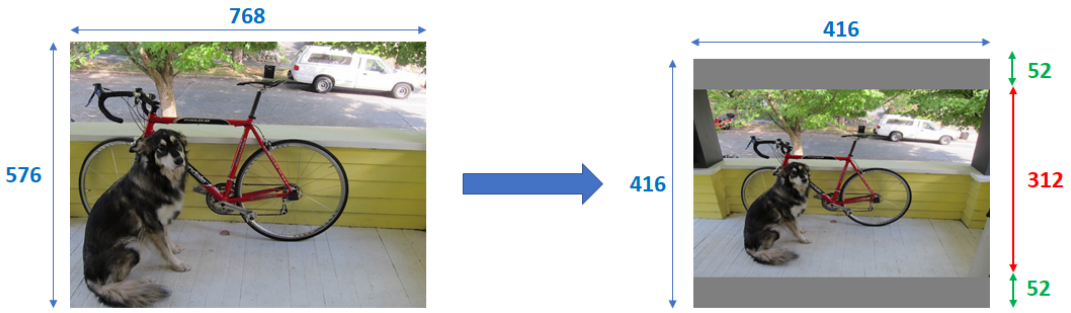

Figure 3.4: Example of image resizing.

Furthermore, the width of the input image is resized by calculating the weighted sum of two of the closest pixels based on the horizontal indexes and factors for each unknown pixel. This second step generates an intermediate image with the desired width of the resized image while keeping the height of the input image. For instance, an 768x576 input image is converted into an 416x576 intermediate image. The final step resizes the height of the image by determining the weighted sum of the other two closest pixels based on the vertical indexes and factors for each unknown pixel. Finally, this converts an 416x576 intermediate image to the desired 416x312 resized image. Note that the resizing of both width and height are done for all the three channels of the input image.

Algorithm 1: Image resizing

Input image dimensions (IMG_W, IMG_H); Resized image dimensions (NEW_W, NEW_H)

1) Pre-processing:

- Width and height scales:

$$w\_scale = \frac{IMG\_W - 1}{NEW\_W - 1} \qquad h\_scale = \frac{IMG\_H - 1}{NEW\_H - 1}$$

- Horizontal index and factor:

for *i* = 1,...,*NEW_W* **do**
*w_pos[i] = i * w_scale* **(integer part)**
*w_factor[i] = i * w_scale* **(fractional part)**
end for

- Vertical index and factor:

for *i* = 1,...,*NEW_H* **do**
*h_pos[i] = i * h_scale* **(integer part)**
*h_factor[i] = i * h_scale* **(fractional part)**
end for

2) Width resize:

for *j* = 1,...,*IMG_H* **do**
for *i* = 1,...,*NEW_W* **do**
*interm_im[i, j] = (1 - w_factor[i]) * input_im[w_pos[i], j]*
*interm_im[i, j] += w_factor[i] * input_im[w_pos[i+1], j]*
end for
end for

3) Height resize:

for *j* = 1,...,*NEW_H* **do**
for *i* = 1,...,*NEW_W* **do**
*output_im[i, j] = (1 - h_factor[j]) * interm_im[i, h_pos[j]]*
*output_im[i, j] += h_factor[j] * interm_im[i, h_pos[j+1]]*
end for
end for

3.3.2 YOLOv3 network

The CNN-based YOLOv3 network is represented in Figure 3.5. This network is composed of 75 convolutional layers, 23 shortcut layers, 3 yolo layers, 2 upsample layers and 4 route layers, making a total of 107 layers. The two route layers after the yolo layers only copy the output of a former layer without concatenating with the output of the previous layer. All convolutional layers include batch-normalization and use Leaky ReLU (with a = 0.1) as activation function, except from the convolutional layer exactly before of each yolo layer, which does not include batch-normalization and uses a linear activation function. There are no fully connected layers and convolutional layers with stride 2 are used instead of maxpool layers. One can also observe that every time the feature map is downsampled by a factor of four, the depth (i.e., number of channels) is duplicated. 3x3 convolutions are done with zero-padding in order to keep the same size between input and output feature maps.

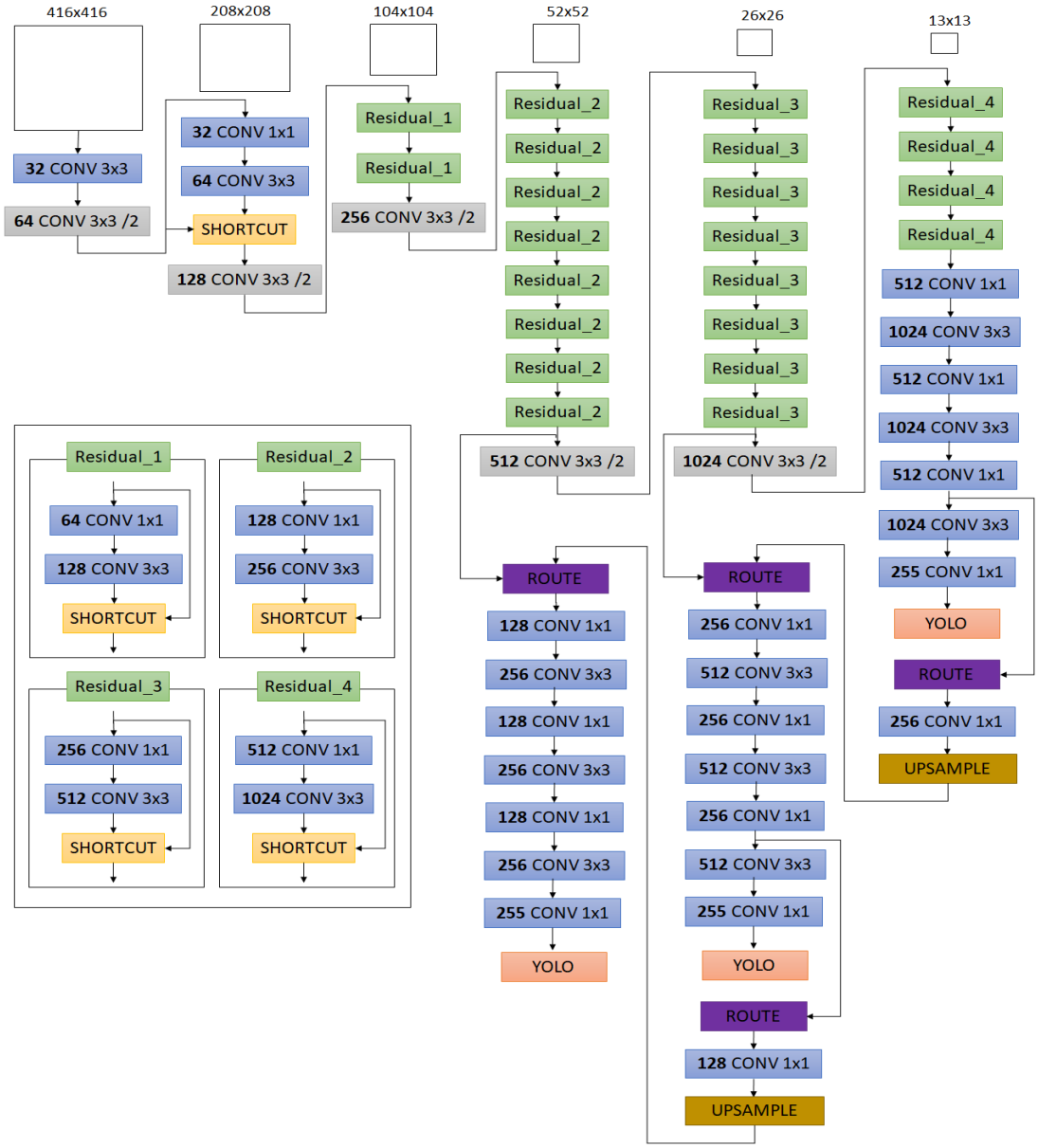

Figure 3.5: YOLOv3 Network.

This network is inspired on some concepts from popular DNN models. For instance, the kernels are 3x3 and 1x1 to reduce the number of weights as first introduced by GoogLeNet. A ResNet-alike structure is followed through the shortcut layers, thereby enhancing feature learning in deep networks. As the network goes deeper, due to downsampling the feature maps, small objects are difficult to detect. Therefore, objects of different sizes are detected with different feature map scales through a structure similar to the Feature Pyramid Network (FPN) [3]. The FPN, represented in Figure 3.6, is used for multi-scale feature learning and consists in merging, through the route layers, upsampled feature maps from deeper layers with feature maps of the same spatial size from early stages in order to capture both low and high-level information from objects [3]. YOLOv3 uses three scales: 52x52 to detect small objects, 26x26 to detect medium objects and 13x13 to detect big objects.

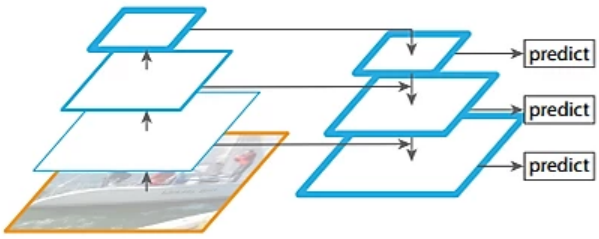

Figure 3.6: Feature Pyramid Network structure [3].

For each scale, the feature map is divided into a grid where each grid cell predicts 3 bounding boxes. The initial size of each bounding box (known as prior box) was set after using K-mean clustering in the training dataset [16]. The sizes are then appropriately adjusted by the network. Each bounding box consists in the predictions of [21]: i) the center of the box relative to the grid cell bounds (2 coordinates); ii) the width and height of the box relative to the whole image; iii) the objectness (or confidence) score which indicates how likely the box contains an object and how accurate are its dimensions regarding to the ground truth box; and iv) the conditional probability of every class given there is an object. As the YOLOv3 network is trained over the COCO dataset [16], the total number of classes is 80. Thus, each grid cell is composed of $3 * (2 + 2 + 1 + 80) = 255$ values, as specified in Figure 3.7.

The logistic activation is used to constrain the center coordinates of the box to fall in the range of the grid cell (i.e., between 0 and 1). The objectness score is predicted using logistic regression (1 means perfect overlapping between predicted and ground truth boxes while 0 means no overlapping). For the class predictions, independent logic classifiers are also used. The **yolo layers** apply the logistic activation in the predictions of each bounding box, excluding the width and height parameters. This new layer added by the YOLOv3 network is used as the ending layer for each scale.

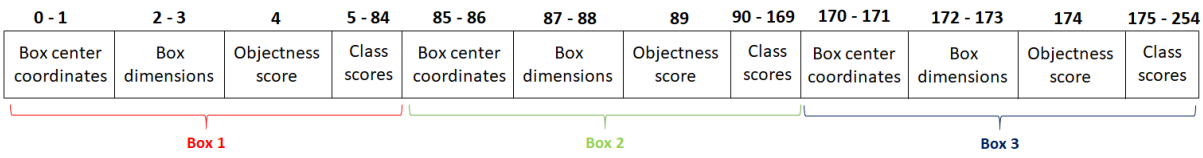

Figure 3.7: Constitution of each grid cell.

3.3.3 Detections phase (post-CNN)

After executing the YOLOv3 network block (Figure 3.3), there are several candidate bounding boxes for each scale, however, only a few of them correspond to true detections (depending on the number of objects in the image). The true detections correspond to the bounding boxes whose product between the objectness score and the conditional probability of each class is above a given threshold (the default value is 0.5). The bounding boxes can be multilabeled, i.e., can have more than one class.

Due to detecting objects with different scales and with three bounding boxes per grid cell, multiple bounding boxes of the same object might be found. The Non-Maximum Suppression (NMS) is used to remove these multiple detections and consists in the following algorithm [21]:

Algorithm 2: Non-maximum suppresion

1) Select bounding box with the highest confidence score
2) Calculate the IoU between selected box and all the remaining boxes
3) Discard boxes whose IoU is greater than a certain threshold (default value is 0.45)
4) Repeat steps 2-4 for the next highest score box until processing all remaining boxes

After applying the non-maximum suppression algorithm, the post-processing phase of the detector ends with the drawing of the bounding boxes and respective class labels over the original input image.

3.4 YOLOv3-Tiny network

The YOLOv3 network comprises a total of 62 million parameters. The author of this detector [16] proposed an alternative network for constrained environments called YOLOv3-Tiny which in turn has a total of 8.8 million parameters. This smaller model presents a mAP (for one IoU of 0.5) of 32.9 in the COCO dataset and runs at a frame rate of 65 FPS in the RTX 2080 Ti GPU. Thus, this version is faster but also less accurate than the YOLOv3 network. The YOLOv3-Tiny network is the CNN to be accelerated in the scope of this work as the object detector to be deployed is intended for embedded systems.

As represented in Figure 3.8, the YOLOv3-Tiny is composed of 13 convolutional layers, 6 maxpool layers, 2 route layers, 2 yolo layers and 1 upsample layer. In comparison with YOLOv3, this network uses maxpooling instead of convolutions with stride 2 to downsample the feature maps. Besides that, objects are detected with only 2 scales (26x26 and 13x13). No shortcut layers are used. Note that the last maxpool layer has a stride of 1, hence, the feature map keeps its resolution in this layer. To achieve that, the last column and last line of the feature map are duplicated to form an input 14x14 feature map.

Table 3.3 summarises the characteristics of each layer of the YOLOv3-Tiny network in terms of the kernel size, the bias, the resolution of the feature maps and the activation function. Note that layers 1 to 13 are responsible for feature extraction and layers 14 to 24 are responsible for object detection.

The execution time of the CNN mainly depends on the number of MAC operations of the convolutional layers. As the stride of these layers in the YOLOv3-Tiny network is always one, the number of MAC operations of each layer can be determined by the following expression:

$$\#\ \texttt{MAC operations} = \texttt{num\_ker} \times \texttt{ker\_w} \times \texttt{ker\_h} \times \texttt{w} \times \texttt{h} \times \texttt{c} \tag{3.1}$$

where num_ker is the number of kernels, ker_w and ker_h are the kernel dimensions, w and h are the input FM dimensions and c is the number of input channels. As shown in Table 3.2, the convolutional layers of the YOLOv3-Tiny CNN present a total of 2.78 GMAC operations.

Table 3.2: Number of MAC operations of the convolutional layers of the YOLOv3-Tiny CNN.

Layer	N.º Kernels	Kernel Size	Input FM	N.º Input Channels	MMAC
1	16	3x3	416x416	3	75
3	32	3x3	208x208	16	199
5	64	3x3	104x104	32	199
7	128	3x3	52x52	64	199
9	256	3x3	26x26	128	199
11	512	3x3	13x13	256	199
13	1024	3x3	13x13	512	797
14	256	1x1	13x13	1024	44
15	512	3x3	13x13	256	199
16	255	1x1	13x13	512	22
19	128	1x1	13x13	256	6
22	256	3x3	26x26	384	598
23	255	1x1	26x26	256	44
Total (GMAC)					2.78

3.5 Final remarks

The backbone for feature extraction of the state-of-art object detectors are based on the popular CNNs introduced in the previous chapter. PASCAL VOC and COCO are the most common benchmarks for the evaluation of object detectors where YOLOv3 presents the best trade-off between accuracy and execution time. For this work, the lighter YOLOv3-Tiny network will be accelerated using an architecture based on the one introduced in the next chapter.

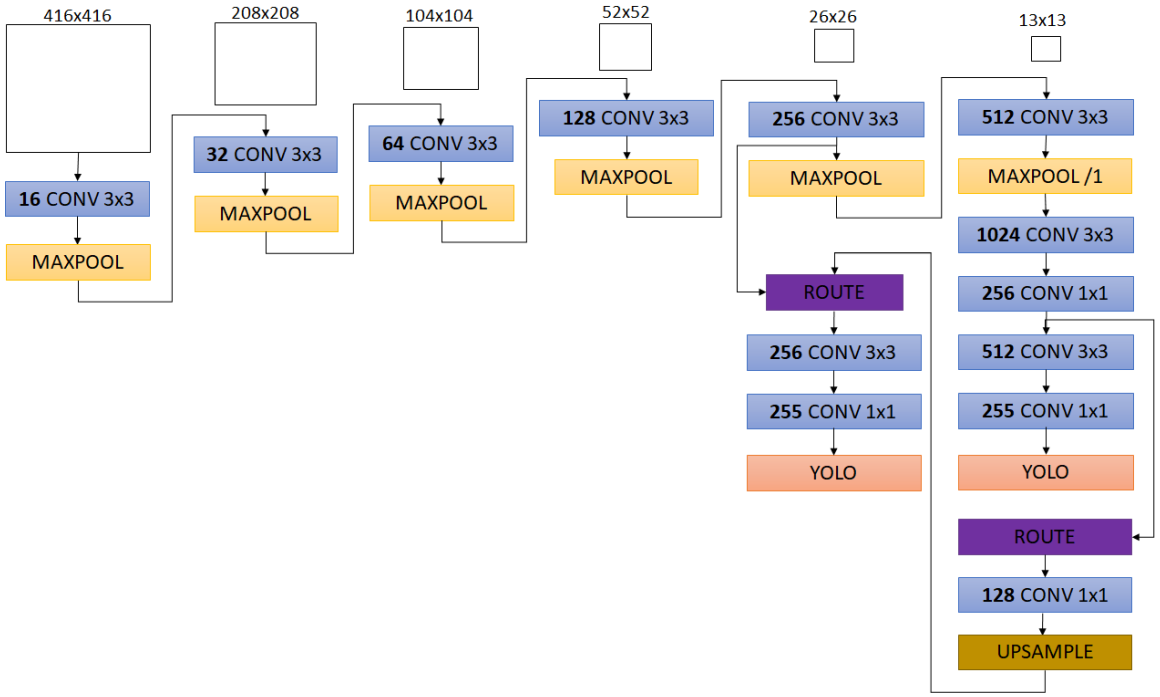

Figure 3.8: YOLOv3-Tiny Network.

Table 3.3: Characteristics of the YOLOv3-Tiny layers.

Layer	Type	Kernel	Bias	Input FM	Output FM	Activation Function
1	Convolutional	16x(3x3x3)	16	416x416x3	416x416x16	Leaky
2	Maxpool			416x416x16	208x208x16	
3	Convolutional	32x(3x3x16)	32	208x208x16	208x208x32	Leaky
4	Maxpool			208x208x32	104x104x32	
5	Convolutional	64x(3x3x32)	64	104x104x32	104x104x64	Leaky
6	Maxpool			104x104x64	52x52x64	
7	Convolutional	128x(3x3x64)	128	52x52x64	52x52x128	Leaky
8	Maxpool			52x52x128	26x26x128	
9	Convolutional	256x(3x3x128)	256	26x26x128	26x26x256	Leaky
10	Maxpool			26x26x256	13x13x256	
11	Convolutional	512x(3x3x256)	512	13x13x256	13x13x512	Leaky
12	Maxpool			13x13x512	13x13x512	
13	Convolutional	1024x(3x3x512)	1024	13x13x512	13x13x1024	Leaky
14	Convolutional	256x(1x1x1024)	256	13x13x1024	13x13x256	Leaky
15	Convolutional	512x(3x3x256)	512	13x13x256	13x13x512	Leaky
16	Convolutional	255x(1x1x512)	255	13x13x512	13x13x255	Linear
17	Yolo			13x13x255	13x13x255	Sigmoid
18	Route			Layer 14	13x13x256	
19	Convolutional	128x(1x1x256)	128	13x13x256	13x13x128	Leaky
20	Upsample			13x13x128	26x26x128	
21	Route			Layer 20 + 9	26x26x384	
22	Convolutional	256x(3x3x384)	256	26x26x384	26x26x256	Leaky
23	Convolutional	255x(1x1x256)	255	26x26x256	26x26x255	Linear
24	Yolo			26x26x255	26x26x255	Sigmoid

Chapter 4

CGRA-based accelerator

In this chapter, the fundamentals about CGRAs and why they can be used for accelerating CNNs are briefly explained. The Deep Versat CGRA architecture, which is the inspiration behind the hardware accelerator developed in this work, is described. The chapter ends by pointing out the limitations of the Deep Versat CGRA regarding CNN acceleration.

4.1 CGRA architecture

A CGRA is a collection of programmable FUs and embedded memories interconnected by programmable switches [22]. The interconnections are reconfigurable at runtime, allowing to form different hardware datapaths to accelerate distinct computations for the same application. CGRAs are reconfigurable at the word-level and the hardware units can be programmed in any execution cycle.

Typically, CGRAs consist of a reconfigurable array, which is mainly used to accelerate program loops, and a conventional CPU, which executes the non-loop code of a given application and controls the configuration of the array. Thus, CGRAs can be used as hardware co-processors to accelerate parts of the algorithms that are slow or energy inefficient in regular CPUs [14].

The FPGA-based architecture for accelerating CNNs proposed on previous works, which was studied in Section 2.5.1, is the same as the CGRA architecture. Both consist of a spatial array of processing elements (i.e., functional units) with data flowing through an interconnection network and memory is located as close as possible to the computational units [12].

The reconfigurable array is suitable for accelerating program loops, which fits with the implementation of the convolutional layers. For all these reasons, a CGRA-based architecture can be used for accelerating CNNs. For instance, [13] implemented the AlexNet network in a CGRA by using 16 PEs and 9 parallel multipliers with fixed-point arithmetic of 8 bits for the image pixels, 16 bits for the weights, bias and output feature maps and 32 bits for intermediate results, achieving a performance of 141 GOPs, which is comparable with the ones obtained in Table 2.3. An auto-tuning compiler to map CNNs in CGRA architectures by exploring loop optimization techniques is proposed in [12] . The author claims that the developed CGRA outperforms, in terms of energy per inference, other ARM-based accelerators.

4.2 Deep Versat

Versat [22] is a 32-bit CGRA architecture developed at the INESC-ID Research Institute capable of being configured on the fly, without using pre-compiled configurations, through partial reconfiguration. The ability of generating configuration sequences from stored routines, instead of storing the configuration itself, allows to exploit the similarity between configurations as only distinct bits need to be changed, which results in a faster and less energy consuming configuration.

Versat is composed of FUs organized in a full mesh topology, which limits the number of FUs to use, due to the increase of the circuit delay caused by the selection multiplexers. To overcome this limitation, a multi-layer architecture composed of a set of Versats, called Deep Versat, was proposed in a master thesis [14]. With more layers, programs can use more FUs. Layers are stacked in a ring structure, as shown in Figure 4.1, to limit the number of connections and prevent a frequency drop.

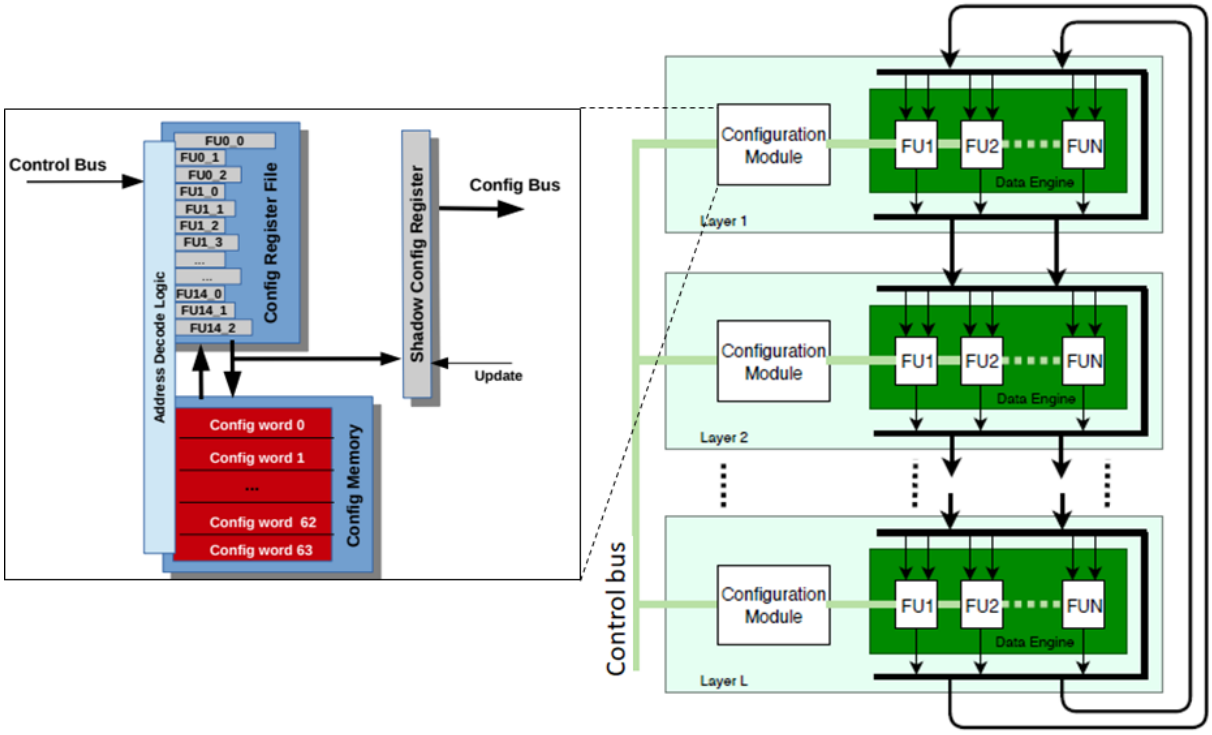

Figure 4.1: Deep Versat architecture (adapted from [14]).

Each Versat has a data engine for computation and a configuration module. In the data engine, the FUs are interconnected by a data bus which concatenates the 32-bit output of each FU of the current and of the previous layer (memories have two 32-bit outputs as they are dual-port). As the interconnection follows a full mesh topology, each FU can select any section from the data bus by means of a programmable multiplexer, depending on the configuration defined in the configuration bus. The versat layers are homogeneous as all layers have the same number and same type of FUs.

The configuration module is composed of (1) the Configuration Shadow Register, which stores the configuration currently being executed by the respective data engine, (2) the Configuration Register File, which holds the next configuration, and (3) the Configuration Memory, which saves frequently used configurations. This composition allows the configuration and execution processes to run in parallel as

the preparation of the next configuration can be done at the same time the current configuration is being executed. The configuration bus connects the data engine and the configuration module.

4.2.1 Address generator unit

The Address Generator Unit (AGU) can be seen as the core of the data engine due to being responsible for controlling the data access pattern within the FUs, besides of managing the start and the end of the execution of a given configuration. The AGU consists of two cascaded counters capable of executing two nested loops in a single configuration. For instance, the AGU can be programmed to generate the address sequence for accessing data from the memory during the execution of a program loop or to generate a counter to control individual FUs.

Figure 4.2a shows the input and output ports of the AGU, which can be divided into two interfaces, one for configuration control and another one for address control. In the configuration control interface, when the *init* input is enabled, the local counters are initialized according to the parameters of the address control interface whilst the *run* input indicates the start of the execution of a given configuration and, consequently, of the counters. When the *pause* input is enabled, the counter registers are not updated (i.e., the address generation process is halted). The *done* output indicates the completion of the address generation process and also of the execution of a given configuration.

The inputs of the address control interface are the parameters that control the nested loops responsible for generating the value of the address (*addr* output) and the value of the enable signal (*mem_en* output), which is typically used to enable memory reads or writes. As represented in Figure 4.2b, the *iterations* and *period* inputs are the number of iterations and the *shift* and *incr* inputs are the increment value of the outer and inner loops respectively. The address is initialized with the value of the *start* input before the loops. Finally, the *duty* input controls the number of iterations within the inner loop in which the address is incremented and enabled, whilst the *delay* input adds a latency in terms of clock cycles between setting the *run* and entering in the nested loops.

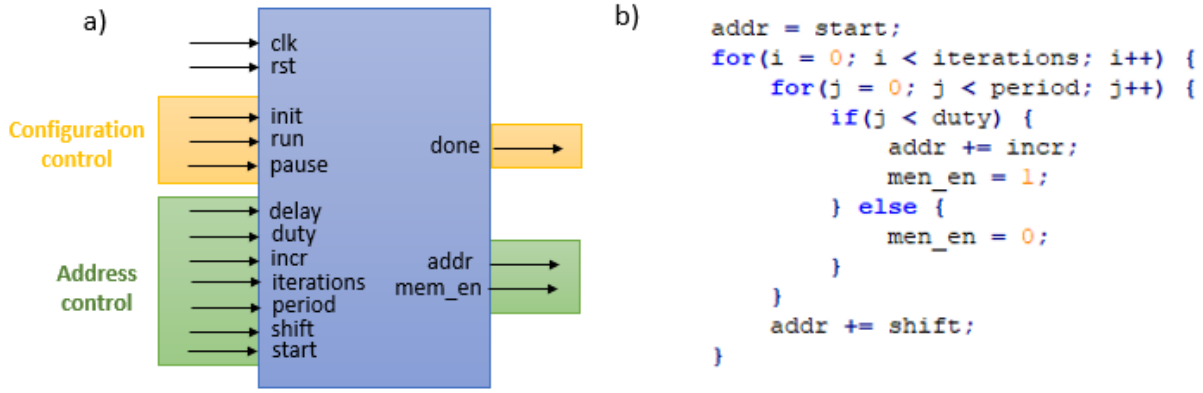

Figure 4.2: AGU a) input and output ports and b) nested loops computation.

Note that the AGU is not considered to be a FU from the Deep Versat CGRA as it is normally coupled with an actual FU such as the dual-port memories.

4.2.2 Functional units

Currently, the functional units available include dual-port memories, Arithmetic Logic Units (ALUs), multipliers, Multiplier and Accumulators (MulAdds) and barrel shifters. The number of FUs (and also the number of layers) is configured at compile time. Each FU holds different configuration parameters, settled via the configuration bus, based on their functionality:

Dual-port memory: Each Versat memory has two AGUs, one per each port. Besides the configurable parameters associated to the AGUs, the memory also has the *sel* parameter for each port to select the desired FU output from the data bus via multiplexers and the *ext* parameter to bypass the AGU. This last parameter is used when the input value selected is not a value to be written into the memory but is the address from which the memory must be read, which is useful when the data access pattern is irregular and cannot be defined by means of the AGU.

ALU / **Multiplier**: These FUs have 3 parameters: the two input operand select and the type of operation. In the case of the ALU, the most common operations include logic operations (e.g. AND, OR) and additions. The multiplier has a latency of three clock cycles and allows to choose the upper or lower 32-bit part of the 64-bit multiplication result.

Barrel shifter: The barrel shifter has a latency of 1 clock cycle and has 3 parameters: the input select of the operand to be shifted, the size of the shift and the type/direction of the shift (e.g, right/left, logic/arithmetic).

MulAdd: The MulAdd was also added by [14]. This FU uses an AGU as a counter to control the number of accumulations to perform. The accumulator resets when the value of the counter is zero. Besides the parameters from the AGU, the MulAdd also has 4 other parameters: two input operand select, the size of the shift (indicates the number of right shifts to perform after accumulation) and the type of accumulation (additive or subtractive).

4.2.3 System integration

As previously mentioned, a CGRA is typically accompanied by a CPU. As shown in Figure 4.3, Deep Versat is controlled by a RISC-V soft-processor. In this system, the processor accesses peripherals through its memory bus. Deep Versat can be seen as two peripherals, one for the control bus to start its execution, check the completion of the execution of a given configuration and manipulate the configuration registers and another one for the data bus, which is used for data transfers. The Universal Asynchronous Receiver-Transmitter (UART) module is another peripheral mainly used for debugging. Note that each peripheral can only be accessed once at a given time, which means that in this system data transfer and FU configuration cannot be done simultaneously.

The RISC-V processor is programmed with a standard GNU toolchain, which contains compilers for C and C++. Therefore, Deep Versat also includes an Application Programming Interface (API) in C++ to enhance its configuration process. Figure 4.4 illustrates the class diagram of the API, where the FUs are represented by classes and their configurations are managed by the methods of the class.

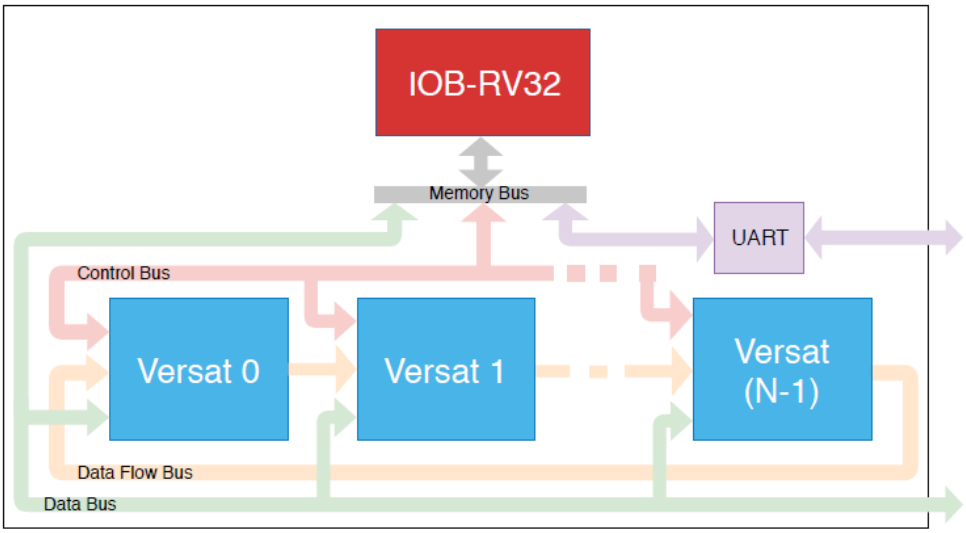

Figure 4.3: Deep Versat system ([14]).

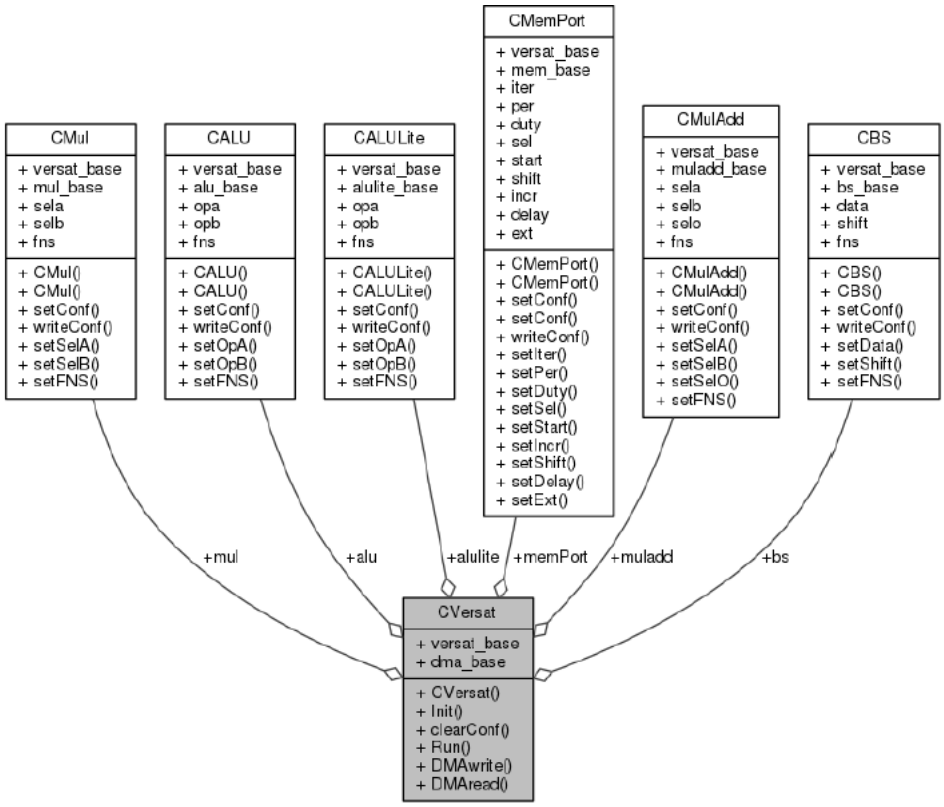

Figure 4.4: Deep Versat API class diagram ([14]).

4.2.4 Limitations for CNN acceleration

Although Deep Versat is scalable in depth (i.e. the number of FUs can be increased by adding more layers), highly configurable and costumisable (i.e., the designer can create and add existing and also its own FUs), this system still presents some limitations for accelerating convolutional layers:

Data transfer: As mentioned above, the data is driven to and from versat memories through the RISC-V processor, which is quite inefficient as the bandwith is limited and dependent on the perfor-

mance of the processor. Knowing that CNNs present high data density (the YOLOv3-Tiny network has nearly 8.8 million parameters), the acceleration would be bottlenecked by the data transfers, especially if the computation is greatly accelerated. Moreover, as both control and data buses are interfaced by the processor, data transfers and versat configurations cannot occur simultaneously. This system needs a DMA to connect directly the data bus to the external memory.

Ping-pong memories: Memories in ping-pong fashion allow to overlap data computation and communication (i.e., data transfer to/from external memory), which is typically done by splitting them in half, as shown in Figure 4.5. For instance, during a given run, data transferred from the external memory is stored in the first half of the memory whilst data stored in the second half of the memory in the previous run is read out for the FUs computation. In the next run, data is stored in the second half of the memory and read out from the first half of the memory. Although this read/write process can be currently handled by manual configuration in software, the hardware could automatically toggle the most significant bit of the read and write addresses of the dual-port memory to opposite states between runs. As a result, this process would be transparent for the user and reduce the number of configurations in the software loaded in the processor.

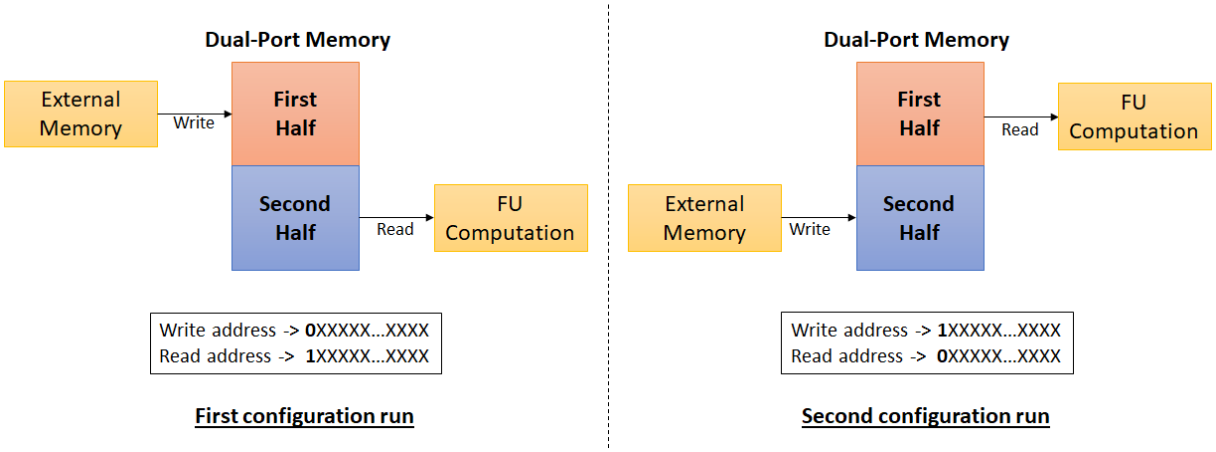

Figure 4.5: Ping-pong memory example.

Homogeneous layers: As seen in Chapter 2, convolutional layers are accelerated in reconfigurable hardware devices by deploying spatial architectures that exploit the parallelism of the MAC operations and boost data reuse. An architecture ideal for unrolling loops 4 (inter-FM parallelism) and 3 (intra-FM parallelism) is represented in Figure 4.6 and is inspired on the architecture shown in [23]. In this matrix fashion architecture, the weights are shared across the columns and the FMs are shared across the lines. The number of FUs inside each line defines the loop 4 unroll factor and the number of FUs inside each column dictates the loop 3 unroll factor (inside each MAC-based FU unit the other 2 loops might also be unrolled). This kind of composition cannot be supported by Deep Versat as two different type of layers are needed: a layer with N weight memories and a layer with one FM memory and N MAC-based FUs. Therefore, Deep Versat is limited by not supporting heterogeneous layers.

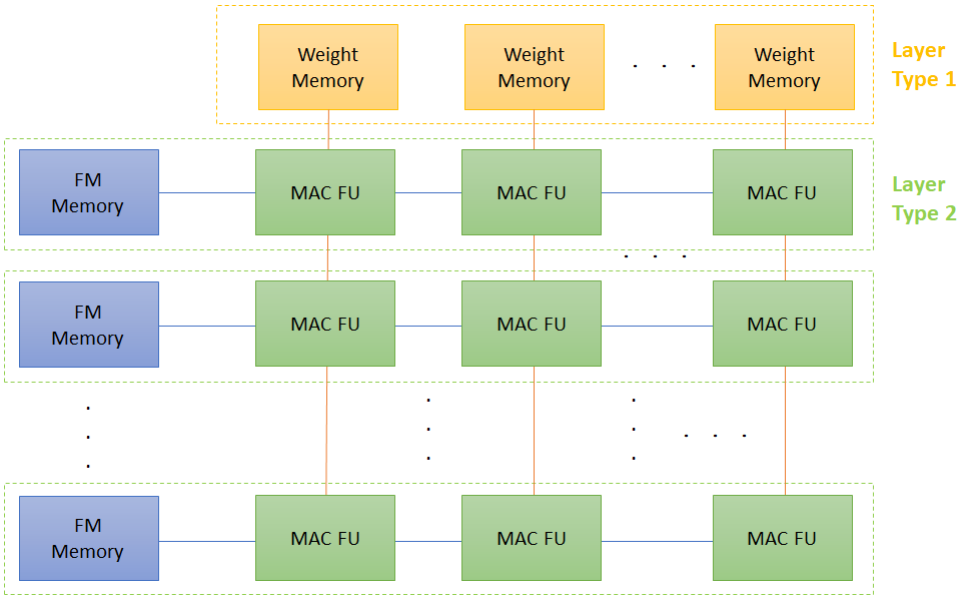

Figure 4.6: Matrix architecture for CNN acceleration.

Individual configurations: In Deep Versat, each FU is configured individually. For complex applications that use hundred of FUs, this system is not scalable neither in hardware due to requiring a high number of configuration shadow registers and configuration register files, nor in software as the growing number of configurations increases the configuration time in the RISC-V processor, besides of also rising the size of the program code. Fortunately, the parallel MACs and memories perform the same operations at the same time but with different data, which means that they could share configurations. Hence, a crucial feature for accelerating CNNs is to allow shared configurations between the same type of FUs.

Number of loops in AGUs: Currently, the AGUs support up to two nested loops. To perform a full 3D convolution in a single configuration, depending on the data format, the access pattern of the FM data requires between four and six nested loops. To achieve that, a possible solution would be to cascade several AGUs. This will be analysed later in Chapter 5.

4.3 Final remarks

CGRAs are reconfigurable hardware devices suitable for accelerating program loops such as 3D convolutions in CNNs. Deep Versat is a highly configurable and customisable CGRA but is limited in terms of the speed for accessing data from/to the external memory, homogeneous layers, individual configurations, among others. Therefore, this work focuses on implementing a hardware accelerator, improved for CNN computation and presented in the next chapter, that overcomes these constraints and inherits features from Deep Versat such as the configuration modules, AGUs and others.

Chapter 5

VersatCNN IP Core

This chapter describes the architecture of the IP core developed to accelerate CNNs based on the Deep Versat CGRA. The global architecture including the main modules and the address mapping are first introduced, highlighting the features inherited from Deep Versat. The modules and their runtime configurable parameters are then explained in a top-down manner. The chapter ends by describing the operation for configuring the AGUs in the core and the additional optimization when reading FM tiles.

5.1 High-level architecture

As explained in Section 4.2.4, the deployment of a matrix architecture, like the one shown in Figure 4.6, capable of enhancing MAC parallelism and data sharing, is not possible with the Deep Versat architecture, which prescribes a ring of fully connected stages. The said matrix architecture neither requires fully connected stages nor connections between them. Therefore, an improved hardware accelerator named VersatCNN was designed, which includes two heterogeneous stages called xWeightRead and xComp, and an AXI-based DMA, as represented in Figure 5.1.

The **xWeightRead** stage reads weights and biases from the external memory and stores them in the on-chip memory. This stage is constituted by an array of a new type of configurable FU called vRead. A vRead unit is a dual-port memory where one the ports has an AGU for writing weights and biases read from the external memory via the DMA, and the other port has an AGU for reading those values and feeding them to the xComp stage. The latter AGU has been described in Section 4.2.1 and is one of the features inherited from Deep Versat.

The **xComp** stage computes the data (3D convolution, activation functions, maxpooling, etc) and stores the results back to the on-chip memory. This stage is composed of a matrix of configurable custom computing FUs, where each row shares a vRead FU, and another new type of FU called vWrite. The vRead FU is used for reading tiles of the input FMs from the external memory. The vWrite FU is the reciprocal of the vRead FU, including a dual-port memory, an internal AGU to store the computation results and an external AGU to write the stored results to the external memory using the DMA. The various vRead and vWrite units share a *merge* block each, which is a priority encoder for allowing DMA access to only one vRead and one vWrite at a given time.

The function of the **AXI-DMA** block is to read/write data from/to the external memory. The DMA handles 256-bit wide data and allows configurable bursts for both reads and writes. It has two data native interfaces, allowing the **xComp** module to read and write from memory and an AXI4 interface to access the external memory. It also has a native configuration interface driven by the CPU. Like the FUs, the DMA can be configured while running, so that configurations, data transfers and FU computing can all happen simultaneously.

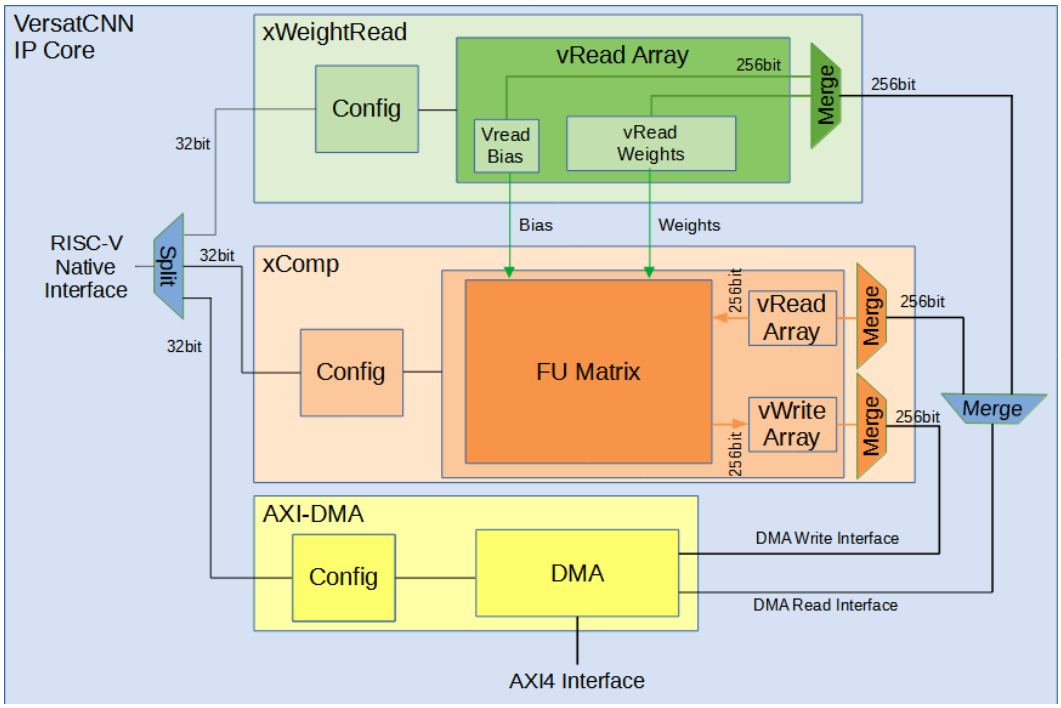

Figure 5.1: High-level architecture of the VersatCNN IP Core.

Another feature inherited from Deep Versat is the use of configuration register files with shadow registers. Each stage has a configuration module with specific configurations that are shared between the same type of FUs within the stage. These configurations are set via the RISC-V native interface taking into account the address mapping shown in Table 5.1. Apart from the internal configurations of each stage, there are global control and status registers that are common to all stages:

- **Run**: starts the execution of the configurations stored in the shadow registers of each stage.
- **Clear**: resets the configurations stored in the register files of each stage.
- **Done**: indicates the end of execution of all configurations of all the stages.

Table 5.1: VersatCNN IP core address mapping.

Configuration/Status	Base address	Operation
xWeightRead	0x7000 0000	Write
xComp	0x7100 0000	Write
AXI-DMA	0x7200 0000	Write
Run / Done	0x7800 0000	Write / Read
Clear	0x7C00 0000	Write

5.2 Detailed architecture

The detailed architecture of the VersatCNN IP core is shown in Figure 5.2. Unlike Deep Versat, the connections between the compute, vRead and vWrite FUs are fixed due to the regularity of the convolutional layers. The custom FUs are reconfigurable as in generic CGRAs, allowing them to form different hardware datapaths for different computations as later explained in Section 5.2.2.

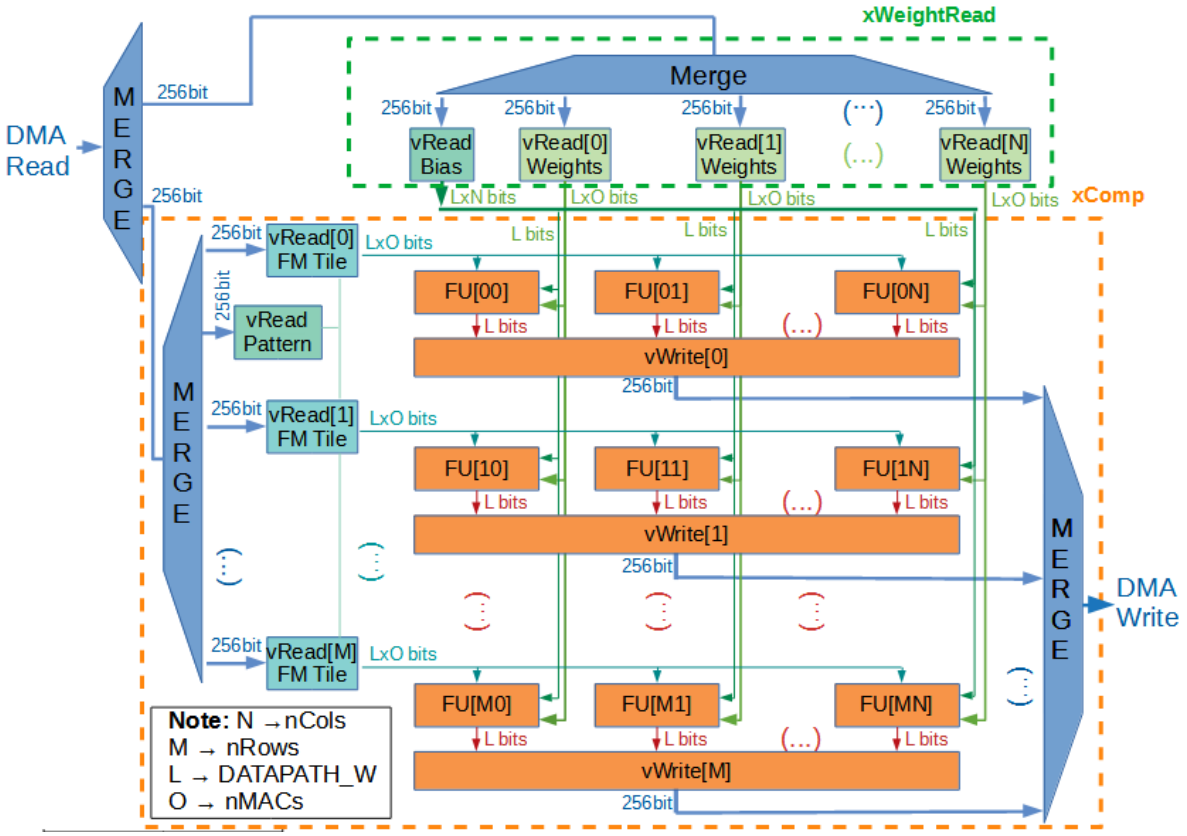

Figure 5.2: Detailed architecture of the VersatCNN IP Core.

The synthesis parameters that determine the internal architecture of the VersatCNN IP core are described in Table 5.2.

Table 5.2: Synthesis parameters of the VersatCNN IP core.

Parameter	Description
nCols	Number of custom FUs per matrix row
nRows	Number of custom FUs per matrix column
nMACs	Number of MAC units inside each custom FU
DDR_ADDR_W	Address width of the external memory
DATAPATH_W	Computation data width for weights, biases and FM tile data
VREAD_TILE_EXT_ADDR_W	External address width of the FM tile vRead memories
VREAD_BIAS_ADDR_W	Internal address width of the bias vRead memory
VREAD_WEIGHT_ADDR_W	Internal address width of the weight vRead memories
VREAD_TILE_ADDR_W	Internal address width of the FM tile vRead memories
VREAD_PATTERN_ADDR_W	Internal address width of the pattern vRead memory
VWRITE_ADDR_W	Internal address width of the vWrite memories

Each FU in the same row receives the same FM tile but a different 3D kernel, which corresponds to computing multiple output FMs in parallel (inter-FM parallelism), corresponding to the loop 4 unroll factor defined by `nCols`. In turn, each FU in the same column receives the same 3D kernel but a different FM tile, corresponding to the computation of multiple pixels of a single output FM in parallel (intra-FM parallelism), where the loop 3 unroll factor is defined by `nRows`. Therefore, the total number of FUs is `nCols x nRows`. Inside each FU, multiple 2D convolutions are computed in parallel (inter-convolution parallelism) and the loop 2 unroll factor is defined by `nMACs`. The remaining synthesis parameters give the address width of the respective module.

The dataflow is the following. The vRead FUs read data from the external memory using the DMA and stores them internally. At the same time, the data in the vRead FUs, obtained from the external memory in the previous run, is read out and broadcast to columns or rows of FUs, depending on the type of vRead FU. Each custom FU computes a different 3D convolution. The computation results of the custom FUs are concatenated and stored in the vWrite FUs, while those of the previous run are written back to the external memory via the DMA.

The following subsections present the detailed architecture of each stage.

5.2.1 xWeightRead stage

Figure 5.3 shows the detailed architecture of the xWeightRead stage, omitting the *merge* module and the dataflow. This module is composed of a Bias vRead FU and an array of Weight vRead FUs. These units read data from the external memory using their External AGUs, write these data to their internal memories (Bias or Weights memories) and, at the same time, read previous data from their internal memories to feed the compute FUs.

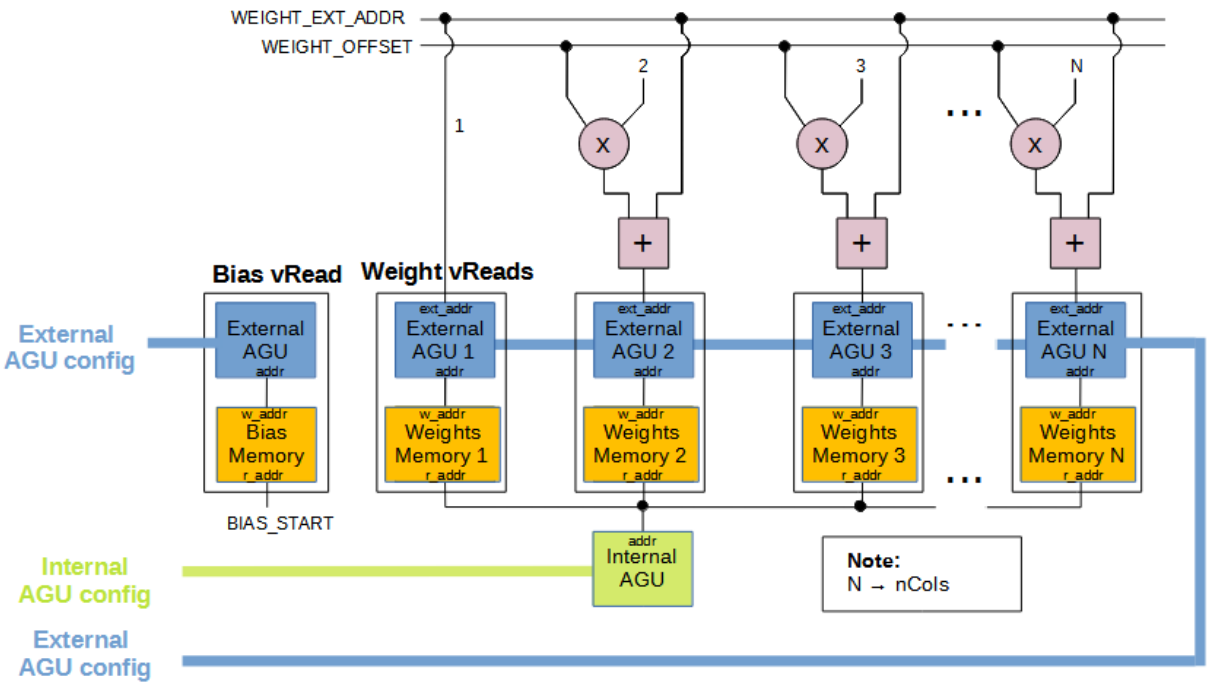

Figure 5.3: xWeightRead stage detailed architecture.

The Weight vRead FU array shares the Internal AGU to read the data from the internal memory. The external AGUs are on the other hand individual as each uses a different base address value. However, their configuration is shared because the base addresses are calculated in hardware. The base address of the first external AGU is configurable. The base address of the other external AGU is calculated by adding the base address of the first external AGU (`WEIGHT_EXT_ADDR` runtime parameter in the figure) with the product of the external AGU position and the address offset (`WEIGHT_OFFSET` runtime parameter). This results from the kernels of the same convolutional layer being typically stored sequentially in the external memory and having the same size. In spite of requiring the use of `nCols`-1 multipliers in the design, performing this calculation in hardware allows keeping the configuration size independent of the number of vReads. The weight memories are asymmetric dual-port memories, having an external bus of 256 bits and an internal bus of `nMACs x DATAPATH_W` bits as `nMACs` weights are read simultaneously from the same 3D kernel to perform inter-convolution parallelism.

Regarding the bias vRead, as the same bias is used by the custom FUs in the same matrix column, no internal AGU is needed, only a single read address defined by the `BIAS_START` runtime parameter. The bias memory is also asymmetric with an external bus of 256 bits and an internal bus of `nCols x DATAPATH_W` bits as each matrix column has a different bias.

In the vRead FUs, the external AGU controls the write address of the memories whilst the internal AGU controls their read address. The internal AGU is the same 2-loop AGU from Deep Versat (Figure 4.2a). The configurable parameters of the weights internal AGU are described in Table 5.3.

Table 5.3: Configurable parameters of the weights internal AGU.

Parameter	Description	Type
delay	Number of clock cycles to wait before performing the nested loops in the run	Configurable at runtime
duty	Number of iterations within the inner loop in which the address is incremented and enabled	Hard-wired to the period parameter
incr	Increment value of the inner loop	Configurable at runtime
iterations	Number of iterations of the outer loop	Configurable at runtime
period	Number of iterations of the inner loop	Configurable at runtime
shift	Increment value of the outer loop	Configurable at runtime
start	Initial value of the AGU	Configurable at runtime

The external AGU, represented in Figure 5.4, is a new module that handles data exchanged between the external and internal memories. The communication with the external memory is done through the native external memory interface where the address is calculated by adding a base value (`ext_addr` parameter) with an offset value. The offset is calculated by using another 2-loop AGU inside the external AGU. In turn, the communication with the internal memory is done via the native internal memory interface where the address is determined by adding a base value set by the `int_addr` parameter with an offset calculated by using a sequential counter inside the external AGU. The `direction` parameter indicates the direction of the data flow. In the case of the vReads, the direction parameter is hard-wired to zero which means that the data is read from the external memory and written into the internal memory.

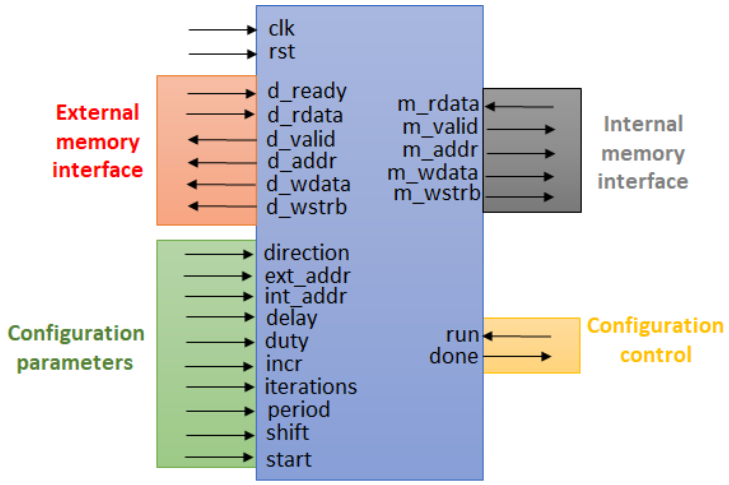

Figure 5.4: Symbol and interface signals of the external AGU.

The configurable parameters of the weights and bias external AGU are described in Table 5.4. In order to reduce the number of runtime parameters, some parameters are hard-wired as there is no need to configure their value throughout the application. For instance, the `delay` parameter is set to zero as there is no need to hold up the storage of weights and biases during a run.

Table 5.4: Configurable parameters of the weights and bias external AGU.

Parameter	Description	Weights	Bias
direction	Direction of dataflow. From external to internal memory (0) or from internal to external memory (1)	Hard-wired to 0	
ext_addr	Base address of the external memory	Computed in hardware	Configurable at runtime
int_addr	Base address of the internal memory	Configurable at runtime	
delay	Number of clock cycles to wait before performing the nested loops in the run	Hard-wired to 0	
duty	Number of iterations within the inner loop in which the address is incremented and enabled	Hard-wired to the period parameter	Hard-wired to 1
incr	Increment value of the inner loop	Configurable at runtime	Hard-wired to 0
iterations	Number of iterations of the outer loop	Configurable at runtime	
period	Number of iterations of the inner loop	Configurable at runtime	Hard-wired to 1
shift	Increment value of the outer loop	Configurable at runtime	Hard-wired to 0
start	Initial value of the AGU	Hard-wired to 0	

The remaining runtime parameters are listed in Table 5.5. Both weight and bias memories perform automatic ping-pong in hardware by toggling the most significant bit of their read and write addresses to opposite states between different runs. This is enabled by setting the `PP` runtime parameter to one.

Table 5.5: Remaining xWeightRead runtime parameters.

Runtime parameter	Description
PP	Enable ping-pong
WEIGHT_OFFSET	Address offset between external AGUs
WEIGHT_EXT_ADDR	Base address of the first external AGU
BIAS_START	Bias read address

5.2.2 xComp stage

The xComp stage is composed of an array of vRead FUs, a matrix of computing FUs and an array of vWrite FUs. The vRead units operate analogously to the vRead FUs from the xWeightRead stage. Each custom FU receives a bias, weights and pixels from the FM tile to compute mainly 3D convolutions. The vWrite units write the results to their internal memories and, simultaneously, read the previous results to be sent back to the external memory.

5.2.2.1 xComp vRead FUs

Figure 5.5 shows the detailed architecture of the xComp vReads, omitting the calculation of the base addresses of the external AGU, the *merge* module and the dataflow for simplification purposes. This module is composed of an Address Pattern vRead FU and an array of FM Tile vRead FUs.

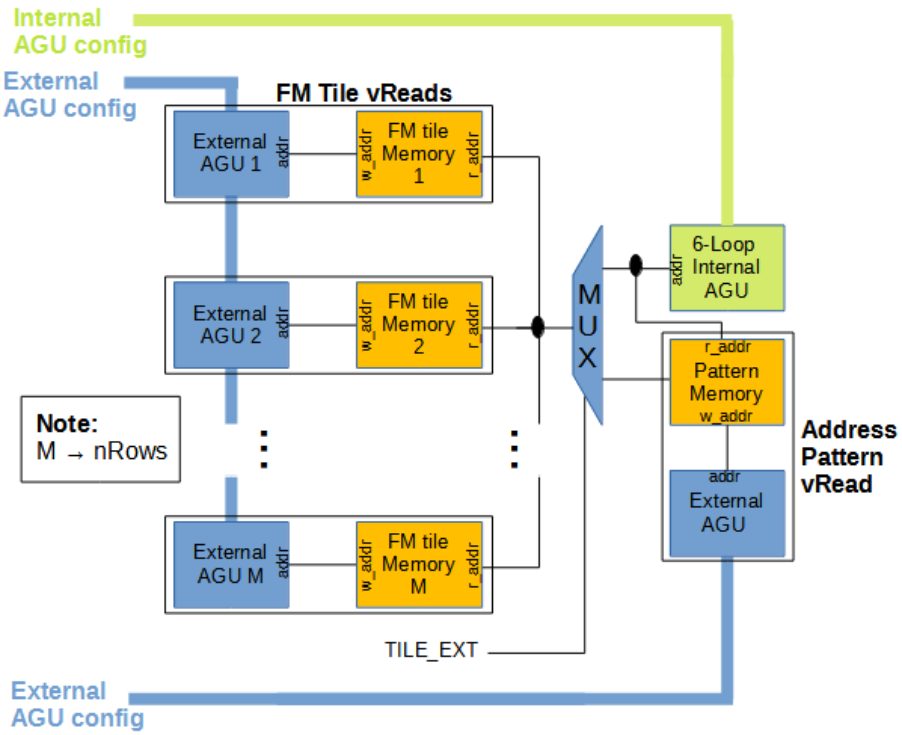

Figure 5.5: Detailed architecture of the xComp vReads.

The FM tile vReads are similar to the weight vReads, namely with: the calculation of the base address of each external AGU, which requires the use of `nRows-1` multipliers; the shared configurations between the external AGUs; the use of a single internal AGU by the asymmetric dual-port memories that store the FM tiles and the automatic ping-pong operation of those memories. The read address of the FM tile memories can be configured to come from the internal AGU or from the values stored in the address pattern memory. The selection is made by the `TILE_EXT` runtime parameter via a multiplexer. This parameter, inherited from Deep Versat, is useful when the data access pattern is not regular and cannot be determined by the internal AGU. The address pattern memory is also linked to an external AGU for pre-loading data access patterns from the external memory. The configurable parameters of the external AGUs of the FM tile and address pattern vReads are described in Table 5.6.

Table 5.6: Configurable parameters of the external AGUs of the xComp vReads.

Parameter	Description	FM Tile	Address Pattern
direction	Direction of dataflow. From external to internal memory (0) or from internal to external memory (1)	Hard-wired to 0	
ext_addr	Base address of the external memory	Computed in hardware	Configurable at runtime
int_addr	Base address of the internal memory	Configurable at runtime	Hard-wired to 0
delay	Number of clock cycles to wait before performing the nested loops in the run	Hard-wired to 0	
duty	Number of iterations within the inner loop in which the address is incremented and enabled	Hard-wired to the period parameter	
incr	Increment value of the inner loop	Configurable at runtime	
iterations	Number of iterations of the outer loop	Configurable at runtime	
period	Number of iterations of the inner loop	Configurable at runtime	
shift	Increment value of the outer loop	Configurable at runtime	Hard-wired to 0
start	Initial value of the AGU	Hard-wired to 0	

To perform a 3D convolution in a single run, the read access pattern of the FM tile memories requires more than 2 loops. Hence, the internal AGU of the xComp vReads was improved to support 6 loops by cascading 3 AGUs, which adds 2 more sets of the *incr, iterations, period* and *shift* configuration parameters, as shown in Figure 5.6a. To follow the computation shown in Figure 5.6b, the lower loop AGU is first run whilst the upper loop AGU is halted. After ending that computation, the upper loop AGU computes during only one clock cycle before halting again and controls the lower loop AGU to run again, with a different *start* value. This process carries on until the upper loop AGU ends its computation.

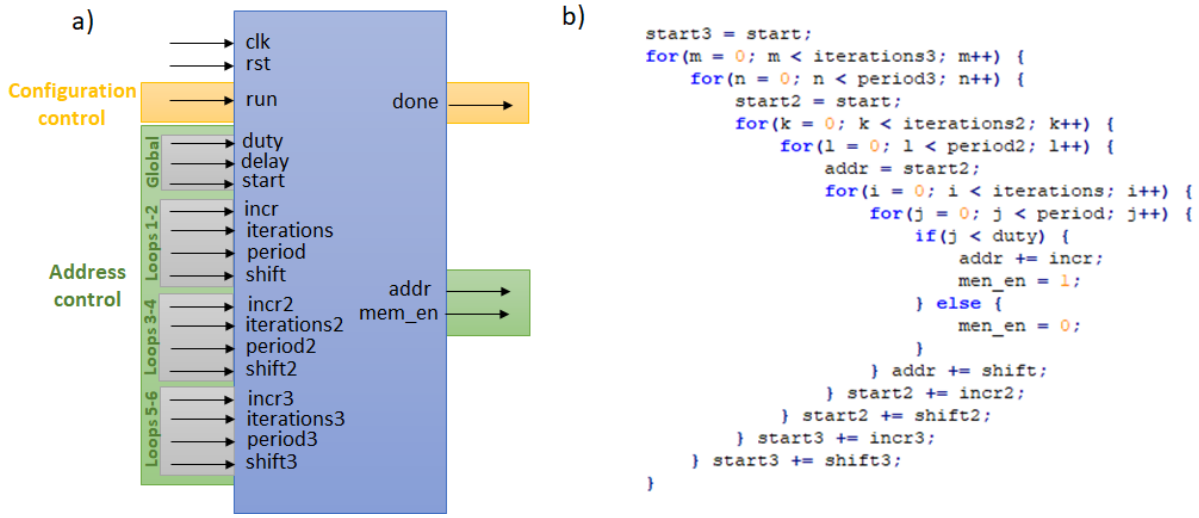

Figure 5.6: 6-loops AGU a) interface signals and b) nested loops computation.

In terms of interconnection, the cascade, which is represented in Figure 5.7, is implemented by initialising the *start* input of the lower loop AGU with the *addr* output of the upper loop AGU. The computation of the upper loop AGU is halted (using the *pause* input) while the *done* output of the lower loop AGU is zero and the *run* input of the lower loop AGU is controlled by the *mem_en* output of the upper loop AGU. The *done* output of the 6-loop AGU is one only when the *done* output of all 2-loop AGUs is also one.

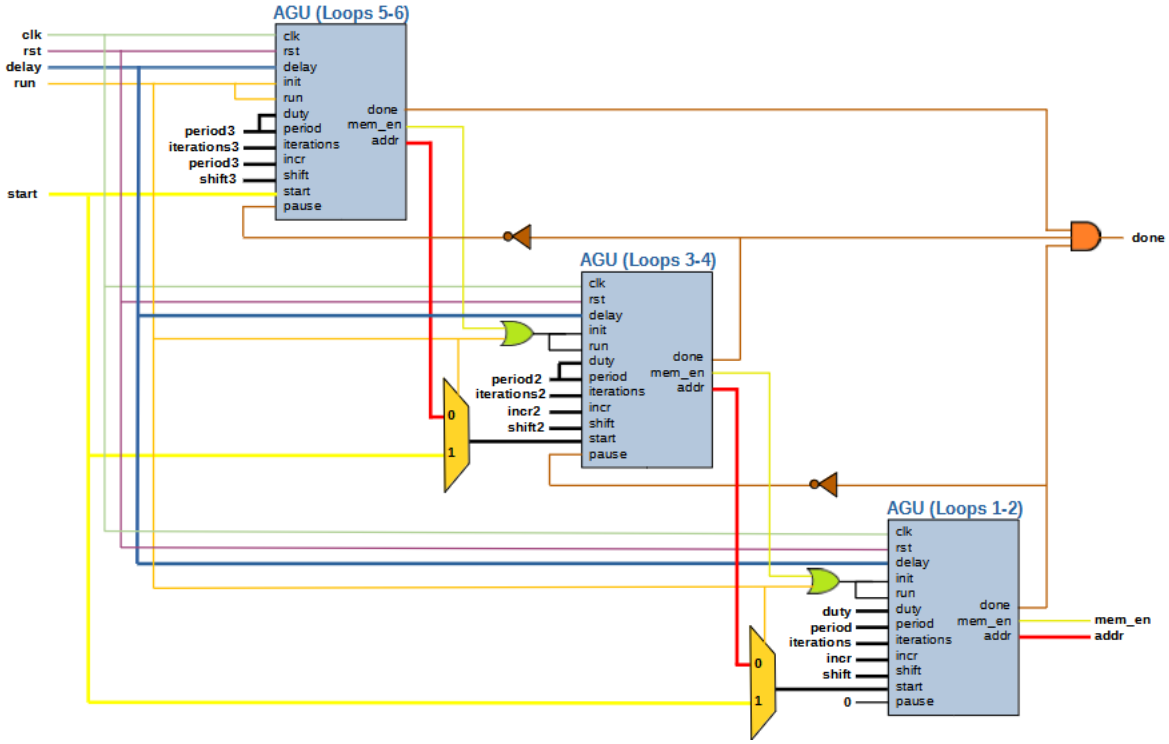

Figure 5.7: Cascade of 2-loop AGUs to form a 6-loop AGU.

The address pattern memory is linked to the same 6-loop internal AGU as the FM tile memories when being read to address them. The configurable parameters of the internal AGU are described in Table 5.7. The remaining runtime parameters are listed in Table 5.8.

Table 5.7: Configurable parameters of the 6-loop internal AGU.

Parameter	Description	Type
delay	Number of clock cycles to wait before performing the nested loops in the run	Hard-wired to 0
duty	Number of iterations within the inner loop in which the address is incremented and enabled	Hard-wired to the period parameter
start	Initial value of the AGU	Configurable at runtime
incr	Increment value of the inner loop 1-2	Configurable at runtime
iterations	Number of iterations of the outer loop 1-2	Configurable at runtime
period	Number of iterations of the inner loop 1-2	Configurable at runtime
shift	Increment value of the outer loop 1-2	Configurable at runtime
incr2	Increment value of the inner loop 3-4	Configurable at runtime
iterations2	Number of iterations of the outer loop 3-4	Configurable at runtime
period2	Number of iterations of the inner loop 3-4	Configurable at runtime
shift2	Increment value of the outer loop 3-4	Configurable at runtime
incr3	Increment value of the inner loop 5-6	Configurable at runtime
iterations3	Number of iterations of the outer loop 5-6	Configurable at runtime
period3	Number of iterations of the inner loop 5-6	Configurable at runtime
shift3	Increment value of the outer loop 5-6	Configurable at runtime

Table 5.8: Remaining xComp vRead runtime parameters.

Runtime parameter	Description
TILE_PP	Enable ping-pong
TILE_OFFSET	Address offset between external AGUs
TILE_EXT_ADDR	Base address of the first external AGU
TILE_EXT	Enables the FM Tile read address to come from the pattern memory

5.2.2.2 xComp custom FU

The detailed architecture of the custom FU is represented in Figure 5.8. The reconfigurable interconnections inside this module allow to form different datapaths to accelerate different CNN layers (e.g., convolutional with or without bias, maxpool) and activation functions (e.g., Leaky ReLU, sigmoid) individually or even in the same run. These reconfigurable interconnections are defined by the runtime parameters of the custom FU described in Table 5.9.

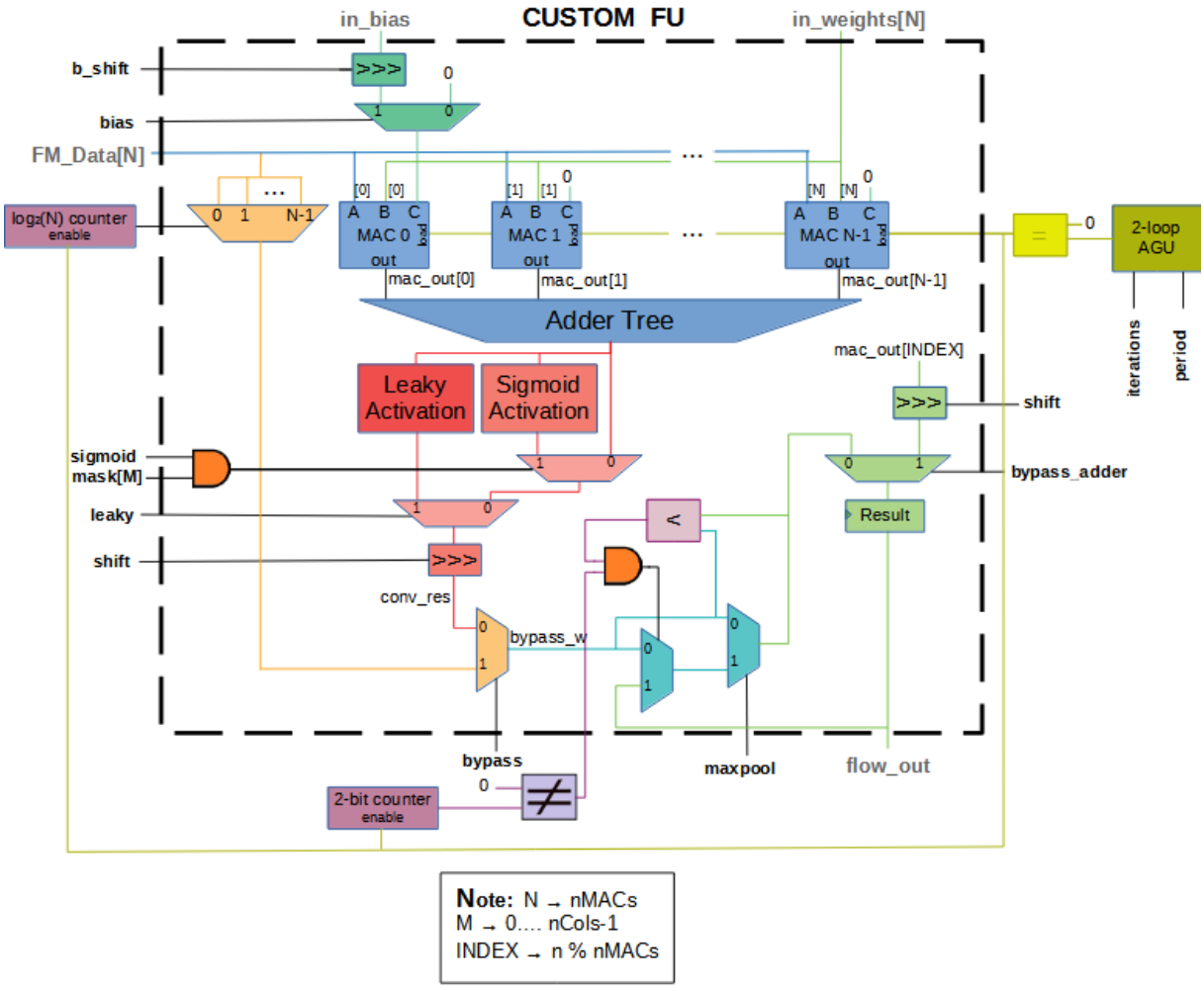

Figure 5.8: Detailed architecture of the custom FUs.

Table 5.9: Runtime parameters of the xComp custom FU.

Runtime parameter	Description
shift	Number of right shifts in convolution result
shift_b	Number of left shifts for the bias
bias	Enable bias addition
leaky	Enable leaky activation function
sigmoid	Enable sigmoid activation function
mask	Sigmoid activation mask
maxpool	Enable maxpooling
bypass	Enable convolutional bypassing
bypass_adder	Enable adder tree bypassing

The custom FU is configured at runtime to perform one of four possible operations: convolution, convolution + maxpool, maxpool and single MAC accumulation. The main operations and their respective runtime parameters are summarized in Table 5.10.

Table 5.10: Main operations performed by the custom FU.

Operation	maxpool	bypass	bypass_adder
Convolution only (default)	0	0	0
Convolution + Maxpool	1	0	0
Maxpool only	1	1	0
MAC output	X	X	1

The default operation is the 3D convolution, which is performed by MACs in parallel, where each MAC performs 2D convolutions of different input channels (inter-convolution parallelism) and by an adder tree that sums the results of the convolution across the channels. The number of MACs is defined by `nMACs`, thereby stating the loop 2 unroll factor. An internal AGU is used as a 2-loop counter to control the number of accumulations to perform by resetting the accumulator of the MACs when the output address is zero (like the MulAdd FU of Deep Versat). As shown in Table 5.11, the only runtime parameters required for the AGU are the `period` and `iterations` parameters that determine respectively the number of MAC accumulations per result and the number of results to compute in a single run.

Table 5.11: Configurable parameters of the custom FU internal AGU.

Runtime parameter	Description	Type
delay	Number of clock cycles to wait before performing the nested loops in the run	Hard-wired
duty	Number of iterations within the inner loop in which the address is incremented and enabled	Hard-wired to the period parameter
incr	Increment value of the inner loop	Hard-wired to 1
iterations	Number of iterations of the outer loop	Configurable at runtime
period	Number of iterations of the inner loop	Configurable at runtime
shift	Increment value of the outer loop	Hard-wired to the negative period parameter
start	Initial value of the AGU	Hard-wired to 0

The architecture of each MAC is represented in Figure 5.9. The accumulations are performed with the double of precision of the weights and pixels.

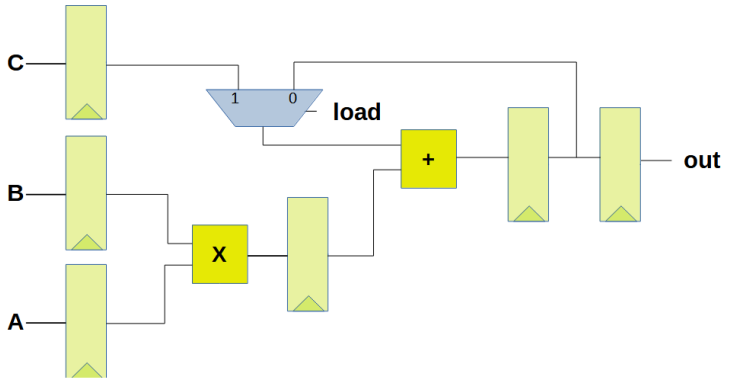

Figure 5.9: Internal architecture of a MAC.

The bias can be included in the computation of the convolutions by enabling the `bias` runtime parameter. When enabled, the accumulator of the first MAC is initialized with the value of the bias for each accumulation (for the other MACs, the accumulator is reset to zero). Thus, the computation of the bias is hidden inside the computation of the accumulations. The `b_shift` runtime parameter indicates the number of shifts to perform to the bias before the accumulation, taking into account its quantization format (different for each convolutional layer due to the dynamic quantization, as seen in Section 6.2.3).

The IP core implements the activation functions after the convolution considering the simplifications explained in Section 6.2.1. The `leaky` runtime parameter enables the leaky activation, which is implemented with 2 adders, 1 multiplexer and shifters, as represented in Figure 5.10. In turn, the `sigmoid` runtime parameter enables the sigmoid activation, which is implemented by means of simple comparators, multiplexers, adder/subtracters and a priority encoder, as shown in Figure 5.11. In case the sigmoid activation is not applied to all output channels, the `mask` runtime parameter is a second enable for the sigmoid computation but is individual for each custom FU in the matrix row. Note that the other parameters are fully shared by all custom FUs in the matrix. After the optional activation blocks, the result is shifted considering the value of the `shift` runtime parameter and the quantization format of the results.

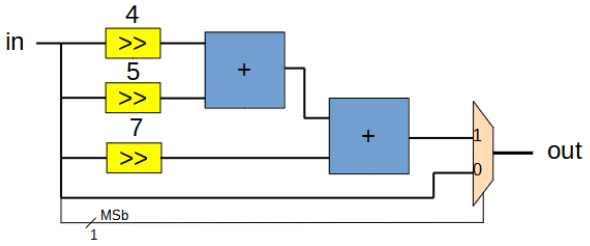

Figure 5.10: Leaky activation function in hardware.

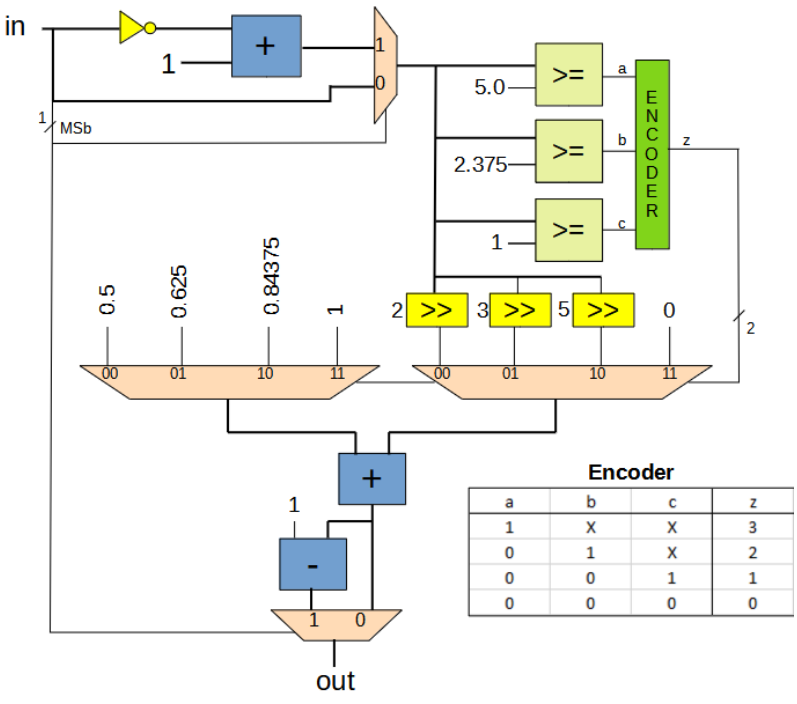

Figure 5.11: Sigmoid activation function in hardware.

The IP core was designed to allow the computation of the convolutional and maxpool layers in the same run. The computation of the maxpool, which is enabled by the `maxpool` runtime parameter, is performed with: a 2-bit counter to handle 2x2 blocks of pixels; a comparator to find the maximum value in the 2x2 block and a multiplexer to select that value. The enable of the counter is also controlled by the AGU. The maxpool can also be performed standalone (without performing convolutions in the same run) by bypassing the pixels from the FM tile to the input of the maxpool computation. This configuration is enabled by the `bypass` runtime parameter. Although the custom FU receives `nMACs` pixels simultaneously, it can only perform the maxpool of one of those pixels at a time. Hence, the design includes a multiplexer controlled by another counter (with $log_2(nMACs)$ bits) to select each pixel individually. The enable of this counter is also controlled by the AGU. Note that the internal AGU and both counters are shared by all custom FUs in the matrix.

The last operation regards to the option of bypassing the result of one of the MACs to the output of the custom FU by enabling the `bypass_adder` runtime parameter. It is used when only needing to compute individual accumulations with a single MAC. The MAC output to be bypassed is fixed and determined by $n\%nMACs$ where *n* is the column number of the custom FU in the matrix.

5.2.2.3 xComp vWrite FUs

The vWrite FUs are also composed of internal and external AGUs for addressing the internal memories storing the computing results from the custom FUs. In contrast with the vRead FUs, the internal AGU controls the write address of the memories whilst the external AGU controls their read address. Therefore, as shown in Table 5.12, the `direction` parameter of the external AGUs is hard-wired to one, which indicates that the data is read from the internal memories and written into the external memory. The runtime parameters are shared between the external AGUs.

Table 5.12: Configurable parameters of the vWrite external AGU.

Parameter	Description	Type
direction	Direction of dataflow. From external to internal memory (0) or from internal to external memory (1)	Hard-wired to 1
ext_addr	Base address of the external memory	Computed in hardware
int_addr	Base address of the internal memory	Configurable at runtime
delay	Number of clock cycles to wait before performing the nested loops in the run	Hard-wired to 0
duty	Number of iterations within the inner loop in which the address is incremented and enabled	Hard-wired to the period parameter
incr	Increment value of the inner loop	Configurable at runtime
iterations	Number of iterations of the outer loop	Configurable at runtime
period	Number of iterations of the inner loop	Configurable at runtime
shift	Increment value of the outer loop	Configurable at runtime
start	Initial value of the AGU	Hard-wired to 0

The vWrite memories are smaller than the vRead memories as the latter must store all the input channels of the FM tiles and kernels whilst the vWrite memories only store a number of output channels equal to the `nCols` parameter. The vWrite are asymmetric dual-port memories, having an external bus of 256 bits and an internal bus of `nCols x DATAPATH_W` bits. The vWrite FU array shares the Internal AGU, in which all parameters are configured in runtime, as shown in Table 5.13.

Table 5.13: Configurable parameters of the vWrite FU internal AGU.

Runtime parameter	Description	Type
delay	Number of clock cycles to wait before performing the nested loops in the run	Configurable at runtime
duty	Number of iterations within the inner loop in which the address is incremented and enabled	Configurable at runtime
incr	Increment value of the inner loop	Configurable at runtime
iterations	Number of iterations of the outer loop	Configurable at runtime
period	Number of iterations of the inner loop	Configurable at runtime
shift	Increment value of the outer loop	Configurable at runtime
start	Initial value of the AGU	Configurable at runtime

The remaining runtime parameters are listed in Table 5.14 and are related to the calculation of the base address of each external AGU, which requires `nRows-1` multipliers.

Table 5.14: Remaining xComp vWrite runtime parameters.

Runtime parameter	Description
VWRITE_OFFSET	Address offset between external AGUs
VWRITE_EXT_ADDR	Base address of the first external AGU

5.2.3 AXI-DMA

The AXI-DMA module consists of two finite state machines (one for the reads and another for the writes) that convert the requests of the vReads and vWrites (native interface) to AXI4 read and write transactions (AXI4 interface). The runtime parameters of this module are related to the number of transactions to perform as shown in Table 5.15.

Table 5.15: Runtime parameters of the AXI-DMA.

Runtime parameter	Description
WEIGHT_LEN	Number of read transfers of 256 bits for the xWeightRead vReads
TILE_LEN	Number of read transfers of 256 bits for the xComp vReads
VWRITE_NBYTESW	Number of bytes to write for the xComp vWrites

For the read transactions (vReads), the runtime parameters account for the total number of 256-bit aligned transactions in a single run. The AXI4 protocol supports a maximum of 256 transfers per burst. Therefore, the DMA contains an internal counter, initialized at the beginning of the configuration run with the total number of required transactions, that decrements each time a transaction is done in order to determine the number of transactions in each burst. For instance, if the total number of 256-bit aligned transactions is 500, the first burst will have 256 transactions whilst the second burst will have 244 transactions (`500-256`).

The runtime parameter for the write transactions (vWrites) accounts for the total number of bytes (and not 256-bit transactions) to transfer in a single run. The difference regards to the fact that the DMA write may require unaligned transactions, i.e., being able to write from any memory address any number of bytes. Hence, the DMA also includes an aligner module that manages the data bytes and the strobe to align the data with the DMA 256-bit databus.

5.3 Operation

The VersatCNN IP core is operated like Deep Versat, as explained in Section 4.2. The IP core is first configured by writing to the configurable register files and then run by writing a command to the run control register, taking into account the address mapping shown in Table 5.1. The run command executes the configurations transferred from the register files to the shadow registers. As a result, the next run can be configured during the current run without affecting its operation.

Excepting the first and last two runs, the IP core can read and write to the external and internal memories, compute and be configured for the next run, all in parallel. In VersatCNN, the AGUs are configured according to the scheme represented in Figure 5.12.

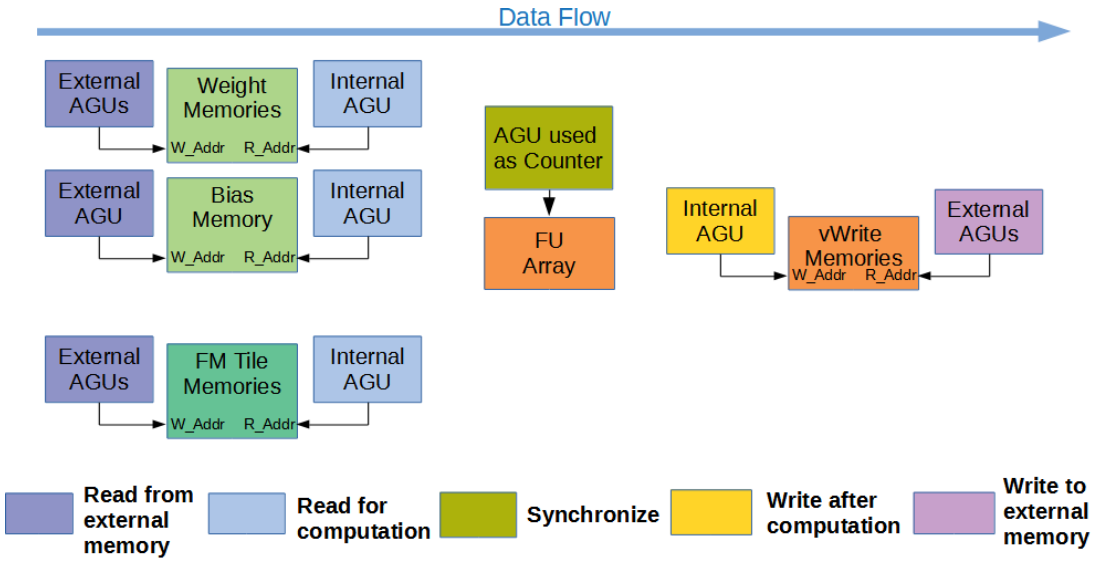

Figure 5.12: Configuration of the AGUs.

The AGUs are configured for the same run taking into account their type of operation:

- **Read**: the data (weights, biases and input FM tiles) are read from the external memory and stored in the vRead memories. This is done by configuring the vRead external AGUs.
- **Compute**: the data stored in the vRead memories is read, then computed by the custom FU matrix and finally stored in the vWrite memories by configuring all the internal AGUs in the design.
- **Write**: the computed data stored in the vWrite memories is transferred back to the external memory by configuring the vWrite external AGUs.

The operations are implemented in a pipelined fashion which means that, after the first two runs, different data is being read, computed and written in the same run. The management of the configurations in pipeline fashion is challenging in software as the programmer would need to program all the configurations at the same time taking into account that some are due in the next run and others in the next two subsequent runs. To ease the software development, the configurations of the internal AGUs (compute operation) are ”delayed” one run by adding an extra level of shadow registers, and the configurations of the vWrite external AGUs (write operation) are ”delayed” two runs by adding two extra levels of shadow

registers. As a result, the pipeline process is transparent to the programmer, who simply programs all the configurations at the same time abstracted from the fact that not all are run at the same time.

5.4 Optimization of the FM tile read process

The core was designed so that each matrix row of custom FUs computes a different line of the output FM. As a result, the tiles between consecutive vReads will share one or more lines from the input FM when performing a 3x3 convolution. For instance, Figure 5.13a shows that 2 lines are repeated between tiles of consecutive vReads when performing a 3x3 convolution over tiles of 4 lines each. Without any further optimisation, each vRead would individually read 4 lines from the external memory (one at a time, taking into account the priority defined by the merge module), resulting in several lines being repeatedly read from the external memory in the same run.

To optimize the read process, each tile vRead is coupled with a comparator that compares the address of the vRead with the address at the databus interface (which corresponds to the address of the vRead that earned the priority in the merge module) and, in case they are equal, a multiplexer chooses the data coming from the databus, as shown in Figure 5.13b. Consequently, the common lines between vReads are stored at the same time, saving communication time in the run.

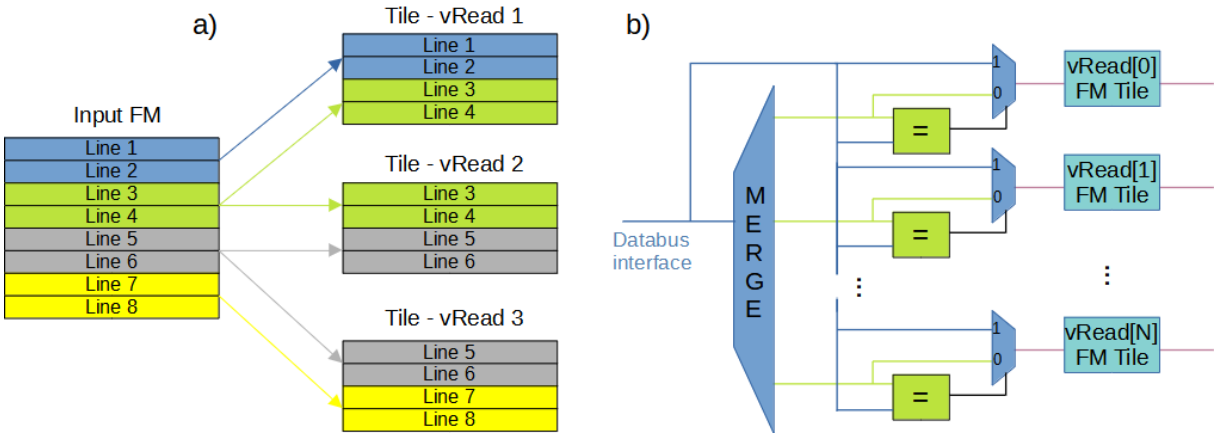

Figure 5.13: a) Example of line sharing between FM tiles and b) Address comparison for tile vReads.

5.5 Final remarks

The VersatCNN IP core is a CGRA designed to accelerate CNNs which, in the next chapter, is going to be tested to accelerate the YOLOv3-Tiny detector. The core is a substantial improvement from the Deep Versat CGRA, although inheriting some relevant features such as the configuration register files and shadow registers, the use of AGUs which kept being the core of the system, among others. The improvements include the integration of a DMA for fast data transfers between the core and the external memory, shared configurations between the same type of FUs, automatic ping-pong memories, higher loop-level AGUs and heterogeneous stages. The CNN is accelerated by a matrix of custom MAC-based FUs that exploits three types of parallelism (Inter-FM, Intra-FM and Inter-Convolution) and enhances both pixel and weight sharing. The design of the core has a total of 72 runtime configurable parameters.

Chapter 6

Implementation of YOLOv3-Tiny

This chapter describes the implementation and acceleration of the YOLOv3-Tiny detector. Firstly, the SoC platform executing the software code and the optimizations for hardware implementation are explained. The software-only version is then profiled. The chapter ends with the acceleration of the detector using the VersatCNN CGRA developed in Chapter 5 by setting the synthesis parameters of the IP core and describing the final firmware running on the SoC platform.

6.1 IOb-SoC

IOb-SoC [24] is an open-source RISC-V-based System-On-Chip platform developed by IObundle. As shown in Figure 6.1, this system is based on the PicoRV32 [25] soft-processor, which is a low-performance 32-bit CPU with an average Cycles per Instruction (CPI) of approximately 4. The CPU is programmed with the RISC-V GNU Compiler Toolchain [26], which contains compilers for C and C++.

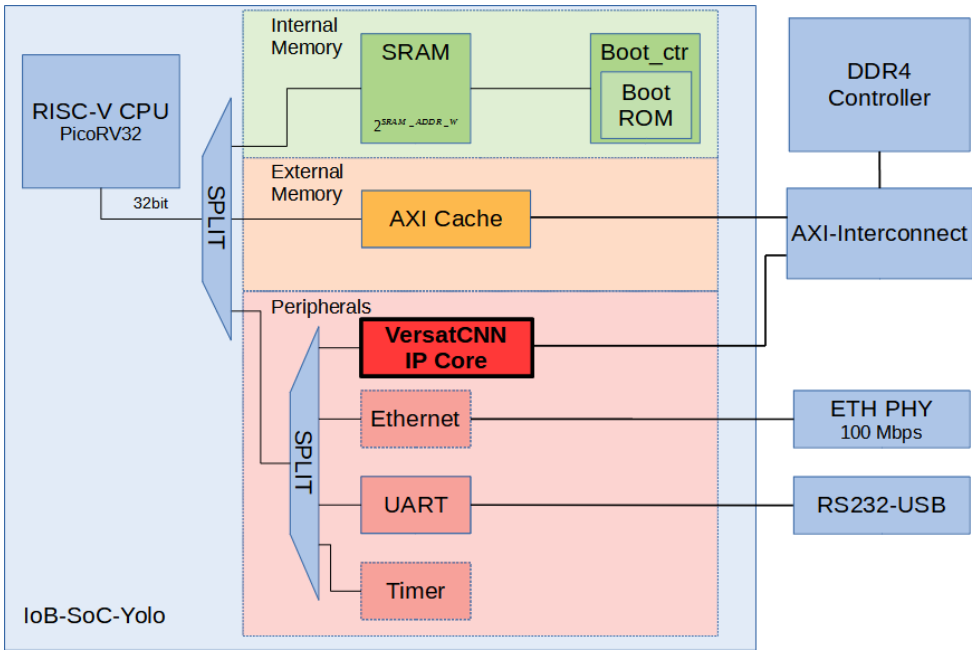

Figure 6.1: IOb-SoC-Yolo block diagram.

The RISC-V soft-processor is the master of the system and accesses to the slaves (i.e., memory sub-system and peripherals) through a native interface, which follows a simple valid-ready protocol that runs one read or write transfer at a time. Table 6.1 shows the address associated to each slave. Note that the split modules are responsible for selecting the desired slave based on the most significant bits of the address. Each slave presents a different functionality in the system:

- **Boot controller**: constituted by a Read Only Memory (ROM) that stores the bootloader, which is a program initially executed by the CPU that receives the firmware via UART and stores it in the internal memory. This controller allows to test different programs under the same hardware system without need for re-synthesis, as the firmware is transferred each time instead of being hard-loaded into the internal memory. The bootloader has 3140 bytes, fitting into a 4kB ROM.
- **Internal memory**: composed of a Static RAM (SRAM) that stores the firmware to be executed after the bootloader. The firmware program for the software-only version of the YOLOv3-Tiny detector has 29012 bytes, which fits into a 32kB SRAM.
- **External memory**: stores the weights, class labels, input image and intermediate results of the YOLOv3-Tiny detector. Its access is done through an AXI4-full-based cache developed by [27]. For testing the software-only version of the detector, the cache was configured with its default values corresponding to an associative cache with 4 ways, 16 lines, 16 words of 32 bits per line, and the Least Recently Used (LRU) replacement policy, which is a 4Kb cache.
- **Timer**: measures the performance (in terms of elapsed time) of the software application in one of three possible resolutions: **1 us**, **1 ms** or **1 s**. In the firmware, the time is calculated as with the C *time* library: the clock function is called before and after the piece of code to be measured, and the measured time corresponds to the subtraction of the start time from the end time.
- **UART**: used by the bootloader to transfer the firmware, and by the firmware for debugging purposes (e.g., print the elapsed time of each YOLOv3-Tiny layer). The IOb-SoC platform provides a C application executed on the user PC, called Console, that sends the firmware to the FPGA and keeps listening to the serial port in order to print all the debug messages sent from the FPGA.
- **Ethernet**: transfers the network weights and biases, the input and output images and the class labels. For the user PC, a Python script that uses raw sockets was developed and tested in a loop-back configuration: the PC sends a frame to the FPGA; the FPGA sends the same frame back to the PC; the PC then checks if the frame received is the same as the frame sent.

Table 6.1: IOb-SoC-Yolo address mapping.

Memory / Peripheral	Address range
Internal memory	0x0000 0000 - 0x1FFF FFFF
Boot controller	0x2000 0000 - 0x3FFF FFFF
UART	0x4000 0000 - 0x4FFF FFFF
TIMER	0x5000 0000 - 0x5FFF FFFF
Ethernet	0x6000 0000 - 0x6FFF FFFF
VersatCNN IP Core	0x7000 0000 - 0x7FFF FFFF
External memory	0x8000 0000 - 0xFFFF FFFF

6.2 Source code optimizations for hardware implementation

Darknet [17] is a full neural network framework developed in C and CUDA, supporting CPU and GPU computation. To be able to run this software application on an resource-constrained embedded system, the source code was firstly reduced to only cover the YOLOv3-Tiny detector, and then adapted to run on the RISC-V soft-processor. This included the removal of dynamic memory allocation and complex data structures. Additional optimizations, such as linear approximation of activation functions, batch-normalization folding and post-training quantization, were deployed with the purpose of simplifying the hardware computation and reducing the size of the network.

6.2.1 Linear approximation of activation functions

The activation functions used in the YOLOv3-Tiny CNN include the Leaky ReLU and the sigmoid. The Leaky ReLU simply multiplies the input, x, with the slope value, 0.1 originally, when the input is negative. The slope value can be approximated by a sum of powers of two, to replace the multiplication by a sum of multiple right shifts of the input value, as shown in Eq. 6.1. As a result, the hardware is simplified replacing a multiplier for two adders in each Leaky ReLU operation.

$$y = x \times 0.1 \approx x \times 0.1015625 = x \times \left(2^{-4} + 2^{-5} + 2^{-7}\right) = (x >> 4) + (x >> 5) + (x >> 7) \tag{6.1}$$

The sigmoid activation is a complex function to implement in hardware due to the exponential and division functions. Hence, the sigmoid is usually approximated by using LUT-based or linear transformation methods. In this work, the sigmoid was implemented by the piecewise linear approximation in [28]. This approximation is good for hardware implementation because the input x is always multiplied by a power of 2 value, which translates into a cost-free shift, as shown in Eq. 6.2.

$$f(x) = \begin{cases} 1, & \text{if } |x| \geqslant 5.0 \\ 2^{-5} \times |x| + 0.84375, & \text{if } 2.375 \leqslant |x| < 5.0 \\ 2^{-3} \times |x| + 0.625, & \text{if } 1.0 \leqslant |x| < 2.375 \\ 2^{-2} \times |x| + 0.5, & \text{if } 0 \leqslant |x| < 1.0 \end{cases} \tag{6.2}$$

Calculations are done for positive values of the input x. For negative values, Eq. 6.3 is used instead.

$$f(-x) = 1 - f(x) \tag{6.3}$$

6.2.2 Batch-normalization folding

In [29], the batch-normalization folding is proposed, which consists of a linear transformation to fold the parameters of the batch-normalization layer into the preceding convolutional layer, consequently reducing the number of parameters and operations of the network. As a result, the pre-trained floating-point weights w and biases b are updated to their new values w' and b' according to Eq. 6.4.

$$w' = \frac{\gamma \times w}{\sqrt{\sigma^2 + \epsilon}} \qquad b' = b - \frac{\mu \times \gamma}{\sqrt{\sigma^2 + \epsilon}} \tag{6.4}$$

Despite not changing the accuracy of the network for the floating-point model, this method has a negative effect on the distribution of weights and biases. As shown in Figure 6.2, the distribution of the new weights of the first convolutional layer not only has a greater value range after batch-normalization folding, as also has a greater concentration of values around zero. In addition, the second convolutional layer (layer 3), shows a distribution with a shorter range of values and also high concentration of values around zero. This range variability leads to wasted precision when performing post-training static quantization (Section 6.2.3): some layers will need more bits for the integer part (e.g., layer 1) and others more bits for the fractional part (e.g., layer 3) of the fixed-point weights and biases. That is an indication that the network would highly benefit from dynamic quantization, leading to a lower accuracy drop.

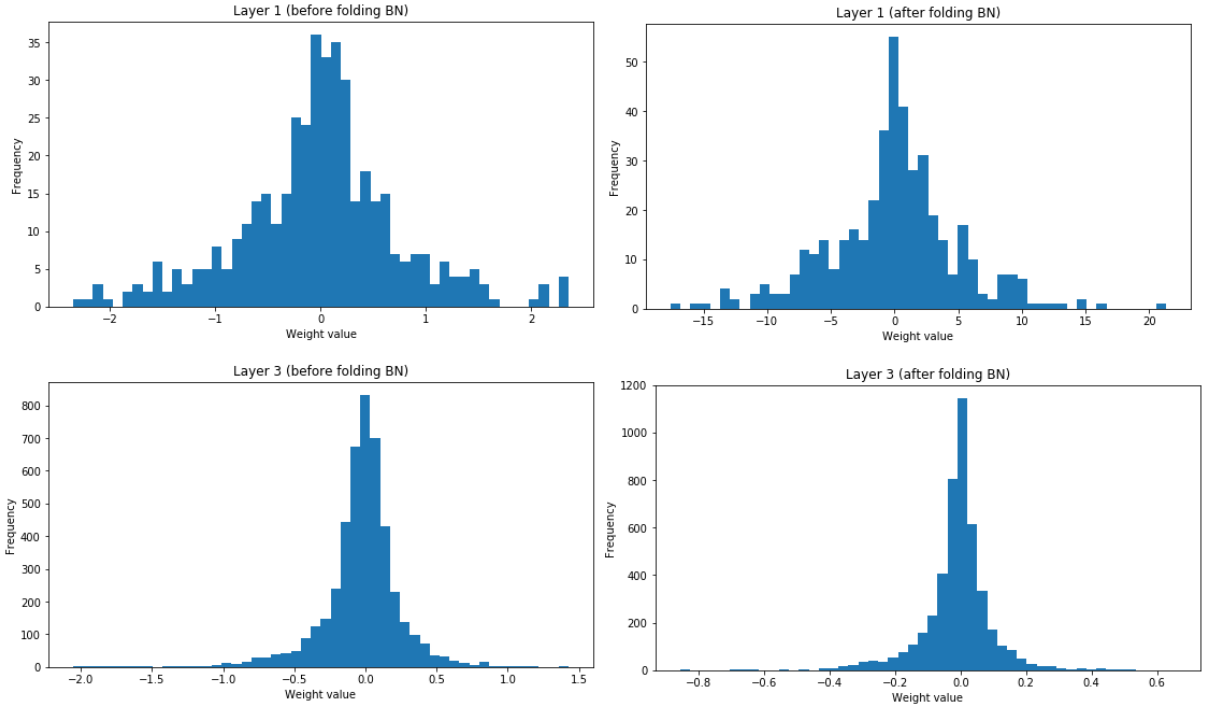

Figure 6.2: Convolutional weights of layers 1 and 3 before and after folding batch normalization.

6.2.3 Post-training quantization

As explained in Section 2.5.2, the CNN computation is typically approximated to fixed-point format for inference in FPGAs, which is also known as post-training quantization. In order to record the range of values of the FMs in different layers, the floating-point model was executed on a regular PC over the MS COCO 2017 test dataset [30], which contains a total of 20288 images. Afterwards, the fixed-point format for the weights, biases and FMs in each layer was chosen by selecting the minimum number of bits needed for the integer part to avoid overflow, leaving the remaining bits for the fractional part. The resulting ranges and respective fixed-point formats are shown in Table 6.2. Taking into account the number of bits required for the integer part, all values were quantized using 16 bits.

This analysis was performed only for the convolutional layers, as the maxpool, route and upsample layers do not process their input values, and the yolo layers preserve the resolution of their input values. Regarding the pre-CNN calculations, the input image is represented in Q8.8 whilst the intermediate and output images are represented in Q1.15. The post-CNN uses the Q3.13 format for the data. If using static instead of dynamic quantization, the weights (Q6.10), biases (Q5.11) and FMs (Q8.8) of all layers must use the highest number of bits for the integer part for preventing overflow.

Table 6.2: Ranges and fixed-point format of weights, biases and FMs per layer.

Layer	Weight			Bias			FM		
	Max	Min	Quant.	Max	Min	Quant.	Max	Min	Quant.
1	21.33	-17.70	Q6.10	3.50	-15.48	Q5.11	84.09	-8.56	Q8.8
3	0.66	-0.86	Q1.15	4.34	-1.58	Q4.12	56.68	-14.09	Q7.9
5	1.53	-1.04	Q2.14	9.40	-5.56	Q5.11	52.48	-11.31	Q7.9
7	0.82	-0.66	Q1.15	7.86	-5.84	Q4.12	45.21	-6.83	Q7.9
9	0.85	-0.52	Q1.15	2.55	-6.05	Q4.12	37.50	-4.28	Q7.9
11	0.58	-0.30	Q1.15	1.05	-4.57	Q4.12	36.97	-3.11	Q7.9
13	1.46	-1.75	Q2.14	2.29	-7.69	Q4.12	104.08	-10.55	Q8.8
14	0.31	-0.12	Q1.15	0.69	-1.09	Q2.14	18.10	-1.82	Q6.10
15	0.39	-0.65	Q1.15	3.04	-2.08	Q3.13	20.91	-3.27	Q6.10
16	0.56	-1.05	Q2.14	0.39	-3.91	Q3.13	2.17	-2.19	Q3.13
19	0.81	-1.00	Q2.14	1.55	-2.27	Q3.13	25.29	-2.59	Q6.10
22	0.22	-0.32	Q1.15	2.53	-1.46	Q3.13	25.90	-2.98	Q6.10
23	0.60	-0.76	Q1.15	0.91	-4.48	Q4.12	2.65	-2.38	Q3.13

To analyse the accuracy drop caused by the optimisations introduced in this section, several versions of the YOLOv3-Tiny detector were run on the MS COCO 2017 test dataset and the mAP_{50} metric was evaluated using the CodaLab platform [31]. The results are shown in Figure 6.3.

The approximation of the activation functions caused the mAP_{50} to drop by only 0.3. Using fixed-point static quantization causes the mAP_{50} to further drop by 2.8. However, as expected, dynamic quantization shows a higher mAP_{50} compared to static quantization, with a difference of only 1. In summary, the final fixed-point model that runs on the IOb-SoC platform presents a mAP_{50} drop of 2.1 in comparison with the original floating-point model.

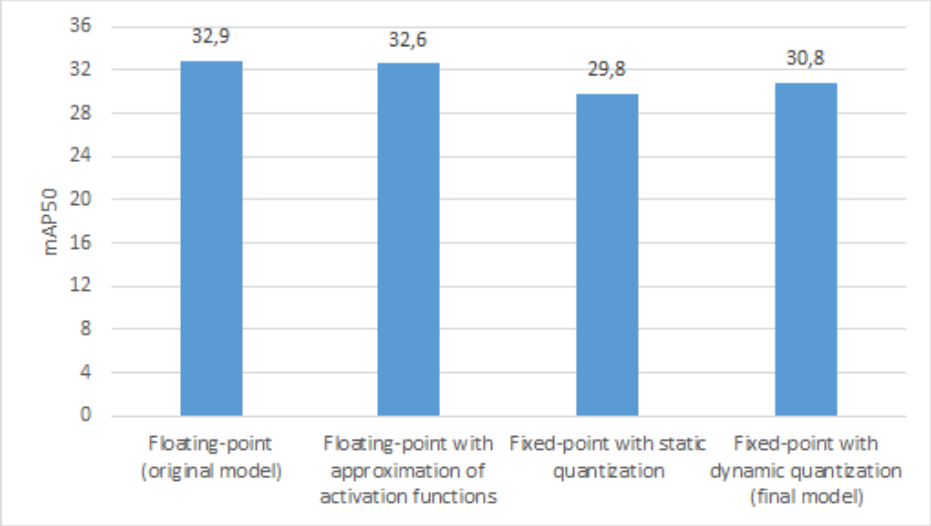

Figure 6.3: mAP_{50} of the YOLOv3-Tiny network according to the optimization applied.

6.3 Software-only version of the detector

In this section, the software-only version of the YOLOv3-Tiny detector is implemented in the RISC-V soft-processor to be used as baseline for the hardware acceleration.

6.3.1 Execution flow

The execution flow of the software-only version of the YOLOv3-Tiny detector is represented in Figure 6.4a. The firmware is divided in 4 sections: setup, pre-CNN, CNN and post-CNN.

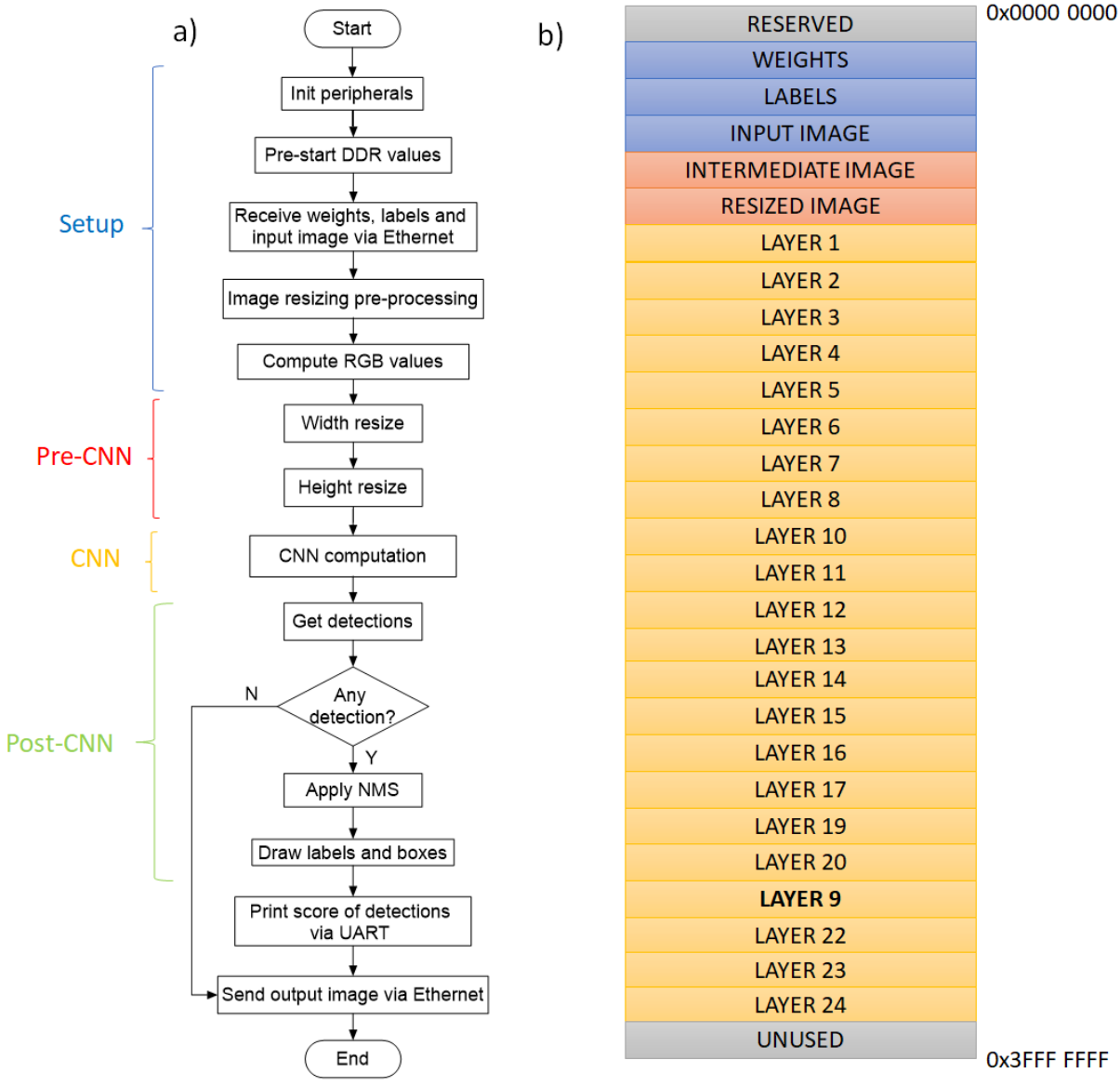

Figure 6.4: Software-only a) execution flow and b) DDR mapping of the YOLOv3-Tiny detector.

Setup: includes the initialization of the peripherals (i.e., specification of their base address) and the preparation of the data in the external memory prior to the computation of the YOLOv3-Tiny detector, thus, is not taken into account when measuring the performance of the detector in terms of elapsed time. The positions in the external memory allocated for the layers that compute convolutions with 3x3

kernels are pre-started with zero for the padding around the edges of the input FMs. The results of each layer are stored back to the external memory without overwriting the positions used for zero-padding. Analogously, the positions allocated for the resized image are pre-started with grey pixels. After receiving the weights, labels and input image via Ethernet with the message flow of Figure 6.5, the horizontal and vertical indexes and factors from the pre-processing step of the image resize algorithm (Algorithm 1) are calculated and stored in the SRAM (i.e., as program data). Finally, the Red Green Blue (RGB) values used for the boxes and labels in the post-CNN are computed and stored in the SRAM.

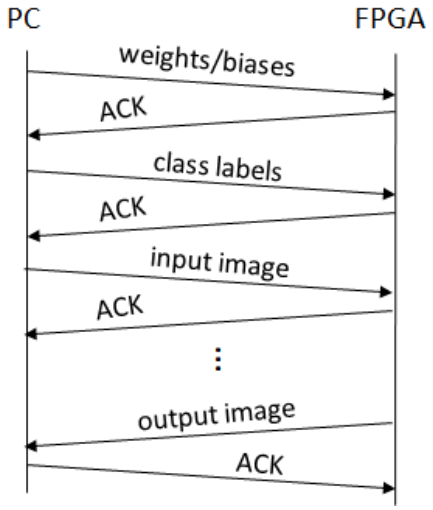

Figure 6.5: Ethernet interface message flow.

Pre-CNN: corresponds to a single function performing the width and height resize of the input image, according to Algorithm 1.

CNN computation: englobes individual functions for computing the convolutional, maxpool, yolo and upsample layers. The arguments of these functions indicate specific characteristics of the layer (e.g., the width and height of the feature map, the number of input channels, etc). The implementation of the route layers resides in basically choosing the desired base values for the output Double Data Rate (DDR) memory pointers. As shown in Figure 6.4b, the output of layer 9 is stored after the output of layer 20, as this position is more convenient to later use the output of these layers as the input of layer 22.

Post-CNN: constituted by three functions to: 1) get the detections with a score above the threshold; 2) apply non-maximum suppression (following Algorithm 2) and 3) draw the labels and bounding boxes, if any detections are found. Note that the detections are drawn over the initial input image. The execution flow of the firmware ends with the transmission of the output image (with or without detections) to the external PC via Ethernet.

6.3.2 Profiling

The execution time of the software-only version of the YOLOv3-Tiny detector running on the IOb-SoC platform (using O3 optimizations) at 143MHz is shown in Table 6.3. The total execution time is 969 seconds (above 16 minutes) which is almost completely spent for the computation of the CNN, where

the convolutional layers alone take 99.8% of the CNN execution time. The target frame rate for this work is 30 FPS, which corresponds to a total execution time of 33.3 ms. Therefore, all CNN layers need to be accelerated in hardware, using the accelerator core introduced in Chapter 5. Note that the route layers (layers 18 and 21) are not included in Table 6.3 because these layers do not consume execution time.

Table 6.3: Performance of the software baseline.

Section	Execution time (ms)	To be done in
Resize (pre-CNN)	1,040	HW
Layer 1	25,786	HW
Layer 2	748	HW
Layer 3	62,122	HW
Layer 4	375	HW
Layer 5	65,437	HW
Layer 6	188	HW
Layer 7	69,110	HW
Layer 8	94	HW
Layer 9	67,923	HW
Layer 10	48	HW
Layer 11	66,084	HW
Layer 12	93	HW
Layer 13	264,160	HW
Layer 14	29,183	HW
Layer 15	66,071	HW
Layer 16	14,332	HW
Layer 17	31	HW
Layer 19	3,607	HW
Layer 20	17	HW
Layer 22	203,425	HW
Layer 23	28,786	HW
Layer 24	126	HW
Get detections (Post-CNN)	1.94	SW
Non-maximum suppression (Post-CNN)	0.14	SW
Draw detections (Post-CNN)	11.91	HW
Total time (s)	969	

The pre-CNN process takes approximately 1 second on the CPU and must also be accelerated in the same hardware as the CNN. This is possible as the resize method (Algorithm 1) only uses multiplications and additions that can be handled by the MACs in the IP core. The post-CNN process, in spite of depending on the number of objects detected in the input image, is fast enough in software, except the draw detections method, which can be accelerated in hardware using a DMA engine.

To achieve 30 FPS, the software baseline needs to be accelerated nearly 29000 times. The software baseline is slow, not only due to the low performance of the soft-processor and the high computation requirements of the detector, but also because of the low bandwidth of the Advanced eXtensible Interface (AXI) cache available at the time, which interfaces the processor and the DDR memory. The cache read burst length was configurable but not pipelined, requiring 2 cycles per word. For cache writes, the scenario is even worst: words are sent to the external memory 1 by 1 because the cache uses the write-through policy. Hence, the IP core developed in Chapter 5 not only must accelerate the computation using parallel MACs but also provide adequate memory bandwidth by using a DMA core.

6.4 Setting the synthesis parameters of the IP core

Most of the synthesis parameters that determine the internal architecture of the VersatCNN IP core (Table 6.4) are defined by the loop unroll and loop tiling factors chosen to accelerate the YOLOv3-Tiny network. As studied in Section 2.5.3, previous works performed design space exploration in order to choose the factors that achieve the maximum computational throughput. This work follows a slightly different approach by theoretically choosing the factors that allow to achieve a target frame rate of 30 FPS (i.e., 33.3 ms) taking into account the characteristics of the YOLOv3-Tiny CNN.

Table 6.4: Synthesis parameters chosen for the IP core.

Parameter	Value	Parameter	Value
nCols	16	VREAD_BIAS_ADDR_W	3
nRows	13	VREAD_WEIGHT_ADDR_W	14
nMACs	4	VREAD_TILE_ADDR_W	15
DDR_ADDR_W	32	VREAD_PATTERN_ADDR_W	10
DATAPATH_W	16	VWRITE_ADDR_W	8
VREAD_TILE_EXT_ADDR_W	15		

6.4.1 Loop unrolling

The execution time for the computation of the convolutional layers in the VersatCNN IP core can be estimated according to Eq. 6.5. As expected, the execution time decreases when increasing the parallelism factor (i.e., the number of MAC units performing computations in parallel in the design) and the clock frequency. For achieving an execution time of 33.3 ms, the parallelism factor must be between nearly 835 and 584 for clock frequencies between 100 and 147 MHz, as represented in Figure 6.6.

$$\texttt{Execution time} \approx \frac{\texttt{\# MAC operations}}{\texttt{Total parallelism factor} \times \texttt{Frequency}} \times \texttt{Efficiency} \tag{6.5}$$

The VersatCNN IP core exploits 3 different sources of parallelism (inter-FM, intra-FM and inter-convolution), hence, the total parallelism factor corresponds to the product of the 3 loop unroll factors. These factors must be carefully chosen taking into account the network characteristics for not leading to the underutilization of the MAC resources:

- **Inter-FM parallelism factor**: the loop 4 unroll factor depends on the number of kernels of the convolutional layers. In the YOLOv3-Tiny network, the layer 1 presents the lowest number of kernels (16) whilst all the other layers have a number of kernels also multiple of 16, except only from layers 16 and 23. Therefore, the loop 4 unroll factor chosen was 16, which is defined by the `nCols` parameter. A kernel of padding is added to layers 16 and 23 to become multiple of 16.
- **Intra-FM parallelism factor**: as each matrix row of custom FUs in the IP core computes a different line of the output FM, the loop 3 unroll factor depends on the height of the input FMs. In the YOLOv3-Tiny network, all layers present an input FM with a height multiple of 13, hence, to ensure that all matrix rows of custom FUs are used, the loop 3 unroll factor selected was 13 (`nRows` synthesis parameter).

- **Inter-convolution parallelism factor**: with the unroll factors chosen for loops 4 and 3, the total parallelism factor is 208 (16x13), which is still insufficient by 3 or 4 times to achieve the target frame rate. The loop 2 unroll factor depends on the number of input channels of each layer. In the YOLOv3-Tiny network, all layers present a number of input channels multiple of 4, except from the first layer. As a result, the loop 2 unroll factor chosen was 4 (`nMACs` synthesis parameter) and an extra input channel of padding is added to layer 1 to also become multiple of 4.

The total parallelism factor chosen for the IP core is then 832 (16x13x4), which leads to estimated execution times between 33.4 ms and 23.4 ms for clock frequencies between 100 and 147 MHz, as represented in Figure 6.6.

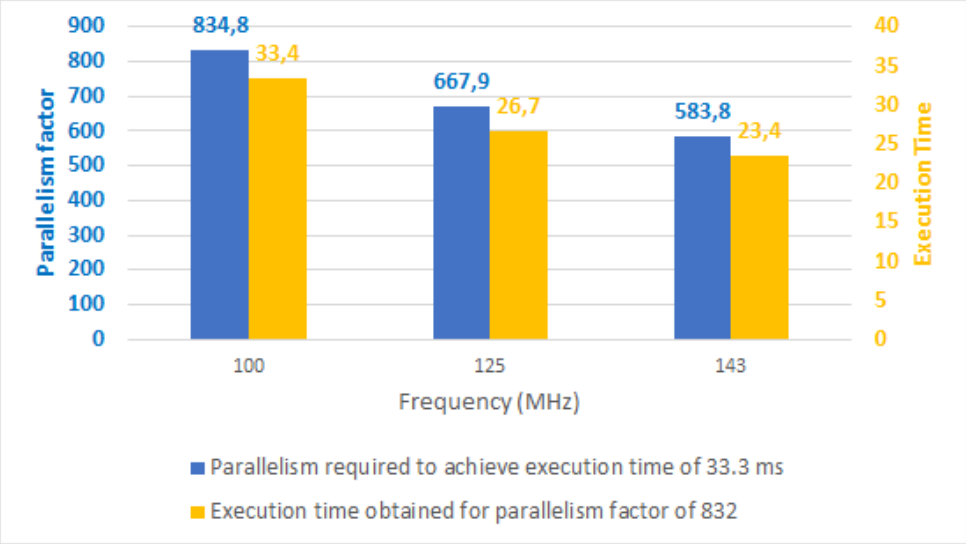

Figure 6.6: Estimated parallelism factor and execution time per frequency.

6.4.2 Loop tiling

The IP core was designed to perform 3D convolutions of input FM tiles in a single run, which requires all input channels to be stored in the on-chip memory for both weight and FM tile vReads. As each matrix row of custom FUs computes a different line of the output FM, each tile shall only have the number of lines (i.e., height) required to compute one line of results from the output FM. For instance, for layer 1, as the kernel is 3x3 and the maxpool is performed in the same run (as further explained in Section 6.5.3), the tile needs 4 lines from the input FM to output one line of results.

The width tiling factor must be a divisor of the width of the input FM to ensure that it is equally splitted, consequently avoiding an utilization ratio of the MACs lower than one. At the same time, the tiles should be as wide as possible in order to reduce the number of pixels repeated in the borders between tiles of the same vRead and to read data from the DMA with large bursts, further reducing the communication time. Figure 6.7 shows how the average time of each read burst of 256 bits decreases when increasing the tile width for the convolutional layer 3, as the number of bursts per AXI transaction is also increased.

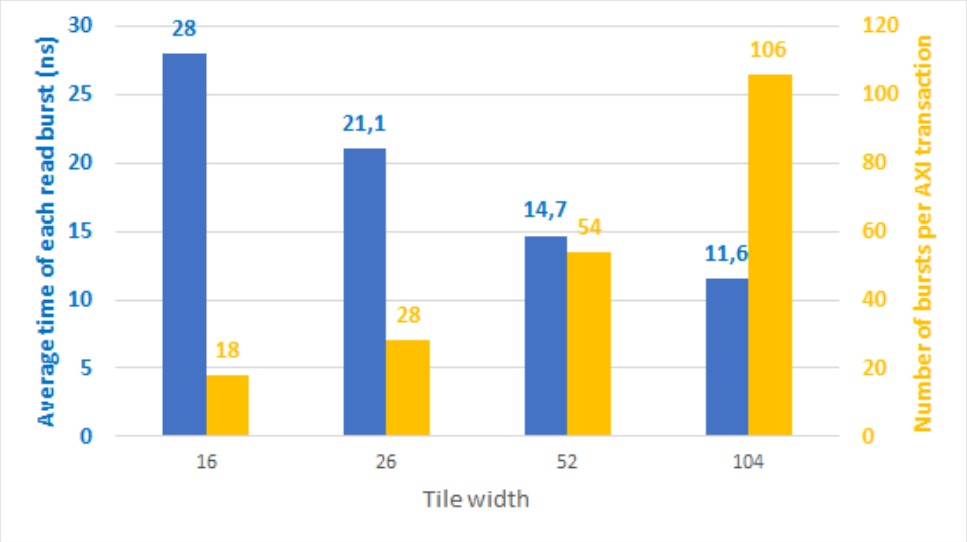

Figure 6.7: Average time of each burst read per tile width for layer 3 .

In this work, the width of the tiles for each layer is chosen so that the communication time is below the computation time in each run. The computation time of the tile convolution is expressed by:

$$\texttt{Computation time} = \begin{cases} (\texttt{ker\_w} \times \texttt{ker\_h} \times \texttt{c} \times \texttt{tile\_w} \times \texttt{clk\_per}) \,/\texttt{nMACs} & \text{, without maxpool} \\ (\texttt{ker\_w} \times \texttt{ker\_h} \times \texttt{c} \times \texttt{tile\_w} \times \texttt{clk\_per} \times 2) \,/\texttt{nMACs} & \text{, with maxpool} \end{cases} \tag{6.6}$$

where `tile_w` is the tile width and `clk_per` is the clock period. In turn, the communication time is the sum of the times to read the kernels and FM tiles, which are estimated by:

$$\texttt{Kernel comm. time} = (\texttt{ker\_w} \times \texttt{ker\_h} \times \texttt{c} \times \texttt{nCols} \times \texttt{burst\_avg}) \,/16 \tag{6.7}$$

$$\texttt{Tile comm. time} = \begin{cases} (\texttt{tile\_w} \times \texttt{nRows} \times \texttt{c} \times \texttt{burst\_avg}) \,/16 & \text{, for 1x1 conv} \\ ((\texttt{tile\_w} + 2) \times (\texttt{nRows} + 2) \times \texttt{c} \times \texttt{burst\_avg}) \,/16 & \text{, for 3x3 conv} \\ ((\texttt{tile\_w} + 2) \times (\texttt{nRows} \times 2 + 2) \times \texttt{c} \times \texttt{burst\_avg}) \,/16 & \text{, with maxpool} \end{cases} \tag{6.8}$$

where `burst_avg` is the average time of each burst. The communication times are divided by 16 as each burst of 256 bits is composed of 16 words of 16 bits each. The DDR4 controller executes at a higher frequency (200MHz), hence, the DMA writes are approximately performed in parallel with the DMA reads. As the data to be written during a run is lower than the data to be read, the communication time estimated only considers the DMA reads. Note that there are runs where only one between the kernel and FM tile is transferred.

The chosen dimensions of the input FM tiles per layer are represented in Table 6.5. The width of the tile must be added by two when the kernel is 3x3. For example, for layer 1, the width tiling factor is 208 (divisor of 416) but the width of the tile is 210. The biggest FM tile belongs to the layer 22 and has 32256 pixels (28x3x384), therefore, `VREAD_TILE_ADDR_W` is 15 (2^{15} = 32768). As each pixel is 16 bits, each FM tile memory requires 64kB. For the layers where the tiles cover the entire width of the input FM (i.e., layers 9 to 23), the FM tile memories are not used in ping-pong fashion, as they are transferred only

once or twice depending on the scale (13x13 or 26x26). As a result, the size of the FM tile memories does not double due to the ping-pong operation as in the other memories in the design.

Table 6.5: Memory requirements for vRead and vWrite FUs.

Layer	3D kernel size	Input FM Tile	Results per tile
1	(3x3)x3	(210x4)x3	104
3	(3x3)x16	(106x4)x16	52
5	(3x3)x32	(54x4)x32	26
7	(3x3)x64	(28x4)x64	13
9	(3x3)x128	(28x3)x128	26
11	(3x3)x256	(15x3)x256	13
13	(3x3)x512	(15x3)x512	13
14	(1x1)x1024	(13x1)x1024	13
15	(3x3)x256	(15x3)x256	13
16	(1x1)x512	(13x1)x512	13
19	(1x1)x256	(13x1)x256	13
22	(3x3)x384	(28x3)x384	26
23	(1x1)x256	(26x1)x256	26
Max	(3x3)x512	(28x3)x384	104

The address width of the vWrite memories depends on the number of results computed per tile in each run, which in turn depends on the tiling factor chosen for the width of the tiles. For instance, layer 1 has a width tiling factor of 208, then, as maxpooling is done in the same run, this layer computes 104 results per tile, which actually corresponds to the maximum number of results per tile, as shown in Table 6.5. As the vwrite memories always work in ping-pong fashion, the double of the results must be considered, hence, `VWRITE_ADDR_W` is 8 (2^8 = 256).

Table 6.5 also shows that layer 13 presents the biggest 3D kernel with a total of 4608 (3x3x512) weights. The double of weights must be taken into account as the weight memories also work in ping-pong fashion, therefore, `VREAD_WEIGHT_ADDR_W` is 14 (2^{14} = 16 384). The weights are also represented in 16 bits, thus, each weight memory requires 32Kb.

6.5 Final firmware using VersatCNN

The execution flow of the final firmware using the VersatCNN IP core to accelerate the YOLOv3-Tiny detector is also divided in 4 parts: setup, pre-CNN, CNN and post-CNN. This section shows how the acceleration of the YOLOv3-Tiny detector is handled by the VersatCNN core, without details about the software code, as that is subject of other dissertation work [15]. Note that the final firmware program occupies over 34kB, hence, the SRAM of the IOb-SoC-Yolo platform is increased from 32kB to 64kB.

6.5.1 Setup

The setup is similar to the software baseline, with some modifications in the data storage. The values calculated in the pre-processing step of the image resize algorithm and the RGB values are stored in the external memory, instead of the SRAM, to later be accessed by the IP core via the DMA. Also, the horizontal indexes are loaded into the pattern memory to be used during the width resize in the pre-CNN.

In the software baseline, the weights and FMs are stored in XYZ format, which means that each channel is first stored by column and row before the next channel. For the final firmware, the data is converted to ZXY format, which corresponds to storing first by channel and only then by column and row. Figure 6.8 illustrates an example of a 3x3x3 FM in both XYZ and ZXY formats. This example also shows that, when reading a 2x2 tile from the FM in XYZ format, 3 loops are required: one for iterating through the columns, one for iterating through the rows and another one for iterating through the channels. However, when using the ZXY format, as the channels and columns are adjacent, only 2 loops are required. As a result, the external AGUs from the vReads and vWrites of the IP core only require 2 loops for addressing the external memory, thereby saving hardware resources and simplifying the software configurations. The usage of the ZXY format also eases the reading from the vRead memories as each MAC inside the custom FU computes different channels in parallel.

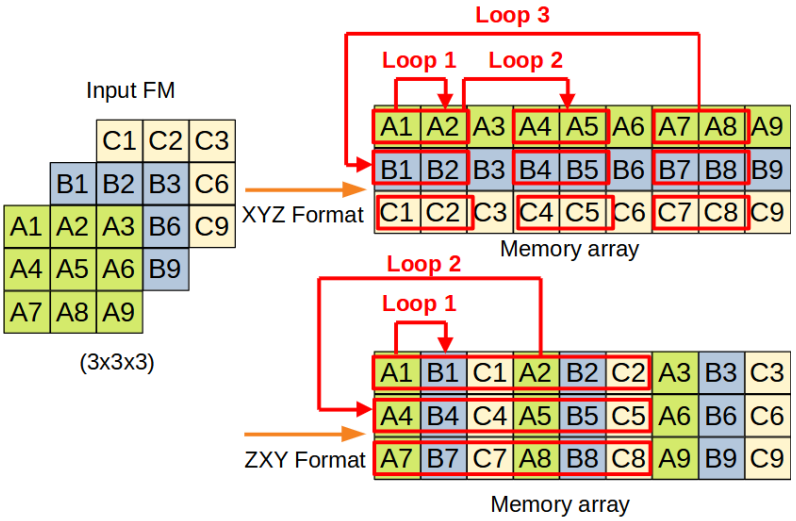

Figure 6.8: Example of 3x3x3 FM in XYZ and ZXY format.

As explained in Section 5.2.3, the DMA reads must consist of 256-bit (i.e., 32-byte) aligned transactions, thus, the base address of each kernel and of each line of the input FM must be multiple of 32. All the layers in the network, except from the first layer, have a number of input channels multiple of 16, therefore, as the data is represented in 2 bytes, the base addresses are always aligned. For the first layer, both kernel and FM lines must be padded until being a multiple of 32.

6.5.2 Pre-CNN acceleration

As stated in Algorithm 1, the pre-CNN image resize method is divided in width and height resize, which are accelerated in the IP core by performing two accumulations per MAC:

- **Width resize**: each line of the input image is stored in a different FM tile vRead, hence, if the height of the input image is not a multiple of 13, padding lines must be added. The pixels in the vRead memories are addressed by the pattern memory that stores the irregular patterns of the horizontal indexes, instead of using directly the internal AGU. Thus, the `TILE_EXT` parameter is set to one. The horizontal factors are stored in the weight vReads. For each matrix row, only one custom FU

is used per input channel instead of using all the custom FUs in the row, as unlike the kernels in the convolutional layers, there is no parallelism between horizontal factors. Also, only one MAC is used per custom FU by setting the `bypass_adder` parameter to one as the channels are not summed as in the convolutions. The input image is padded to include 4 channels, therefore, the total parallelism factor for the computation of the width resize is 52 (13x4).

- **Height resize**: the lines from the intermediate image (i.e., the image resulting from the width resize) to be stored in the FM tile vReads depend on the values of the vertical indexes. As these values do not follow any regular pattern, the `ext_addr` parameter of each external AGU cannot simply be calculated by adding the base address with an offset. Consequently, only the first matrix row of custom FUs is used and the `ext_addr` parameter of its external AGU is determined by the base address configured in software according to the lines selected by the vertical indexes. Analogously to the width resize, the vertical factors are stored in the weight vReads and only 4 MACs are used per matrix row. Given that only one matrix row is used, the total parallelism factor for the computation of the height resize is only 4, which will not result in a bottleneck as the pre-CNN is bounded in communication and not in computation.

6.5.3 CNN acceleration

The configurations of the custom FUs for each layer (or pair of layers) are represented in Table 6.6.

Table 6.6: Configurations of custom FUs per layer.

Layers	Bias	Leaky	Sigmoid	Maxpool	Bypass
(1-2), (3-4), (5-6), (7-8)	1	1	0	1	0
10, 12	X	X	X	1	1
9, 11, 13, 14, 15, (19-20), 22	1	1	0	0	0
(16-17), (23-24)	1	0	1	0	0

As explained in Section 5.2.2.2, the VersatCNN IP core can compute the convolutional and maxpool layers in the same run. Therefore, the first 4 maxpool layers are computed jointly with their previous convolutional layer and, as a result, the computation of these maxpool layers is hidden in the computation of the convolutional layers. Moreover, the communication time in these layers is further reduced as less input and output FMs are transferred between the IP core and the external memory.

Figure 6.9 shows that, when performing both convolution and maxpool, the access pattern when reading pixels from the tiles requires 5 loops: 2 for iterating according to the kernel size, other 2 for iterating according to the 2x2 block and the last one for iterating through the tile width. This access pattern is supported by the internal AGU shared among all tile vReads that performs 6 loops.

The maxpool in layer 10 cannot be computed alongside the convolution in layer 9 as the output of layer 9 is required for the second route layer in order to be part of the input of layer 22. Therefore, the maxpool in layer 10 is computed standalone. The same applies for layer 12 as the maxpool is performed with a stride of 1, thus adding an extra column and line to the input FM.

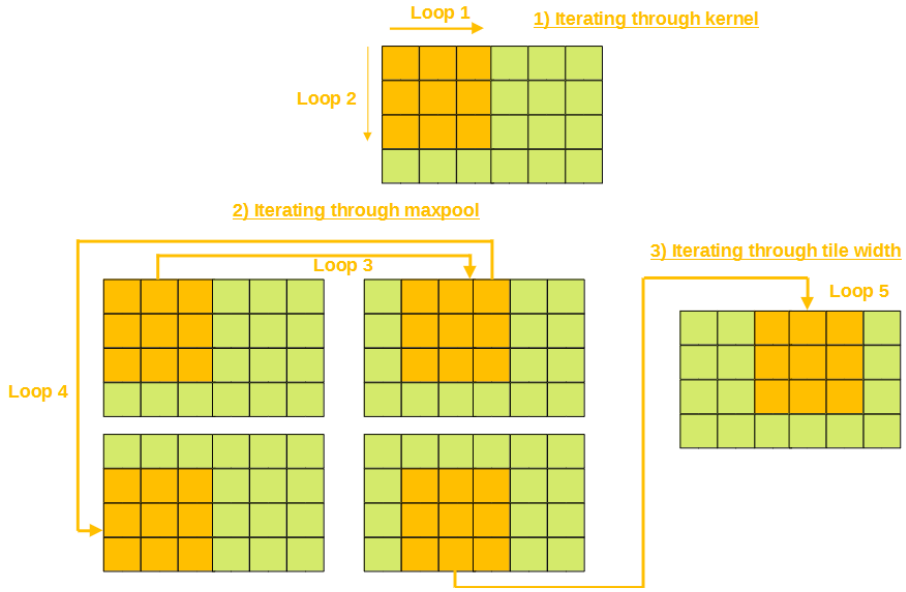

Figure 6.9: Example of 5 loops for simultaneous convolution and maxpool computation.

The yolo layers are also computed in the same run as the previous convolutional layer as they simply consist of the application of the sigmoid activation function to some of the channels. The route layers are handled the same way as in the software baseline (i.e., pointer manipulation).

Finally, the upsample layer is also coupled with the previous convolutional layer and is handled by configuring the internal and external AGUs of the vWrite FUs. For instance, when writing the result of the computation in the vWrite memories, the internal AGU is configured with a `duty` of two in order to store the same result twice in adjacent positions, which corresponds to upsampling inside the line. Then, the external AGU is configured to send the same line twice to the external memory in adjacent positions, which corresponds to upsampling the line.

6.5.4 Post-CNN

As concluded in Section 6.3.2, the methods to get the detections with a score above the threshold and to apply non-maximum suppression are fully handled in software by the soft-processor. The only method from the post-CNN accelerated by the VersatCNN IP core regards to the drawing of labels and boxes, if any detections are found.

When drawing labels, each label pixel has to be multiplied by the RGB value computed during the setup (each class is associated to a different RGB colour). Hence, the FM tile vRead stores the label and the weight vRead stores the RGB values. Analogously to the pre-CNN, the post-CNN is bounded in communication, therefore, only 4 MACs are used for the computation of the coloured labels (`bypass_adder` parameter set to one). In turn, when drawing boxes, only the RGB values are stored in the FM tile vRead and the weight vRead is not used (`bypass` parameter set to one).

6.6 Final remarks

The software baseline of the YOLOv3-Tiny detector is executed on the IOb-SoC platform, which includes additional peripherals such as a timer to profile the embedded code, and the Ethernet module to send the network parameters to the FPGA. Several optimizations were implemented in the source code, namely the approximation of activation functions, batch-normalization folding and post-training quantization. The final fixed-point model presents a mAP_{50} of 30.8 and takes over 16 minutes per frame running on the CPU only. The functionalities of the VersatCNN IP core are tested by accelerating the YOLOv3-Tiny detector. To achieve the target frame rate of 30 FPS, the IP core is pre-configured with a total of 16 weight memories of 32kB, 13 FM tile memories of 64kB and 208 custom FUs, each with 4 MACs, making a total parallelism factor of 832. The FM tiles are chosen so the communication time is lower than the computation time for each run. The IOb-SoC-Yolo system integrates the VersatCNN IP core with the IOb-SoC platform and is responsible for executing the final firmware for accelerating the full YOLOv3-Tiny detector (i.e., including pre and post-CNN). The next chapter presents the performance results achieved by IOb-SoC-Yolo.

Chapter 7

Results

This chapter reports the performance results achieved for accelerating the YOLOv3-Tiny detector within the IOb-SoC-Yolo platform. After describing the target device used, the results are presented in terms of the FPGA resource consumption, the execution time of the detector and a comparison with other FPGA-based works.

7.1 Target device

The development board available for this work is the Kintex UltraScale KU040 [32]. Within the features provided by this board, the most relevant for this work include:

USB-UART interface: for terminal communications between the board and the external PC. This interface is used by the bootloader of the SoC platform described in Section 6.1 and for debugging.

Ethernet PHY interface: to implement high-speed data links (in this case, 100 Mbps). This interface is needed to transfer the parameters of the YOLOv3-Tiny CNN, which would be too slow over UART.

DDR4 SDRAM: this 1GB external memory stores the data transferred through the Ethernet interface and the intermediate results, as they do not fit in the FPGA on-chip memory.

Xilinx XCKU040 FPGA: the available resources of this FPGA are represented in Table 7.1.

Table 7.1: Available resources of Xilinx XCKU040 FPGA.

36Kb BRAM	FF	LUT Logic	LUT memory	DSP
600	484,800	242,400	112,800	1,920

The main primitives of this FPGA include Look-Up Tables (LUTs), Flip-Flops (FFs), Block RAMs (BRAMs) and DSPs. A LUT is a small RAM that stores a function truth table. In the UltraScale FPGAs, the LUTs have 6 inputs (LUT6), which is equivalent to 64 bits RAM. The BRAMs provide on-chip data storage and can be configured as two independent 18Kb RAMs or one 36Kb RAM and as simple or true dual port. When mapping the VersatCNN IP core into the FPGA, the vWrite and the bias memories are implemented using LUTRAMs whilst the other vRead memories are implemented with BRAMs.

The UltraScale FPGAs have DSP48E2 blocks, whose internal constitution is represented in Figure 7.1. The main function of the DSP block is *P = (A + D) x B + C* which consists of a 27-bit pre-adder, a 27x18 multiplier and a 48-bit ALU (post-adder/subtracter, accumulator or logic unit). Each MAC of the VersatCNN IP core is implemented in the FPGA by one DSP (due to 16-bit data as concluded in Table 2.2) with 4 pipeline stages, no pre-adder and the ALU configured as an accumulator.

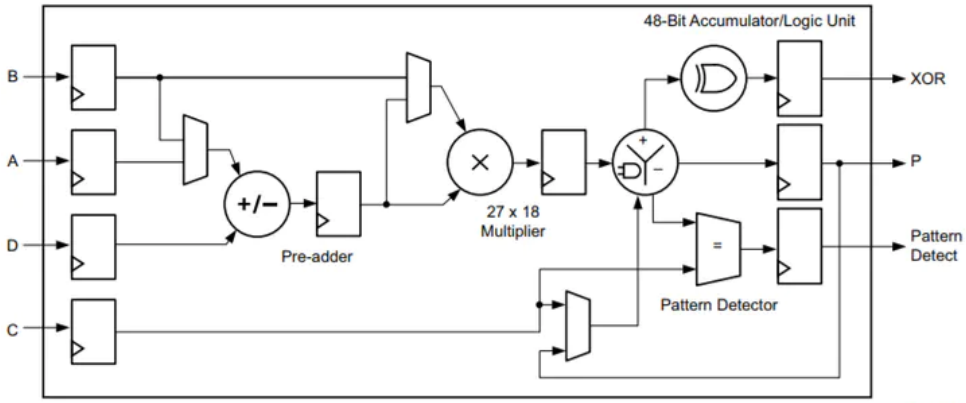

Figure 7.1: DSP48E2 internal constitution (adapted from [33]).

7.2 Resource consumption

Table 7.2 presents the resource consumption in terms of the FPGA primitives of the IOb-SoC-Yolo system, highlighting the resources required by the hardware accelerator developed in the scope of this work. Most of the resources are occupied by the hardware accelerator.

Table 7.2: IOb-SoC-Yolo resource consumption.

Component	36Kb BRAM	FF	LUT Logic	LUT memory	DSP
AXI Interconnect	0	9,887	1,366	2,176	0
DDR4 Controller	25.5	11,918	9,245	452	3
RISC-V CPU	0	902	2,521	48	4
Internal memory	17	41	60	0	0
AXI Cache	1	592	349	280	0
VersatCNN IP	339	86,319	103,655	16,792	871
Ethernet	1	382	193	0	0
UART	0	89	86	0	0
Timer	0	130	2	0	0
Others	0	728	448	32	0
Total	383.5 (64%)	110,988 (23%)	119,166 (49%)	19,780 (18%)	878 (46%)

Overall, the design requires less than half of the resources available at the target device, with the exception of the BRAMs, which are used as on-chip memory to store all channels from 3D kernels and input FM tiles in order to perform 3D convolutions in a single run. The implementation of shared configurations between the same type of FUs allowed the system to be scalable in terms of both Flip-Flop and LUT consumption. Only 46% of the DSPs are used, thus, the parallelism factor could still be doubled from 832 to 1664 if requiring a higher target frame rate.

Regarding the timing characteristics, for a frequency of 143 MHz, the post-router timing reports a Worst Negative Slack (WNS) of 0.05 ns and a Worst Hold Slack (WHS) of 0.03 ns, therefore, the final system meets the timing constraints for a clock period of 7 ns.

7.3 Execution time

The execution time of the YOLOv3-Tiny detector implemented over the IOb-SoC-Yolo platform is represented in Table 7.3. The total execution time is 30.9 ms, exceeding the target frame rate of 30 FPS.

Table 7.3: Performance of the YOLOv3-Tiny detector in the IOb-SoC-Yolo platform.

Section	Execution time (us)	Section	Execution time (us)
Width resize (pre-CNN)	1,089	Layer 13	6,629
Height resize (pre-CNN)	1,941	Layer 14	539
Layer 1/2	856	Layer 15	1,657
Layer 3/4	1,627	Layer 16/17	274
Layer 5/6	1,645	Layer 19/20	116
Layer 7/8	1,644	Layer 22	4,966
Layer 9	1,679	Layer 23/24	632
Layer 10	236	Get detections (post-CNN)	1,941
Layer 11	1,642	Filter boxes (post-CNN)	136
Layer 12	267	Draw detections (post-CNN)	1,370
Total time (ms)	30.9		

In comparison with the software baseline (Table 6.3), the pre-CNN was accelerated from 1s to only 3ms and the drawing detections method from the post-CNN was improved from approximately 12ms to nearly 1.4ms, both mainly due to the reduction of the communication time between the FPGA and the external memory by using the DMA engine inside the IP core. The highest speed-up was achieved for the acceleration of the CNN from 968 seconds to only 24.4ms, which is very close to the theoretical value estimated in Section 6.4.1 (i.e., 23.4 ms).

Table 7.4 compares the execution time of the fixed-point model of the YOLOv3-Tiny detector implemented over the IOb-SoC-Yolo platform with the original floating-point model from Darknet executed in both CPU and GPU. IOb-SoC-Yolo is nearly 27 times faster than the CPU version and only 2 times slower than the GPU version, being however more suitable for embedded systems.

Table 7.4: Performance comparison of the YOLOv3-Tiny detector in different platforms.

Platform	Execution time (ms)	FPS
CPU (Intel i7-8700 @ 3.2 GHz)	828.3	1.2
GPU (RTX 2080 Ti)	15.4	64.9
FPGA (IOb-SoC-Yolo)	30.9	32.4

One key aspect when accelerating the detector with the developed IP core was to ensure that the communication time of the weights and FMs between the external memory and the accelerator was lower than the time required to compute the results in each run by choosing adequate width tiling factors. Figure 7.2 compares the communication and computation times of the detector for each CNN layer. In most layers, the computation time doubles the communication time, which means that the CNN could still be accelerated with a higher parallelism factor.

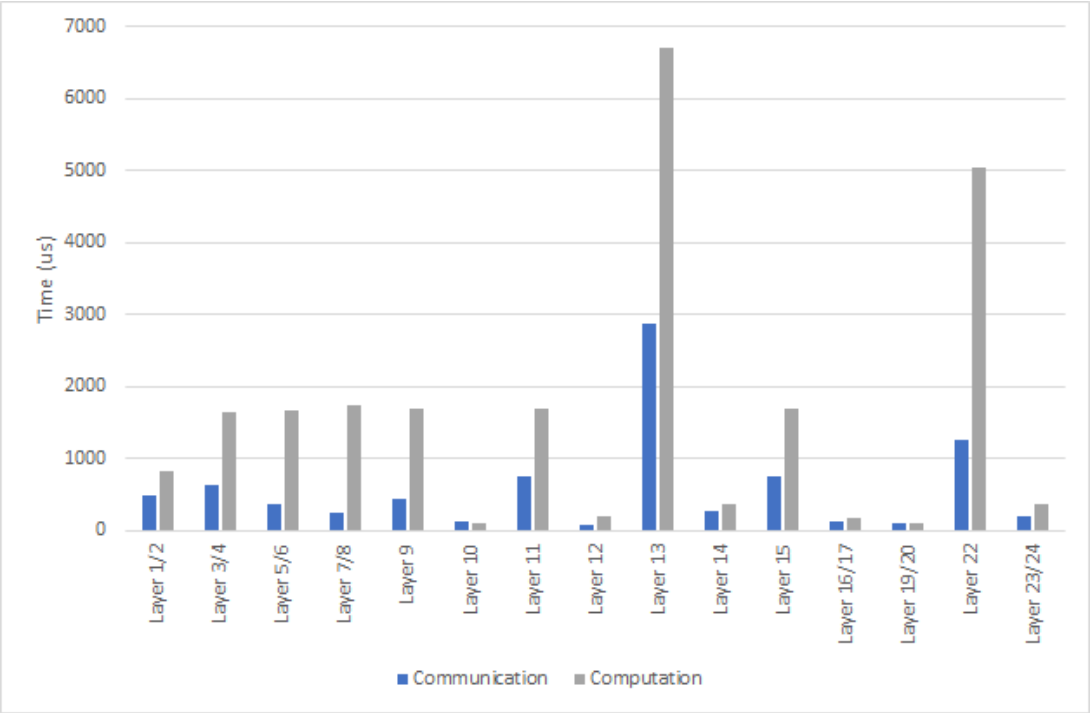

Figure 7.2: Comparison between the computation and communication times.

The execution time of the CNN is always constant, as long as the parallelism factor is kept. On the other hand, the pre-CNN execution time depends on the dimensions of the input image whilst the post-CNN execution time depends on the number of candidate and final detections. For both software baseline and final firmware, the timing results obtained were for an 768x576 input image from which the detector finds 5 candidate boxes and 4 final detections.

Figure 7.3 represents the variation of the execution time of the pre-CNN according to the resolution of the input image. The width resize method is slower for wider input images and the height resize method increases its execution time for squarer input images.

Figure 7.3: Execution time of the pre-CNN depending on the input image resolution.

In turn, Figure 7.4 shows how the post-CNN execution time rises when detecting more candidate boxes and final detections, which could result from the reduction of the threshold score. Note that the increase of the execution time of the detector when requiring a higher input image resolution or a lower score threshold can be compensated with the reduction of the execution time of the CNN by applying a higher parallelism factor, as mentioned above.

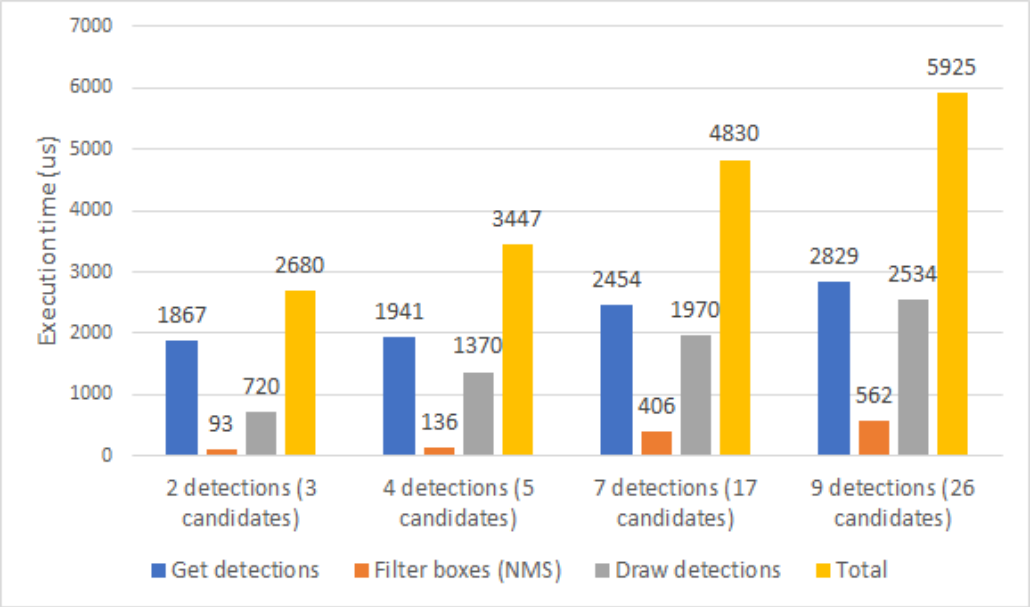

Figure 7.4: Execution time of the post-CNN depending on the number of detections.

7.4 Comparison with FPGA-based implementations

At the time of writing of this document, three others implementations of the YOLOv3-Tiny detector in FPGA are reported in the literature. All three implementations consist of hardware/software co-designs. The IOb-SoC-Yolo is compared with these implementations in Table 7.5.

Table 7.5: Comparison of FPGA-based implementations of the YOLOv3-Tiny detector.

	[34]	[35]	[36]	IOb-SoC-Yolo
FPGA	UltraScale+ XCZU9EG	Virtex-7 XC7VX485T	Zynq 7020	UltraScale XCKU040
Frequency (MHz)	-	200	100	143
LUT (K)	-	49	26	119
BRAM	-	70	93	384
DSP	-	2304	160	878
Images/s	104.2	-	1.9	32.4
Unrolled loops	-	1,2,4	1,2,4	2,3,4
Precision (bits)	8	18	16	16
GMAC/s	-	-	5.3	90
MMAC/s/kLUT	-	-	203.8	756.3
MMAC/s/DSP	-	-	33.1	102.5

In [34], the YOLOv3-tiny model is first trained using the Caffe framework over a dataset for pedestrian signalling and then quantized with 8-bit fixed-point. The backbone network is accelerated by the FPGA whilst the detection layers are handled in software by the hard processor. The author claims a throughput of 104.2 FPS without detailing the hardware architecture or resource consumption.

[35] applies batch-normalization folding and post-training quantization of 18 bits. Only the convolutions are handled in hardware and all the other layers and activation functions are implemented in software. The hardware architecture exploits the inter-FM, inter-convolution and intra-convolution parallelisms with a total parallelism factor of 2304. In comparison with IOb-SoC-Yolo, the total parallelism factor is over 2.5x higher, which would justify a higher throughput. However, the only performance metric reported is the number of MAC operations per second, calculated from the product between the number of DSPs and the frequency. Therefore, the actual throughput is not reported and no fair performance comparison can be done between the two works.

[36] accelerates all YOLOv3-Tiny layers in hardware. In comparison with IOb-SoC-Yolo, the intra-convolution parallelism (loop 1) is exploited instead of the intra-FM (loop 3), the data is also quantized with 16 bits and the throughput is about 17x lower. Note that the Zynq 7020 device has between 4 and 8x less hardware resources than the UltraScale XCKU040 (depending on the specific FPGA primitive). Both works perform 16-bit MAC operations and, based on the MAC operations per second, IOb-SoC-Yolo has better performance and also better area efficiency in terms of LUT and DSP consumption.

The IOb-SoC-Yolo platform is constituted by a soft-processor with a lower performance and executed at a lower frequency in comparison with the hard processor present in the other works. Still, the system is capable of achieving a real-time performance as the software development is simplified at the expense of the features supported by the hardware accelerator. For instance, the integration of the DMA frees the CPU from handling data transfers to only execute the configurations of the FUs. The number of configurations are invariant to the number of FUs in the design, consequently reducing the configuration time and the size of the firmware code. The hardware also abstracts the CPU from managing the configurations in pipeline fashion and the ping-pong operation of the memories.

All the other works only focus on the acceleration of the CNN part of the YOLOv3-Tiny detector. In turn, this work executes the full process flow of the detector by adding the image resize prior to the CNN and the drawing of the detections after the CNN. The reconfigurable interconnections of the VersatCNN IP core allow to form different datapaths to accelerate more than only CNNs. As a result, the core is also able of accelerating the pre and post-precessing parts of the detector.

7.5 Final remarks

Most of the target FPGA resources used by the IOb-SoC-Yolo system are occupied by the VersatCNN IP core. Less than half of the majority of the resources available are required, which means that higher parallelism can still be exploited if requiring a higher target frame rate. The system achieves 32.4 FPS for the full YOLOv3-Tiny detector (i.e., including pre and post-CNN), which is nearly 27x faster than the original floating-point model running on an external CPU and only 2x slower than the GPU version. In comparison with other implementations of the YOLOv3-Tiny detector in FPGAs, IOb-SoC-Yolo presents better performance and better area efficiency for the case where a fair comparison is possible. The real-time performance obtained demonstrates that the VersatCNN IP core is a valid solution for accelerating CNN-based networks.

Chapter 8

Conclusions

This work presents the development of a new extension for the Versat reconfigurable processor called VersatCNN, which is optimized for the acceleration of CNNs. This hardware accelerator is added as another peripheral in the IOb-SoC platform with the purpose of accelerating the YOLOv3-Tiny object detector. The IP core is parameterized taking into account the characteristics of the CNN and the available resources in the target device, achieving a performance over 30 FPS.

8.1 Achievements

The state-of-art object detectors are firstly studied and divided in two categories: two-stage and one-stage detectors. The two-stage detectors achieve higher localization and object recognition accuracy. In turn, one-stage detectors obtain higher inference speed. Besides of being the detector with the best trade-off between accuracy and execution time, YOLOv3 includes an alternative network for constrained environments called YOLOv3-Tiny, which is the one accelerated in this work.

The source code from the Darknet framework is reduced to cover only the YOLOv3-Tiny detector and adapted for resource-constrained embedded systems, which includes the removal of dynamic memory allocation and complex data structures. The activation functions are approximated by replacing the multiplications with adders and shifts. The batch-normalization layers are fused with their previous convolutional layers, thereby reducing the number of parameters and operations of the network. The original floating-point parameters are quantized for inference using dynamic fixed-point quantization with a precision of 16 bits. As a result, the fixed-point model presents a mAP_{50} drop of only 2.1 in comparison with the original floating-point model.

The software-only version of the YOLOv3-tiny detector is executed on the IOb-SoC platform as baseline for the hardware acceleration. This system is composed of several peripherals such as a timer to profile the embedded code, the Ethernet module to send the network parameters to the FPGA and the UART module to receive the firmware and for debugging. The software baseline presents an execution time over 16 minutes and the profiling results show that the pre-CNN, all CNN layers and the drawing detection method of the post-CNN must be accelerated in hardware to achieve the target frame rate.

The Deep Versat CGRA is analysed and consists of a highly configurable and customisable hardware accelerator suitable to speed up loop-based applications. This system presents limitations for the acceleration of CNNs, which lead to development of the VersatCNN IP core for efficient CNN computation. The core is an improvement from Deep Versat but inherits features such as the configuration register files and shadow registers and the AGUs. The improvements involve the implementation of vector FUs that share configurations between the same type of FUs, the integration of a DMA for fast data transfers, heterogeneous stages, automatic ping-pong memories and higher loop-level AGUs.

VersatCNN is composed of vRead FUs for reading weights and FM tiles from the external memory, custom FUs for computation and vWrite FUs for writing the results back to the external memory. The reconfigurable connections inside the custom FUs allow to form different datapaths to accelerate different CNN layers and activation functions. The custom MAC-based FUs are structured in a matrix form to exploit three types of parallelism (Inter-FM, Intra-FM and Inter-Convolution) and to enhance both pixel and weight sharing. The core embraces a total of 72 run-time configurable parameters.

The IP core is tested for the acceleration of the YOLOv3-Tiny detector in a FPGA. Its architecture is pre-configured with a total of 16 weight memories of 32kB, 13 FM tile memories of 64kB and 208 custom FUs, each with 4 MACs, for a total parallelism factor of 832. The yolo, upsample and most of the maxpool layers are executed alongside the previous convolutional layer. The pre-CNN and the drawing of the detections from the post-CNN are also accelerated by the IP core. As a result, the system composed of IOb-SoC and VersatCNN achieves a performance of 32.4 FPS for the full YOLOv3-Tiny detector, which shows that VersatCNN is a valid solution for accelerating CNN-based networks.

8.2 Future Work

Future developments on this work may include the deployment of a generic infrastructure to demonstrate the object detector in real-time. For instance, the system could include a camera for receiving input images in real-time and an appropriate video interface for displaying the resulting images (e.g., HDMI monitor), instead of transmitting both via Ethernet. Thus, two extra modules for interfacing the camera and the video stream need to be developed and added as peripherals in the IOb-SoC platform.

In turn, the IOb-SoC platform can be improved in terms of the performance of the soft-processor and the Ethernet module. Using a CPU with a lower CPI would accelerate the performance of the post-CNN methods fully executed in software (i.e., getting the detections with score above the threshold and filtering boxes with NMS). The transfer of the CNN parameters via Ethernet still takes minutes as the data received goes through the soft-processor and the cache. Coupling a DMA with the Ethernet module would allow to bypass both soft-processor and cache, reducing the transfer time to seconds.

Finally, the VersatCNN IP core could be further validated by accelerating other CNN-based networks as the core handles the most common layers (i.e., convolutional, non-linear activation, maxpool). This hardware accelerator can also be improved for supporting other type of layers. For example, to accelerate the shortcut layers present in the YOLOv3 network, an optional adder can be placed between each two FM tile vReads to sum pixels from different input FMs prior to the convolution operation.

Bibliography

[1] L. Jiao, F. Zhang, F. Liu, S. Yang, L. Li, Z. Feng, and R. Qu. A Survey of Deep Learning-Based Object Detection. *IEEE Access*, 7:128837–128868, 2019. ISSN 2169-3536. doi: 10.1109/ACCESS.2019.2939201.

[2] L. Liu, W. Ouyang, X. Wang, P. W. Fieguth, J. Chen, X. Liu, and M. Pietikäinen. Deep Learning for Generic Object Detection: A Survey. *International Journal of Computer Vision*, pages 1 – 58, 2018.

[3] Z. Zhao, P. Zheng, S. Xu, and X. Wu. Object Detection With Deep Learning: A Review. *IEEE Transactions on Neural Networks and Learning Systems*, 30(11):3212–3232, Nov 2019. ISSN 2162-2388. doi: 10.1109/TNNLS.2018.2876865.

[4] E. Unlu, E. Zenou, N. Riviere, and P.-E. Dupouy. Deep learning-based strategies for the detection and tracking of drones using several cameras. *IPSJ Transactions on Computer Vision and Applications*, 11:1–13, 2019.

[5] R. Simhambhatla, K. Okiah, S. Kuchkula, and R. Slater. Self-Driving Cars: Evaluation of Deep Learning Techniques for Object Detection in Different Driving Conditions. *SMU Data Science Review*, 2(1), 2019.

[6] N. O. Mahony, S. Campbell, A. Carvalho, S. Harapanahalli, G. Velasco-Hernandez, L. Krpalkova, D. Riordan, and J. Walsh. Deep Learning vs. Traditional Computer Vision. *Advances in Computer Vision Proceedings of the 2019 Computer Vision Conference (CVC)*, pages 128–144, Oct. 2019. doi: 10.1007/978-3-030-17795-9.

[7] X. Feng, Y. Jiang, X. Yang, M. Du, and X. Li. Computer vision algorithms and hardware implementations: A survey. *Integration*, 69:309–320, 2019. ISSN 0167-9260. doi:10.1016/j.vlsi.2019.07.005.

[8] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 45–54. ACM, 2017. doi: 10.1145/3020078.3021736.

[9] 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.

[10] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry. Accelerating CNN inference on FPGAs: A Survey, 2018.

[11] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang. A Survey of FPGA-Based Neural Network Accelerator, 2017.

[12] I. Bae, B. Harris, H. Min, and B. Egger. Auto-Tuning CNNs for Coarse-Grained Reconfigurable Array-Based Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2301–2310, Nov 2018. ISSN 1937-4151. doi: 10.1109/TCAD.2018.2857278.

[13] B. Zhao, M. Wang, and M. Liu. An energy-efficient coarse grained spatial architecture for convolutional neural networks AlexNet. *IEICE Electronics Express*, 14(15):20170595–20170595, 2017. doi: 10.1587/elex.14.20170595.

[14] V. Mário. Deep Versat: A Deep Coarse Grain Reconfigurable Array. Master's thesis, Instituto Superior Técnico, November 2019.

[15] P. Miranda. CGRA-Based Deep Neural Network For Object Identification. Master's Thesis, Jan 2021.

[16] J. Redmon and A. Farhadi. YOLOv3: An Incremental Improvement, 2018.

[17] J. Redmon. Darknet. `https://github.com/pjreddie/darknet`, 2020.

[18] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, and 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, 2016. doi: 10.1145/2847263.2847265.

[19] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, page 161–170, 2015. doi: 10.1145/2684746.2689060.

[20] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao. Throughput-Optimized OpenCL-Based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, page 16–25, 2016. doi: 10.1145/2847263.2847276.

[21] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You Only Look Once: Unified, Real-Time Object Detection, 2015.

[22] J. D. Lopes and J. T. Sousa. Versat, a Minimal Coarse-Grain Reconfigurable Array. In *VECPAR*, 2016.

[23] M. P. Véstias, R. P. Duarte, J. T. de Sousa, and H. C. Neto. A fast and scalable architecture to run convolutional neural networks in low density FPGAs. *Microprocessors and Microsystems*, 77: 103136, 2020. ISSN 0141-9331. doi: https://doi.org/10.1016/j.micpro.2020.103136.

[24] IObundle. IOb-SoC. `https://github.com/IObundle/iob-soc`, 2020.

[25] C. Wolf and et. al. PicoRV32 - A Size-Optimized RISC-V CPU. `https://github.com/cliffordwolf/picorv32`, 2019.

[26] K. Cheng. RISC-V GNU Compiler Toolchain. `https://github.com/riscv/riscv-gnu-toolchain`, 2020.

[27] J. Roque. Development Environment for a RISC-V Processor: Cache. Master's Thesis, Jan 2021.

[28] I. Tsmots, O. Skorokhoda, and V. Rabyk. Hardware Implementation of Sigmoid Activation Functions using FPGA. In *2019 IEEE 15th International Conference on the Experience of Designing and Application of CAD Systems (CADSM)*, pages 34–38, 2019. doi: 10.1109/CADSM.2019.8779253.

[29] P. Kluska and M. Zieba. Post-training Quantization Methods for Deep Learning Models. In N. T. Nguyen, K. Jearanaitanakij, A. Selamat, B. Trawiński, and S. Chittayasothorn, editors, *Intelligent Information and Database Systems*, pages 467–479, Cham, 2020. Springer International Publishing. ISBN 978-3-030-41964-6.

[30] Microsoft COCO datasets. `https://cocodataset.org/#download`, 2020.

[31] CodaLab. COCO Detection Challenge (Bounding Box). `https://competitions.codalab.org/competitions/20794#participate`, 2020.

[32] Avnet. Kintex UltraScale KU040 Development Board: Hardware User Guide, December 2015.

[33] Xilinx. UltraScale Architecture DSP Slice, September 2020.

[34] S. Oh, J. H. You, and Y. K. Kim. Implementation of Compressed YOLOv3-tiny on FPGA-SoC. In *2020 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia)*, pages 1–4, 2020. doi: 10.1109/ICCE-Asia49877.2020.9277266.

[35] A. Ahmad, M. A. Pasha, and G. J. Raza. Accelerating Tiny YOLOv3 using FPGA-Based Hardware/Software Co-Design. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2020. doi: 10.1109/ISCAS45731.2020.9180843.

[36] Z. Yu and C.-S. Bouganis. A Parameterisable FPGA-Tailored Architecture for YOLOv3-Tiny. In F. Rincón, J. Barba, H. K. H. So, P. Diniz, and J. Caba, editors, *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, pages 330–344, Cham, 2020. Springer International Publishing. ISBN 978-3-030-44534-8.