TÉCNICO LISBOA

Object Detection and Classification on the Versat Reconfigurable Processor

Daniel Garigali Pestana

Electrical and Computer Engineering

Supervisor(s): Prof. Horácio Cláudio Neto
Prof. José Teixeira de Sousa

January 2020

Contents

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].

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 being 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.

¡¡¡¡¡¡¡ HEAD One motivation behind this work is the development of an object detection dedicated system that minimizes energy consumption and maximizes performance with reduced hardware size and cost. The deployment of a real-time FPGA-based general-purpose object detector is a major and current challenge. This detector is entitled to be accelerated using the Deep Versat CGRA architecture [14], which was developed at the INESC-ID Research Institute.

Most FPGA-based object detectors in literature use development boards. Some systems [15, 16] receive images from a camera using specialized protocols and transfer their results (i.e., detections) to a host PC through well known interfaces such as UART or Ethernet. Other works install an operating system in the board to harness the integration of the input camera and the output video streaming [17]. ======= One motivation behind this work is the development of an object detection dedicated system that minimizes energy consumption and maximizes performance with reduced hardware size and cost. The deployment of a real-time FPGA-based general-purpose object detector is a major and current challenge. This detector will be accelerated using the Deep Versat CGRA architecture [14], which was developed at the INESC-ID Research Institute.

Most FPGA-based object detectors in previous works use development boards. Some systems [15, 16] receive images from a camera using specialized protocols and transfer their results (i.e., detections) to a host PC through well known interfaces such as UART or Ethernet. Other works install an operating system in the board to harness the integration of the input camera and the output video streaming [17]. ¿¿¿¿¿¿¿ 69e43af3d375f9039aaeb0b9750de9dd40718dde

Another motivation behind this work is to develop a generic infrastructure where different FPGA-based object detectors can be tested. This platform should be capable of receiving images from a camera and displaying the resulting images through an appropriate video interface for demonstrations in development boards, without requiring external hardware or additional software.

1.2 Objectives

The training of DNNs is typically performed on cloud servers where there are no strict limitations in terms of energy consumption, computational power and memory capacity [9]. On the other hand, the inference is desired in embedded systems near the sensor to reduce communication latency and security risks.

Thus, the main objective of this report is to propose a methodology for deploying a real-time FPGA-based general-purpose object detector, over a generic infrastructure, using a pre-trained DNN. This

implies studying the state-of-art object detectors, besides understanding how DNNs are deployed in FPGAs and CGRAs. A brief description about the Deep Versat CRGA architecture is also considered.

1.3 Report Outline

This report is organized as follows. Chapter 2 introduces DNNs and shows how they have been 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. Chapter 5 presents the proposed methodology and the expected work-plan/results.

Chapter 2

Deep Neural Networks

This chapter introduces the concept of Deep Neural Networks (DNNs) and proceeds to Convolutional Neural Networks (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. Field Programmable Gate Arrays (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 Fig. 2.1.

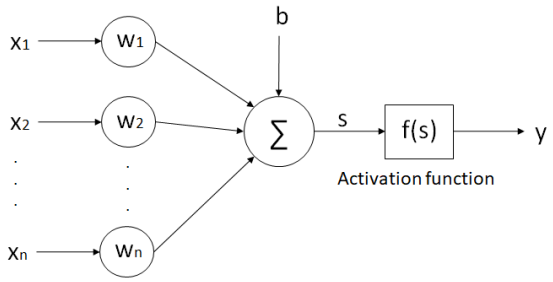

Figure 2.1: Implementation of a neuron.

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 Fig. 2.2.

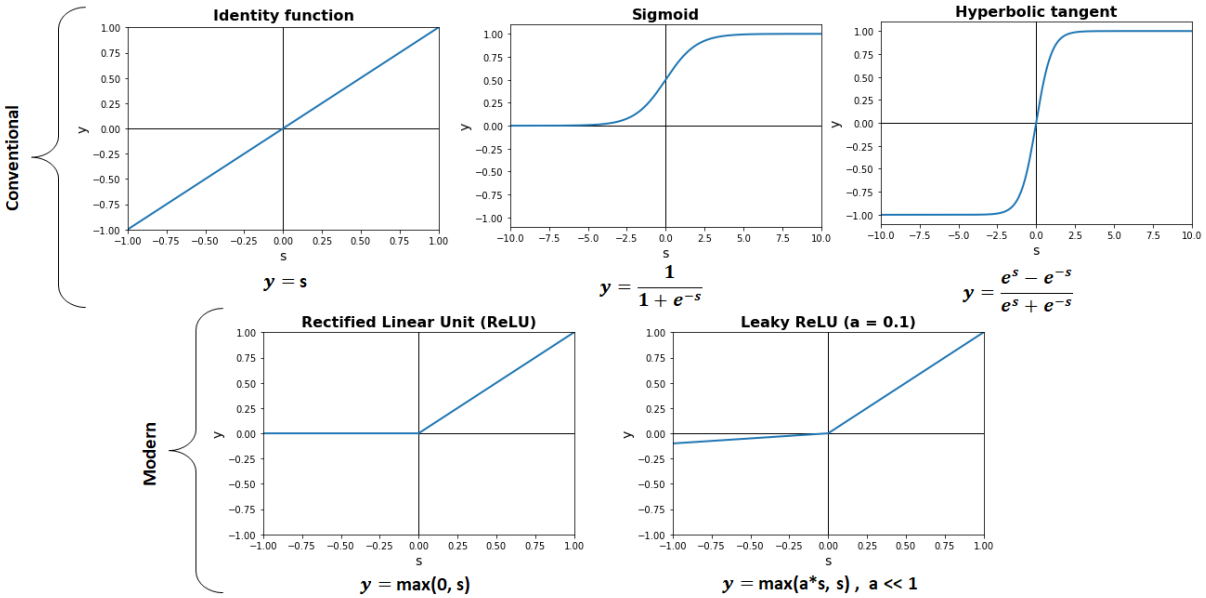

Figure 2.2: Plot and expression of the most common activation functions.

Neural networks are composed by 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. Fig. 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.

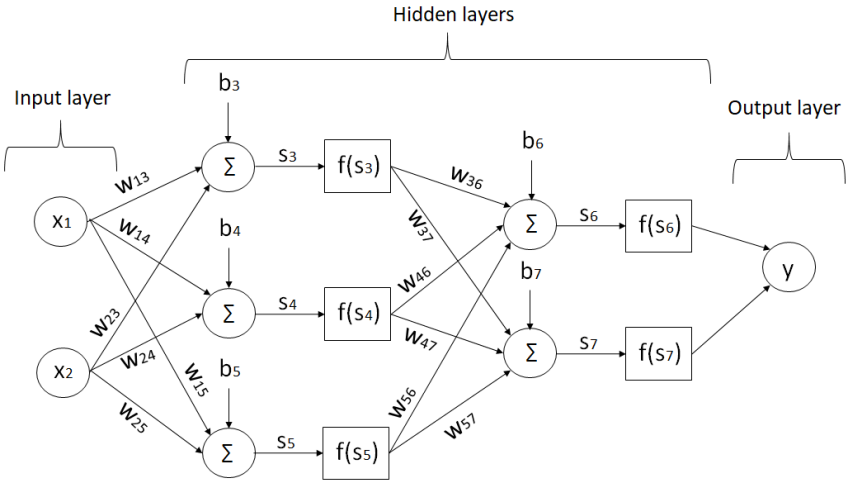

Figure 2.3: Example of a fully connected DNN.

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 Fig. 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 by 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.

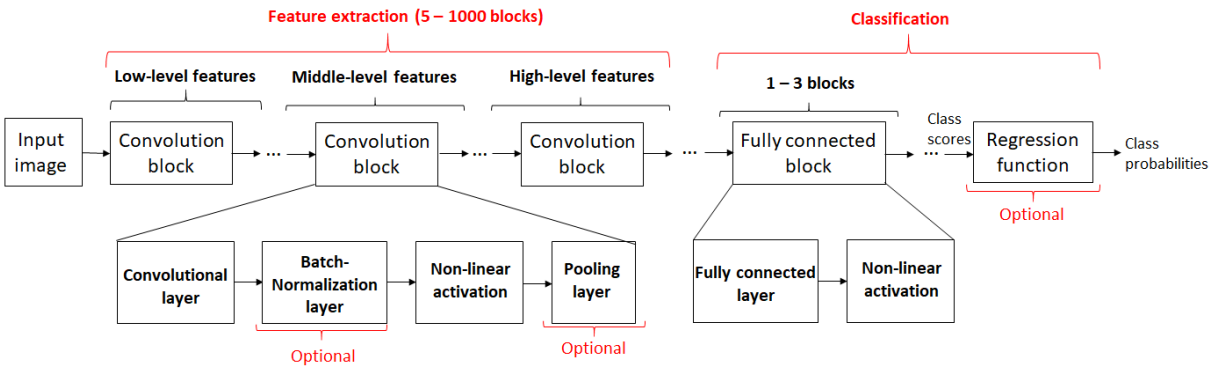

Figure 2.4: Constitution of a typical CNN.

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 multiply and accumulate (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. Fig. 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.

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

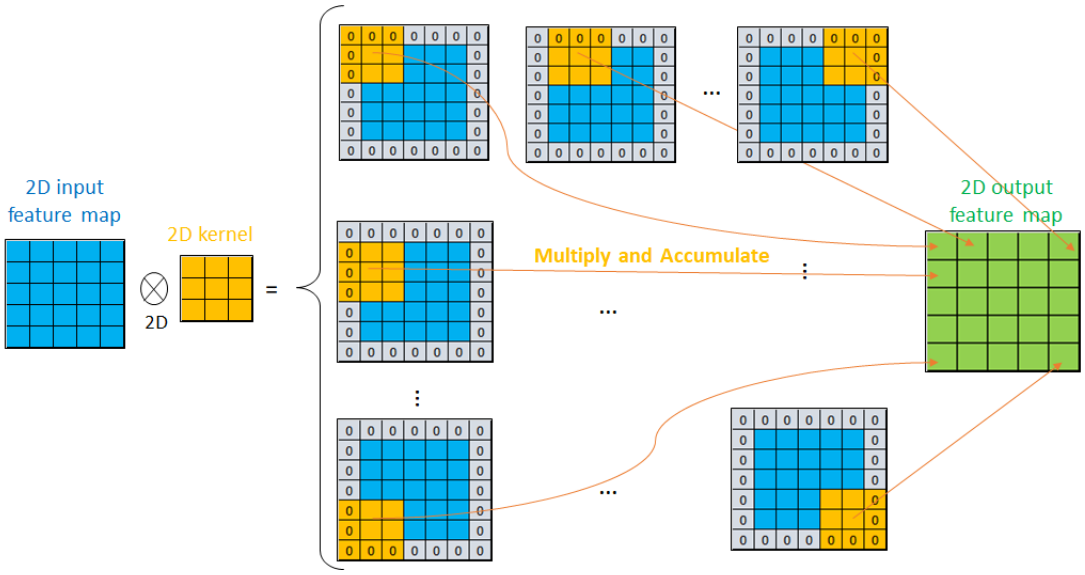

Figure 2.5: Example of an 5x5 input feature map and 3x3 kernel 2D convolution.

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. Fig. 2.6 exemplifies a 3D convolution between an 5x5 input feature map and two 3x3 kernels, all with 3 channels and zero-padding.

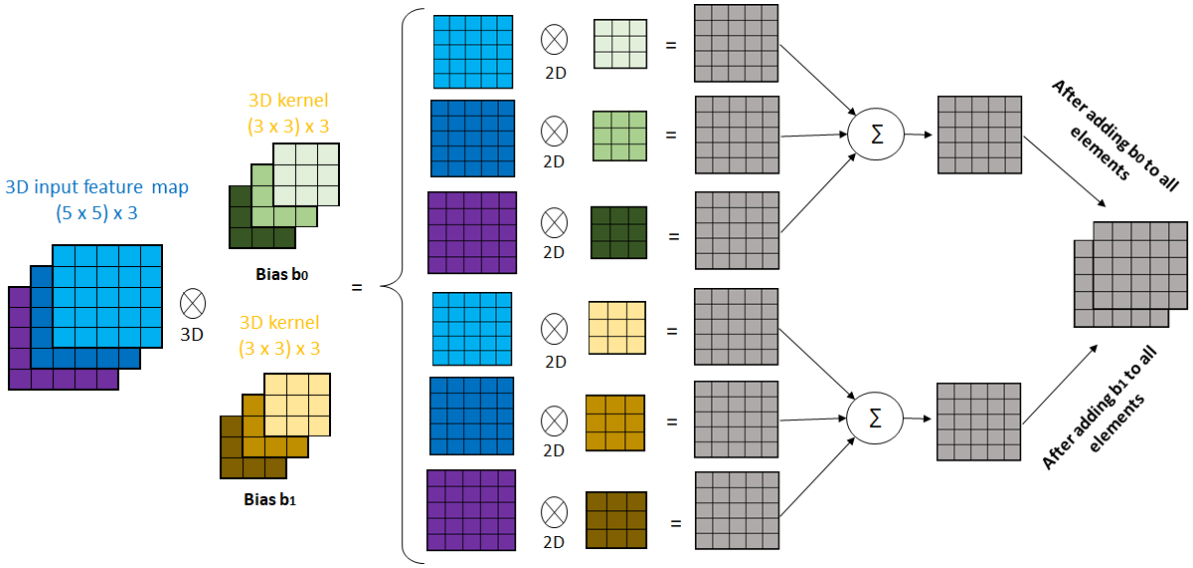

Figure 2.6: Example of an 5x5 input feature map and two 3x3 kernels 3D convolution (3 channels).

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 parameters in the next layers. Each 2D channel is divided into non-overlapping blocks, which are further replaced by the maximum (max-pooling) or the mean (average pooling) value of the block. The most common operation is a 2x2 max-pooling, as shown in Fig. 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].

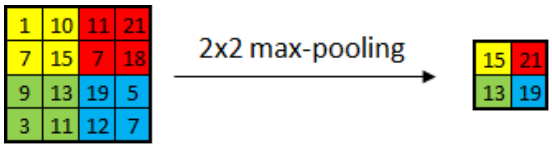

Figure 2.7: Example of 2x2 max-pooling.

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. Fig. 2.8 exemplifies a shortcut layer generated from adding the output from 2 layers before.

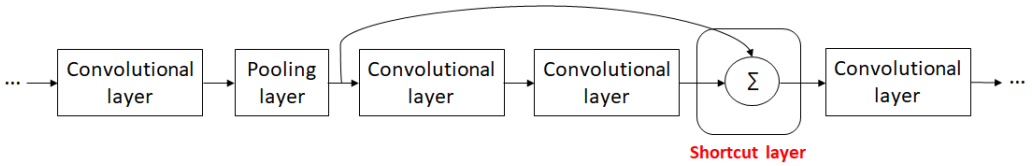

Figure 2.8: Example of a shortcut layer.

2.2.5 Route and upsample layers

Route and upsample layers were introduced for CNNs focused on object detection tasks [18]. 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. Fig. 2.9 exemplifies the simplest way to upsample a 2x2 feature map by a factor of two.

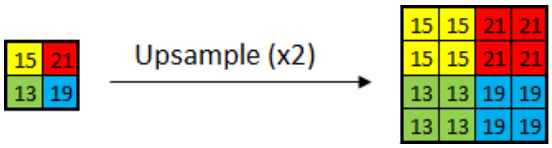

Figure 2.9: Example of upsampling by a factor of 2.

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 [19]. 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 Fig. 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 [18] is a more recent CNN model that also employs the shorcut layers first introduced by ResNet to allow a deeper network.

2.5 FPGA-based CNNs

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 CPUs or GPUs, is the flexibility to design costumed 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. Consequently, 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 [20–22] introduce several levels of memory hierarchy, as shown in Fig. 2.10. The system is composed by two on-chip input buffers, one for fetching the feature maps and the other for fetching the parameters (i.e., weights and bias) 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.

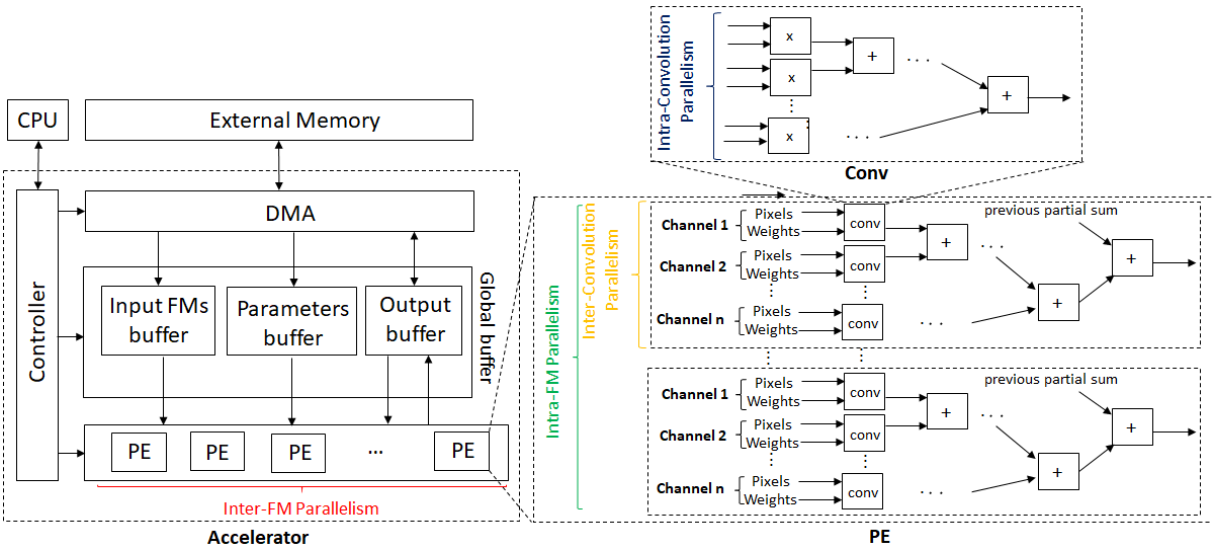

Figure 2.10: Typical FPGA-based CNN accelerator (adapted from [10]).

Fig. 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 Fig. 2.11.

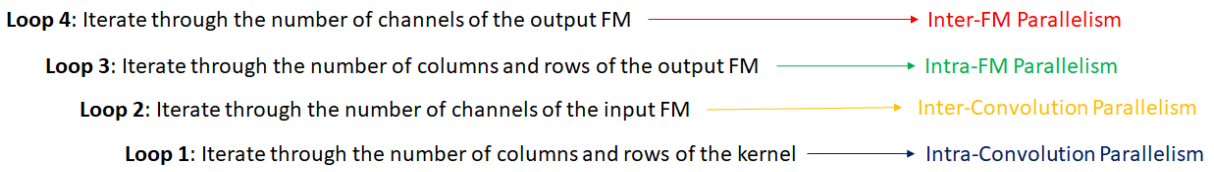

Figure 2.11: Association between loops and source of parallelism.

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 Fig. 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]. Fig. 2.12 shows that the weights and the pixels can be reused by unrolling loops 3 and 4 respectively [8].

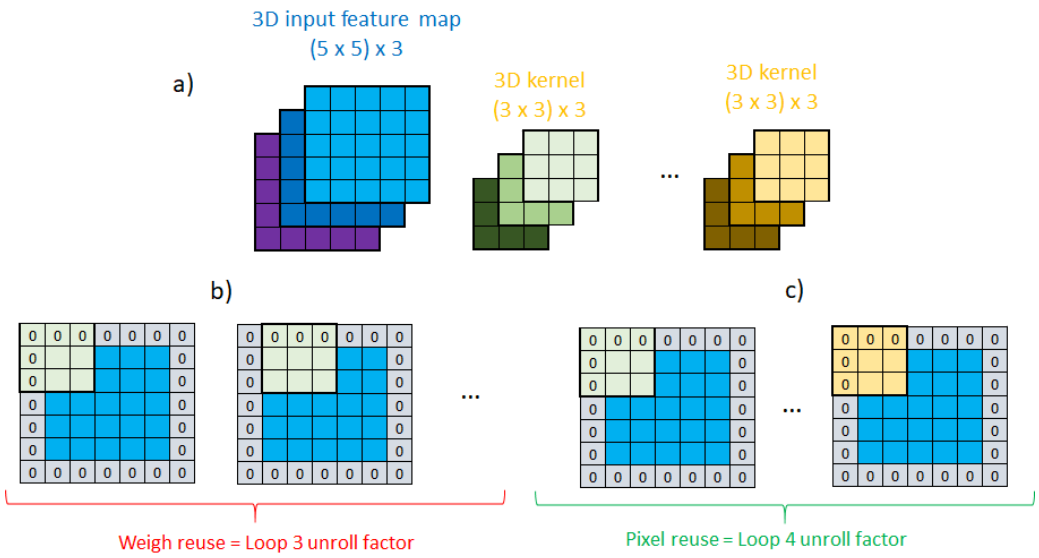

Figure 2.12: a) 3D convolution between one FM and several kernels with b) weight and c) pixel reuse.

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]. Fig. 2.13 exemplifies the division of the feature maps and kernels when tiling all loops.

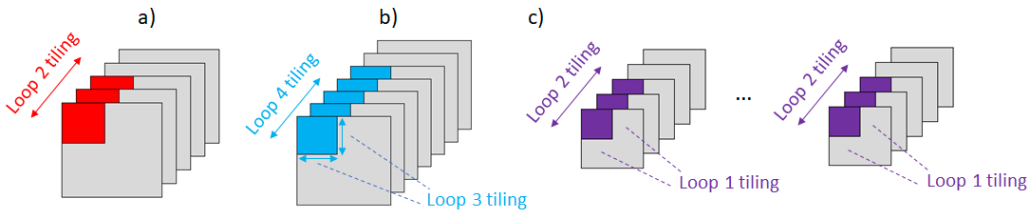

Figure 2.13: Example of loop tiling in the a) input FMs, b) output FMs and c) kernels.

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. Fig. 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

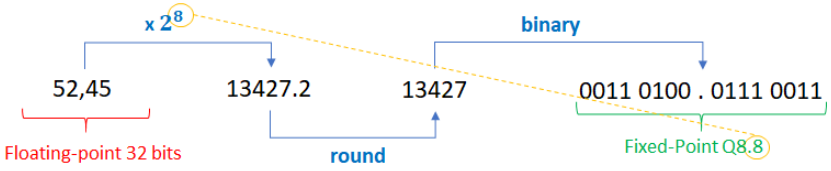

Figure 2.14: Example of conversion from 32 bits floating-point to Q8.8 fixed-point.

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 feature maps [11]. Moreover, 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.

[21] 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 computational throughput

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, [22] reached a computational throughput 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 [20] for the VGG-16 network, achieving a throughput 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 throughput of 645.25 GOPs.

Table 2.3 compares the four FPGA-based accelerators in terms of resource consumption, optimization strategy and throughput. All these approaches unroll loop 4 by employing several PEs. [21] has the major DSP consumption mainly due to using 32-bit floating point operands. [8] presents 3 times higher throughput than [20] 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 a design exploration is essential for achieving 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 [21]	AlexNet [22]	VGG-16 [20]	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
Throughput (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

Final remarks

CNNs are composed by 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. A similar study will need to 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 Fig. 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 the section 2.4, excluding the last fully connected layers [1].

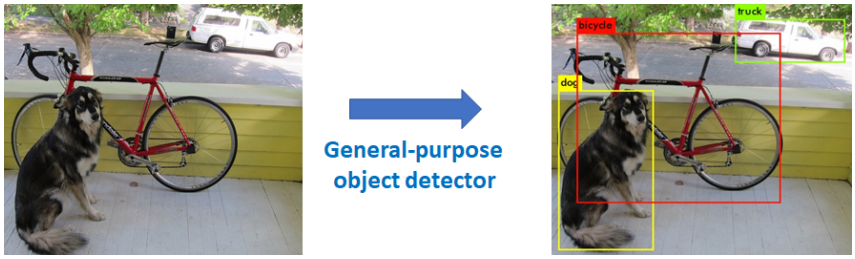

Figure 3.1: Example of bounding box usage to locate objects in an image (adapted from [19]).

3.1 Benchmarks and metrics

Two of the most common benchmarks for general-purpose object detection are PASCAL VOC 2007/2012 and Microsoft COCO [3]. The official metric for measuring the performance of detectors is based on the mean average precision (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 Fig. 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].

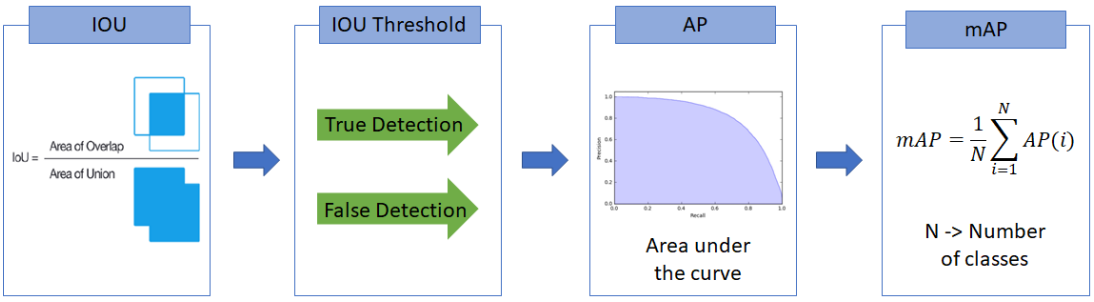

Figure 3.2: Steps for obtaining the mean average precision (adapted from [4]).

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 by 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 R-CNN, Fast R-CNN, Faster R-CNN, Mask R-CNN and 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 Yolo and its successors Yolov2 and Yolov3, SSD and its successor 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, 18]	—	28	45.4	61	16.4	ResNet-101	
	SSD-513 [1, 7, 18]	76.8	31.2	50.4	125	8		
	DSSD-321 [1, 3, 18]	78.6	28	46.1	85	11.8		
	DSSD-513 [1, 18]	—	33.2	53.3	156	6.4		
	YOLOv3-320 [18]	—	28.3	51.5	22	45.5	DarkNet-53	
	YOLOv3-416 [18]	—	31	55.3	29	34.5		
	YOLOv3-608 [18]	—	33	57.9	51	19.6		
	RetinaNet-500 [1, 18]	—	34.4	53.1	90	11.1	ResNet-101	M40 GPU
	RetinaNet-800 [1, 18]	—	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). Therefore, this is the object detector that will be implemented in the scope of this work.

3.3 YOLOv3 detector

Fig. 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.

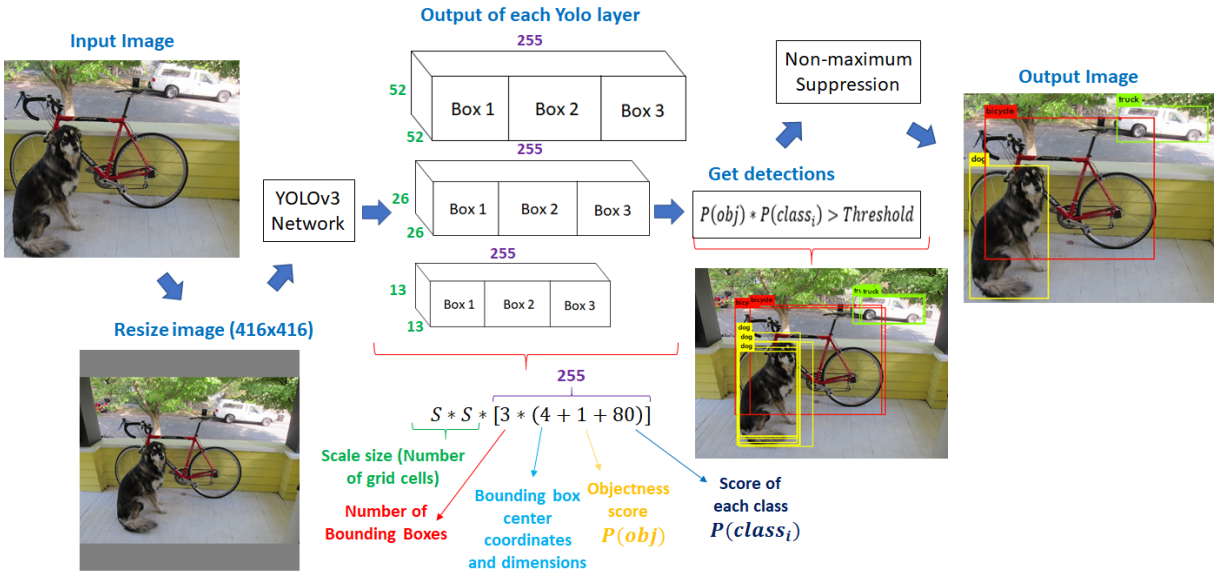

Figure 3.3: YOLOv3 process flow.

3.3.1 YOLOv3 network

The CNN-based YOLOv3 network is represented in Fig. 3.4. This network is composed by 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.

This network is inspired on some concepts from popular DNN models. For instance, the kernels are mostly 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.

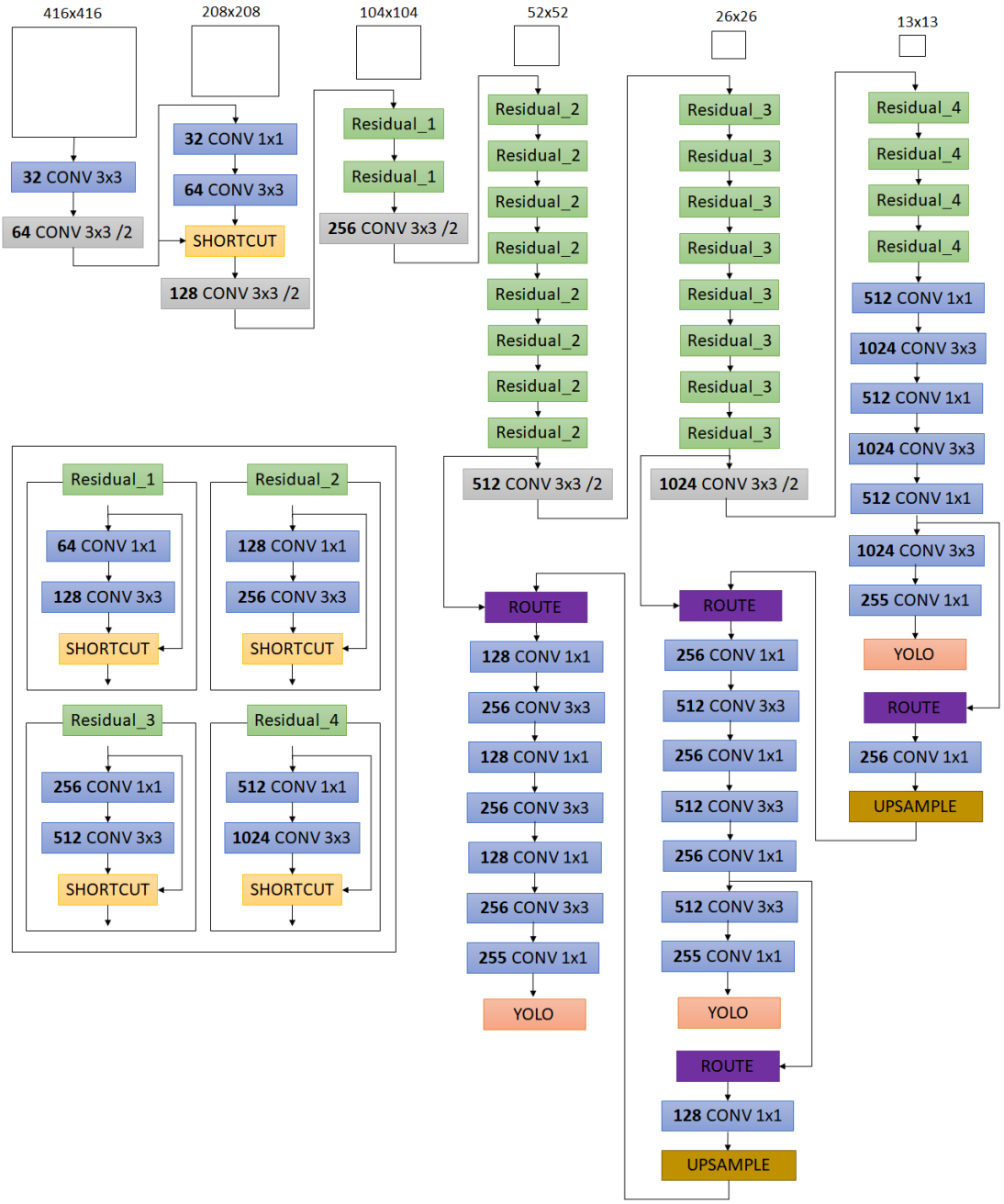

Figure 3.4: YOLOv3 Network.

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 Fig. 3.5, 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.

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 [18]. The sizes are then appropriately adjusted by the network. Each bounding box

consists in the predictions of [19]: 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 [18], the total number of classes is 80. Thus, each grid cell is composed by $3 * (2 + 2 + 1 + 80) = 255$ values, as specified in Fig. 3.6.

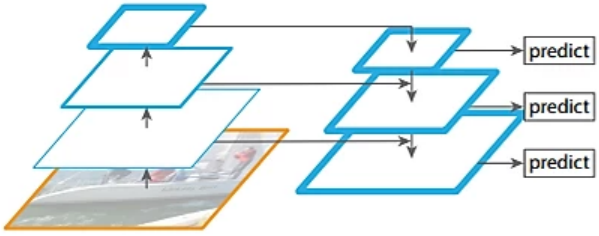

Figure 3.5: Feature Pyramid Network structure [3].

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 application of the logistic activation in the predictions of each bounding box, excluding the width and height parameters, is performed by the **yolo layers**. This new layer added by the YOLOv3 network is used as the ending layer for each scale.

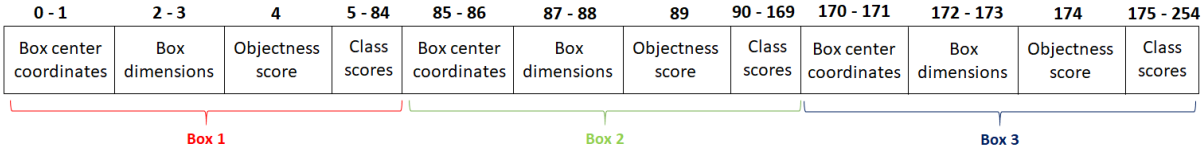

Figure 3.6: Constitution of each grid cell.

3.3.2 Detections phase

After executing the YOLOv3 network block (Fig. 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 three different scales and with three bounding boxes per grid cell, multiple bounding boxes of the same object might be found. The non-maximum suppression is used to remove these multiple detections and consists in the following algorithm [19]:

Algorithm 1: 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

3.3.3 YOLOv3-Tiny network

The YOLOv3 network comprises a total of nearly 62 million parameters. The author of this detector [18] proposed an alternative network for constrained environments called YOLOv3-Tiny which in turn has a total of approximately 8.8 million parameters. This smaller model presents a mAP (for one IoU of 0.5) of 33.1% in the COCO dataset and runs at a frame rate of 220 fps in the Titan X GPU. Thus, this version is faster but also less accurate than any other detector in Table 3.1. In the scope of this work, this network can be used as a starting point for acceleration before moving to the YOLOv3 network.

As represented in Fig. 3.7, the YOLOv3-Tiny is composed by 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.

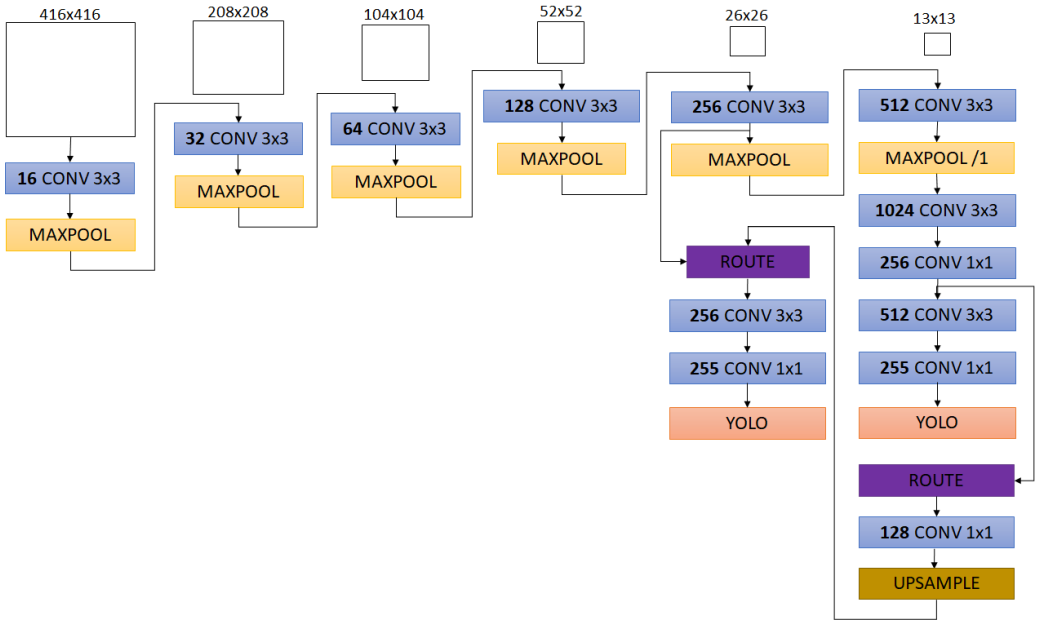

Figure 3.7: YOLOv3-Tiny Network.

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 initially accelerated using the architecture introduced in the next chapter.

Chapter 4

CGRA-based accelerator

In this chapter, the fundamentals about Coarse Grained Reconfigurable Arrays (CGRAs) and why they can be used for accelerating CNNs are briefly explained. The Deep Versat CGRA architecture, which will be used in this work for accelerating the YOLOv3 detector, is described. The chapter ends with a basic example of the implementation of a 2D convolution using a single layer of the Deep Versat core.

4.1 CGRA architecture

A CGRA is a collection of programmable functional units (FUs) and embedded memories interconnected by programmable switches [23]. The interconnections are reconfigurable at run-time, which allows 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 throughput 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 [23] 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 that an application could 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 Fig. 4.1, to limit the number of connections and prevent a frequency drop.

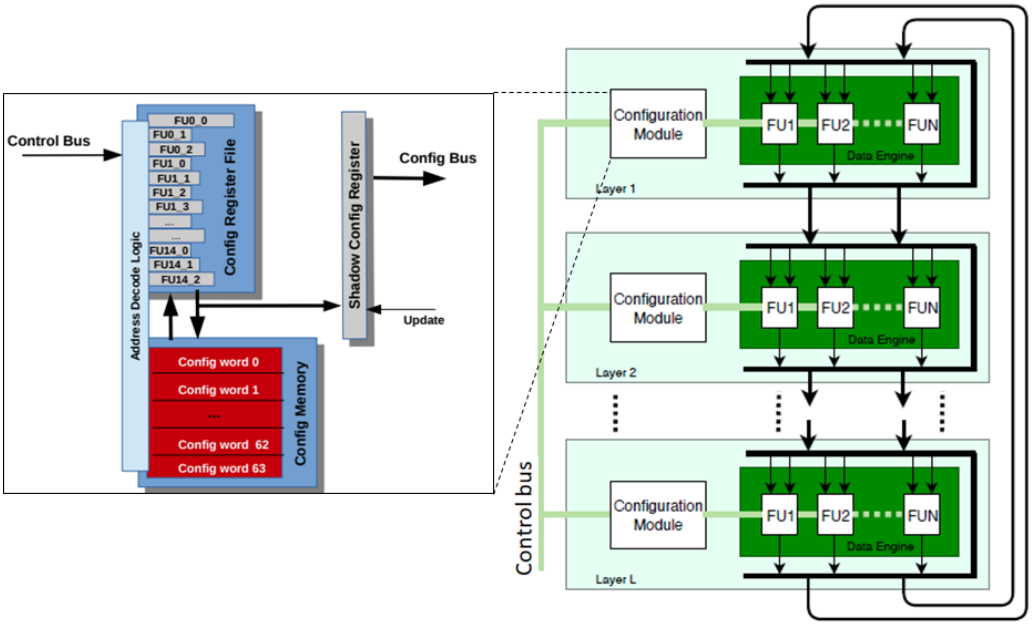

Figure 4.1: Deep Versat architecture (adapted from [14]).

Each Versat has a data engine 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 configuration module is composed by (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. The configuration bus connects the data engine and the configuration module.

Currently, the functional units available include dual-port memories, arithmetic logic units (ALUs), multipliers, multiplier and accumulators (called MulAdd) and barrel shifters. The number of FUs is configured at compile time. Each FU holds different configuration parameters based on their functionality:

Dual-port memory: Besides the 2 input and output ports, each Versat memory has two Address Generation Units (AGUs) which consist in two cascaded counters that allow for the execution of two nested loops in one configuration. 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, for example, a MulAdd. The parameters are described in Table 4.1.

Table 4.1: Parameters of the dual-port memories.

Parameter	Description	Parameter	Description
start	Memory start address	per	N.º of inner loop iterations (period)
iter	N.º of outer loop iterations	duty	N.º of cycles in a period where memory is enabled
incr	Increment of the inner loop	sel	Address of the input FU
delay	N.º of clock cycles before AGU starts	shift	Additional increment at the end of each period
ext	Flag for bypassing the AGU		

ALU / **Multiplier**: These FUs have 3 parameters: the two input operand addresses 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 address 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 has 4 configuration parameters: the two operand addresses, the reset and the type of operation. The reset controls how many accumulations to perform and can be applied with a counter from a memory AGU.

4.2.1 System integration

As previously mentioned, a CGRA is typically accompanied by a CPU. As shown in Fig. 4.2, 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 and manipulate the configuration registers and another one for the data bus, which is used for data transfers. The UART module is another peripheral mainly used for debugging.

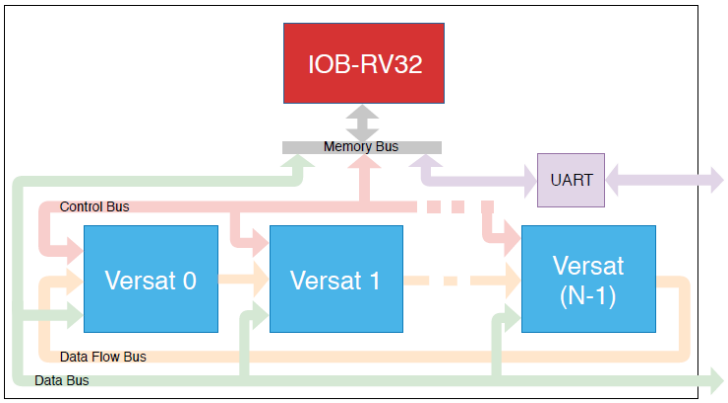

Figure 4.2: Deep Versat system ([14]).

4.2.2 Basic implementation

The RISC-V processor is programmed with a standard GNU toolchain, which contains compilers for C and C++. Therefore, Deep Versat also includes an API in C++ to enhance its configuration process. To show the usage of this API, a simple 2D convolution, using a single Versat layer, between a 5x5 feature map and a 3x3 kernel (without zero-padding) is exemplified in Fig. 4.3.

The feature map and the kernel are stored in different memories (mem0 and mem1). To read a 3x3 block from the feature map, a correct manipulation of the parameters in Table 4.1 is needed. The parameters $iter$, per and $duty$ are set to 3 in order to read one pixel per clock cycle across the 3 columns and 3 lines. The shift must be 2 to read the next row as data is stored in row-major order. For the kernel, 9 consecutive positions from mem1 are read. The second port (B) of this memory is used to generate a counter from 0 to 8 to control the number of accumulations in the MulAdd, whose inputs are connected to both memories. The results are stored in another memory (mem2). Finally, the loops for iterating through the feature map are defined in software to configure the start address of the read memories and to run the Versat at each iteration.

```
//Create versat and clean previous configurations
CVersat vl (VERSAT_1);
vl.clearConf();

//Configure mem0A to read 3x3 block from feature map (base, start, iter, incr, delay, per, duty, sel, shift, ext)
vl.memPort[0].setConf(CONF_MEMA[0], 0, 3, 1, 0, 3, 3, 0, (5-3), 0); vl.memPort[0].writeConf();

//configure mem1A to read kernel (base, start, iter, incr, delay, per, duty, sel)
vl.memPort[2].setConf(CONF_MEMA[1], 0, 9, 1, 0, 1, 1, 0); vl.memPort[2].writeConf();

//configure mem1B to generate counter between 0 and 8 to control MULADD (base, start, iter, incr, delay, per, duty, sel)
vl.memPort[3].setConf(CONF_MEMB[1], 0, 8, 1, MEMP_LAT, 1, 1, sADDR); vl.memPort[3].writeConf();

//configure muladd to perform MAC operation (muladd_base, sela, selo, selb, fns)
vl.muladd[0].setConf(CONF_MULADD[0], sMEMA[0], sMEMB[1], sMEMA[1], MULADD_MUL_LOW_MACC); vl.muladd[0].writeConf();

//configure mem2A to read result of muladd (base, start, iter, incr, delay, per, duty, sel)
vl.memPort[4].setConf(CONF_MEMA[2], 0, 1, 1, 8 + MEMP_LAT + MUL_LAT, 1, 1, sMULADD[0]); vl.memPort[4].writeConf();

//Loop for performing convolution
int i, j;
for (i = 0; i < 2; i++) {
    for (j = 0; j < 2; j++) {

        //configure start value of memories
        vl.memPort[0].setStart(i*5 + j);
        vl.memPort[4].setStart(i*2 + j);

        //run versat
        while (RAM_GET32(VERSAT_1, (ENG_RDY_REG)) == 0);
        versatRun();
    }
}
```

Figure 4.3: Example of 2D convolution with one Versat.

Final remarks

This simple example showed how to accelerate loops using a single Versat layer. By using more layers from the Deep Versat architecture and by applying loop optimization techniques, convolutional layers from complex CNNs such as the YOLOv3 detector can be accelerated. In the next chapter, the proposed methodology and the expected planning of this work are described.

Chapter 5

Proposed methodology and planning

The main goal of this work is to implement and demonstrate a real-time object detector in a FPGA-based development board. Taking into account the study performed in chapter 3, the YOLOv3 detector presents the best trade-off between accuracy and execution time. The Deep Versat system, introduced in chapter 4, alongside the techniques for CNN acceleration in FPGAs studied in chapter 2, will be used to implement the detector.

5.1 Infrastructure

The development board available for this work is the Kintex UltraScale KU040 [24]. To receive images in real-time, the cost-effective OV7670 camera module [25] was chosen as it can be easily connected to the PMOD modules of the board. This camera operates at maximum 30 fps and has a maximum resolution of 640x480 pixels. Given the interfaces available on the board, the HDMI interface was selected to display the resulting images in a monitor. The proposed infrastructure is represented in Fig. 5.1.

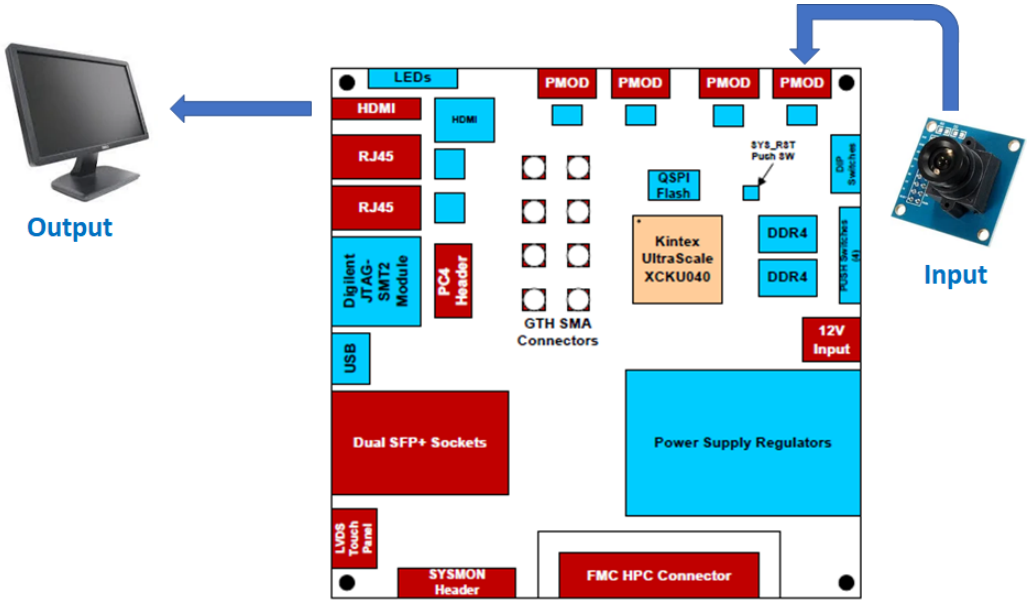

Figure 5.1: Composition of the proposed infrastructure (adapted from [24]).

5.1.1 OV7670 camera module

The pinout of the camera is represented in Table 5.1. The camera has two different communication protocols: one for its configuration and another for data acquisition. The pins SDIOD and SDIOC are used for configuring the camera using a specific protocol similar to I2C. Possible configurations include the data output format (e.g., RGB565, YCbCr 4:2:2), the frame resolution (e.g., 640x480, 320x240) and the frame rate. For this work, the camera will be configured in the RGB565 format with a resolution of 640x480 and a frame rate of 30 fps.

Table 5.1: Parameters of the dual-port memories.

Type	Pin	Description	Pin	Description
Supply	VDD	Power supply	GND	Ground
Input/Output	SDIOD	Control bus data		
Input	SDIOC	Control bus clock	XCLK	System clock
	RESET	Reset (active low)	PWDN	Power down (active high)
Output	VSYNC	Vertical synchronization	HREF	Horizontal synchronization
	PCLK	Pixel clock	D0-D7	Pixel parallel output

Changing the configuration corresponds to updating the value of a specific register in the camera module. When idle, both SDIOC and SDIOD are at logical one. The data transmission starts by driving the SDIOC pin at logical zero. A logical one during data transmission indicates that one bit was sent. The SDIOD pin can only change while SDIOC is at logical zero. After transmitting the address of the register and the value to be updated, the pins are driven at logical one, indicating the end of the transmission.

The pixel data is transmitted in parallel (8 bits). First, a frequency between 10 and 48 MHz must be supplied to the XCLK pin [25]. As shown in Fig. 5.2, the 640 pixels that form one row must be captured while HREF is at logical one and the 480 rows are acquired while VSYNC is at logical zero. The data must be sampled at the rising edge of the PCLK signal. For the RGB565 format, two clock cycles are needed to collect each pixel. The reset and power down signals are kept at logical zero and logical one.

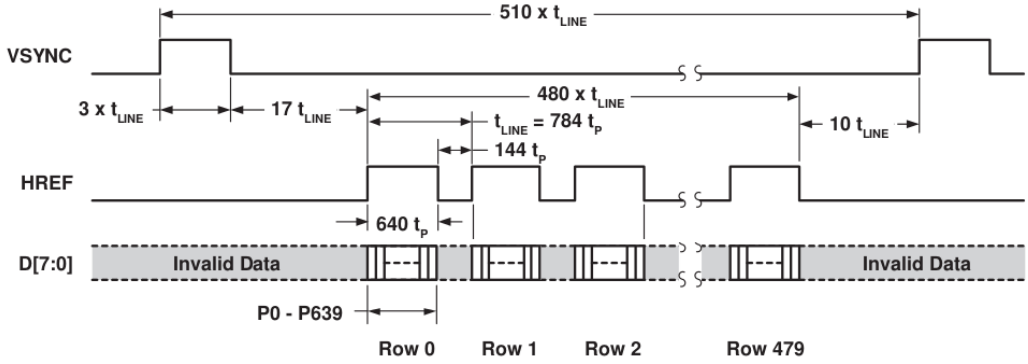

Figure 5.2: Protocol for the camera data acquisition [25].

5.1.2 HDMI module

The board has an ADV7511 HDMI transmitter [26] to drive the interface. Although the ADV7511 module accepts 36 bits per pixel, the board is only connected to 16 pixels (from bit 8 to bit 23) [24]. As a consequence, the format of the data must be YCbCr 4:2:2. This implies a color space conversion from the RGB format of the camera to the YCbCr format of the HDMI interface [27]:

$$\begin{cases} Y = 16 + \frac{65.738}{256} \times R + \frac{129.057}{256} \times G + \frac{25.064}{256} \times B \\ Cb = 128 - \frac{37.945}{256} \times R - \frac{74.494}{256} \times G + \frac{112.439}{256} \times B \\ Cr = 128 + \frac{112.439}{256} \times R - \frac{94.194}{256} \times G - \frac{18.285}{256} \times B \end{cases} \tag{5.1}$$

The video streams have active video periods and blanking periods, as represented in Fig. 5.3 . Only the active video is shown on the display. When transmitting video, the signals HSYNC (horizontal synchronization), VSYNC (vertical synchronization) and DE (data enable) must be generated.

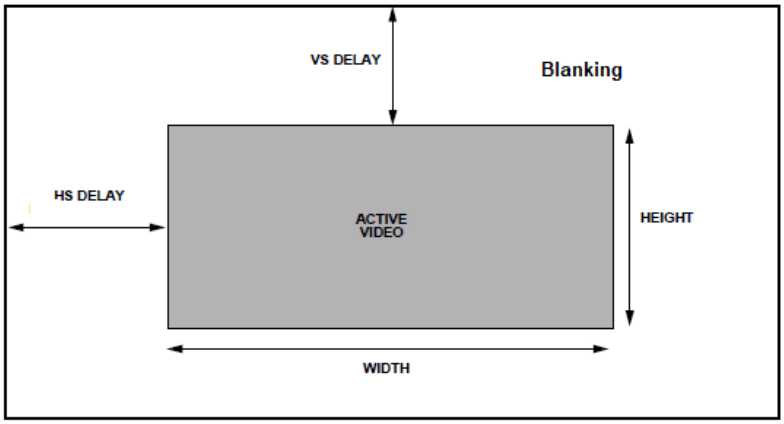

Figure 5.3: Active video (adapted from [25].)

A logic one DE indicates an active pixel (i.e., the visual part of the video signal) and a logic zero DE indicates the blanking period of the video signal. The generation of the DE is represented in Fig. 5.4. The range of values for the synchronization signals depends on the resolution being used. For configurations such as the data format, the I2C protocol is used.

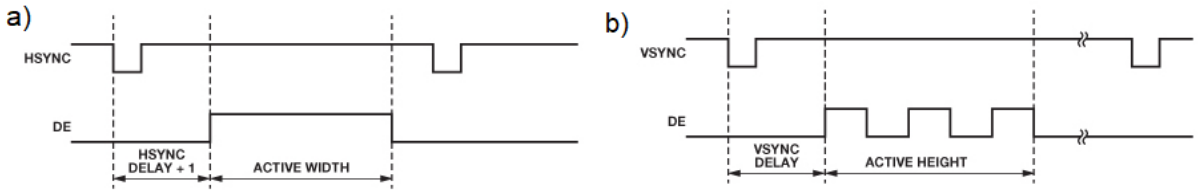

Figure 5.4: a) Horizontal and b) vertical DE generation (adapted from [25]).

The camera and HDMI modules will be developed in Verilog. These modules will be integrated as peripherals in the Deep Versat system as represented in Fig. 5.5.

5.2 Planning

This thesis work is planned for a period of nine months, as shown in the Gantt Chart presented in Fig. 5.6. The first task will consist in implementing a software-only version of the YOLOv3 detector in the RISC-V processor to be used as baseline. As this processor currently does not support floating

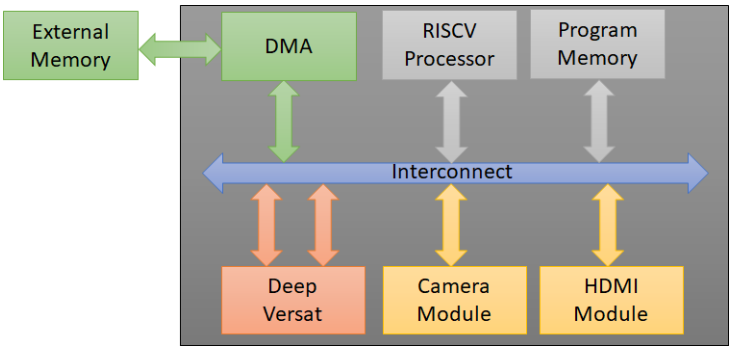

Figure 5.5: Integration of the modules in the Deep Versat system.

point operations, the application will need to be already converted to fixed point arithmetic by applying a pre-defined quantization. The detector will initially be based on the YOLOv3-Tiny network.

For the second task, the convolutional layers will be accelerated using the Deep Versat architecture. Following the approach in previous works, a design space exploration will be conducted in order to find the best parameters for the loop optimization techniques, namely the unroll and tiling factors.

Furthermore, a study will be conducted in order to measure the impact of implementing the other layers and other routines of the detector (e.g., image resize, non-maximum suppression) in the RISC-V processor. If after accelerating the convolutional layers, the bottleneck of the application resides on any of these routines, then hardware-based alternatives will have to be considered. For instance, the logistic operation performed by the yolo layers can be approximated by using pre-defined values stored in LUTs.

The following two tasks will focus on the deployment of the camera and HDMI modules for the infrastructure proposed in this chapter. Finally, the system will be fully integrated and tested. The thesis will be written throughout the implementation of the work.

Task	feb/20	mar/20	apr/20	may/20	jun/20	jul/20	aug/20	sep/20	oct/20
Software baseline									
Convolutional layers acceleration									
Acceleration of other layers and routines									
Interfacing camera module									
Interfacing HDMI module									
Integration, tests and results									
Thesis writing									

Figure 5.6: Planning of the thesis.

Final remarks

The expected results with the development of this work include the infrastructure for receiving images from the OV7670 camera in real-time and displaying the resulting images from the YOLOv3 detector in an HDMI monitor. For the YOLOv3-Tiny network, the expected frame rate should be near the maximum frame rate of the camera (i.e., 30 fps).

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. 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.

[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.

[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.

[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.

[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.

[14] V. Mário. Deep versat: A deep coarse grain reconfigurable array. Master's thesis, Instituto Superior Técnico, November 2019.

[15] X. Long, S. Hu, Y. Hu, Q. Gu, and I. Ishii. An fpga-based ultra-high-speed object detection algorithm with multi-frame information fusion. *Sensors*, 19, 2019. doi: 10.3390/s19173707.

[16] A. Bochem, K. B. Kent, and R. Herpers. Fpga based real-time object detection approach with validation of precision and performance. In *2011 22nd IEEE International Symposium on Rapid System Prototyping*, pages 9–15, May 2011. doi: 10.1109/RSP.2011.5929969.

[17] C. A. Llorente, E. P. Dadios, J. B. Monzon, and W. E. D. Leon. Fpga-based object detection and classification of an image. *International Journal of Engineering & Technology*, 7(4):83–86, 2018.

[18] J. Redmon and A. Farhadi. Yolov3: An incremental improvement, 2018.

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

[20] 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.

[21] 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.

[22] 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.

[23] J. D. Lopes and J. T. Sousa. Versat, a minimal coarse-grain reconfigurable array. In *VECPAR*, 2016.

[24] Avnet. Kintex ultrascale ku040 development board: Hardware user guide, December 2015.

[25] OmniVision. Ov7670 cmos vga (640x480) camerachip implementation guide, September 2005.

[26] AnalogDevices. Low-power hdmi 1.4 compatible transmitter with audio return channel: Programming guide, March 2012.

[27] K. Jack. *Video Demystified: A Handbook for the Digital Engineer, 5th Edition*. Newnes, USA, 5th edition, 2007. ISBN 0750683953.