TÉCNICO LISBOA

FPGA Implementation of a CNN for Oriented Object Detection in Aerial Images

Francisco Miguel Correia Torcato Carrilho

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

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

Examination Committee

Chairperson: Prof. Pedro Filipe Zeferino Aidos Tomás
Supervisor: Prof. Horácio Cláudio De Campos Neto
Member of the Committee: Prof. Rui António Policarpo Duarte

June 2024

Declaration

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

Acknowledgments

I would like to express my deepest gratitude to Prof. Horacio Neto and Prof. Mario Véstias whose provided me with their knowledge, insight, non stop support, constant availability, patience, belief and tremendous amounts of encouragement that helped me make the completion of this project a reality.

I would also like to thank my friends for their encouragement, help, belief, and the moments of joy we shared. I would like to offer my particular thanks to my friend Pedro Ferreira, with whom I shared this journey. Together, we kept pushing each other to get through the finish line.

Last but not least, I would like to thank my family for their unwavering support, not only throughout this journey but also in life. My brother, for his belief, positive outlook, and sound advice. My parents, for their patience, belief in my capabilities, and emotional support that helped me through the roughest stages. And finally, my sister, whose complete support and belief enabled me achieve most of what I have in my life.

Abstract

The objective of this work is to design and implement a hardware/software system for oriented object detection of aerial images. The system is based on a convolutional neural network (CNN) detector and is aimed at processing one aerial image per second using a system-on-chip field- programmable gate array (SoC FPGA).

Object detection in aerial images and videos is an important and challenging computer vision problem with important real-world applications such as emergency rescue, disaster relief, and surveillance. Oriented object detection considers both the position of the object and its rotation angle or orientation which makes detections significantly more accurate but more computationally intensive. Most target applications must be locally computed on unmanned aerial vehicles (UAVs), which requires implementing efficient solutions on edge devices, namely SoC FPGAs.

The hardware/software system implements an optimized version of an oriented object detection model based on the YOLO object detection algorithm. The original YOLO model was optimized and quantized, with both weights and activations represented with a specific 8-bit fixed-point format, to provide an efficient hardware-friendly solution. The system is composed by a dedicated hardware accelerator, which accelerates the inference of the main layers of the CNN model by executing 256 multiply-accumulate operations in parallel, and by a software processor that executes the less computing intensive functions. The final hardware/software system, implemented in a Zynq SoC FPGA, executes the inference of the R-YOLOv4 with a frame rate close to 1 FPS and a power consumption of only 7.3 W.

Keywords

Object Detection, Quantization, Convolutional Neural Network, FPGA, Hardware/Software Co-design, Hardware Acceleration.

Resumo

O objetivo deste trabalho é projetar e implementar um sistema hardware/software para deteção de objetos em imagens aéreas. O detetor utiliza uma rede neuronal convolucional (CNN) e visa processar uma imagem aérea por segundo num sistema em circuito integrado programável (SoC FPGA).

A deteção de objetos em imagens e vídeos aéreos é um problema importante com muitas aplicações relevantes. Por considerar simultaneamente a posição do objeto e o seu ângulo de rotação ou orientação, a deteção orientada de objetos permite aumentar significativamente a precisão das deteções, mas é computacionalmente mais exigente. A maioria das aplicações alvo requer computação local em veículos aéreos não tripulados (UAVs), o que obriga a implementar soluções eficientes em dispositivos de baixo consumo de energia, nomeadamente em SoC FPGAs.

O sistema desenvolvido implementa uma versão otimizada de um modelo de deteção orientada de objetos baseado no algoritmo de deteção de objetos YOLO. O modelo YOLO original foi otimizado e quantizado, com pesos e ativações representados num formato de vírgula fixa de 8 bits, para maximizar o desempenho da sua execução em hardware dedicado. O sistema é composto pelo acelerador hardware, que acelera a inferência das camadas principais da CNN executando 256 operações de multiplicação-acumulação em paralelo, e por um processador genérico que executa em software as funções menos exigentes. O sistema final, implementado num SoC FPGA Zynq, executa a inferência do R-YOLOv4 com um ritmo de processamento de imagens próximo de 1 FPS e um consumo de energia de apenas 7,3 W.

Keywords

Detecção de Objetos, Quantização, Rede Neural Convolucional, FPGA, Co-projeto Hardware/Software, Acelerador em Hardware.

Contents

List of Tables

List of Figures

Acronyms

APU	Application Processing Unit
AXI	Advanced eXtensible interface
BRAM	Block random access memory
CNN	Convolutional Neural Network
CPU	Central processing unit
CSP	Cross-Stage-Partial-connections
DMA	Direct memory access
DNN	Deep Neural Network
ELU	Exponential linear unit
FM	Feature map
FPGA	Field-programmable gate array
GPU	Graphics processing unit
HBB	Horizontal bounding box
HLS	High level synthesis
IoU	Intersection over Union
IP	Intellectual Property
MAC	Multiply–accumulate
MPSoc	MultiProcessor System-On-Chip
NMS	Non-Maximum Surpression
OBB	Oriented bounding box
ODAI	Object detection in aerial images
ODNI	Object detection in natural images
PANet	Path Aggregation Network
PE	Processing element
PL	Programmable Logic
PS	Processing System
R-CNN	Regional Convolutional Neural Network
ReLU	Rectified linear unit
RGB	Red-Green-Blue
ROI	Region of interest

RTL Register-transferlevel

SoC System-on-chip

SPP Spatial Pyramid Pooling

SSD Single Shot Detector

SVM Support Vector Machine

tanh Hyperbolic tangent

UAV Unmanned Aerial Vehicle

URAM Ultra random access memory

YOLO You Only Look Once

Chapter 1

Introduction

The goal of this work is to develop a deep learning-based object detection accelerator using a System-on-chip (SoC) Field-programmable gate array (FPGA). The system will perform real-time object detection in aerial images captured by edge devices such as Unmanned Aerial Vehicle (UAV). This section describes the motivation and main objectives of the project, as well as the report outline.

1.1 Motivation

Object detection on images or videos is an important and challenging computer vision problem with a wide range of applications, like surveillance, medical healthcare, and autonomous driving. In aerial images, object detection has become more important with the rise of the UAV and has many real-world applications such as emergency rescue, disaster relief, urban planning, surveillance, weather prediction, etc. [1].

Advances in deep learning technology have led to the creation of Convolutional Neural Network (CNN) based models for object detection whose accuracy surpasses the classical computer vision solutions [2] and are being used for research in aerial sensing using UAVs [3].

State-of-the-art object detectors are usually run on Central processing unit (CPU) or Graphics processing unit (GPU) that, for edge computing, namely in UAVs, are unsuitable solutions due to their high energy consumption and monetary cost. Therefore, for this kind of application, the use of FPGAs can be an efficient alternative even if, due to the limited resources, a compromise between performance and accuracy would need to be found in order for one of these systems to be deployed in a FPGA.

1.2 Objectives

The objective of this work is the implementation of a state-of-the-art CNN-based object detector in an SoC FPGA able to perform object detection in aerial images with a throughput of one frame per second.

State-of-the-art object detection models will be studied, in particular models for oriented object detection in aerial images. These models are based on CNNs and therefore it is important to study their

structures and how they can be implemented in a SoC FPGA. Afterward, the selection, optimization, training, and quantization of the selected model is done. The training is performed on a machine with a high-performance GPU to mitigate the time spent on this time-consuming task. Furthermore, the hardware accelerator is designed in hardware and validated. The hardware accelerator is then integrated in a SoC architecture and mapped in a SoC FPGA. The embedded software necessary to for the network deployment will be developed and then, the HW/SW system is validated and tested.

1.3 Report Outline

This report has the following structure:

- Chapter 2 introduces deep learning concepts, in particular CNNs, reviews the state-of-the-art object detection solutions based on deep-learning, strategies, and techniques that help to implement CNN on an FPGA and CNN accelerators;
- Chapter 3 describes the design and optimizations done of the object detection model selected for this project, the dataset used for training and testing, details about the software tools used and the development workflow. The results from the model optimizations and model quantizations are detailed;
- Chapter 4 presents the design of the hardware accelerator that will be integrated in the FPGA, along side its implementation results;
- Chapter 5 the describes the integration of the hardware accelerator into the embedded system. The entire HW/SW architecture is detailed and the experimental results such resource utilization and performance results are presented;
- Chapter 6 concludes the work and proposes future development to further improve the project.

Chapter 2

Background and State of the Art

This chapter introduces the concept of convolutional neural networks by explaining their functionalities and applications, as well as their role in state-of-the-art object detection models that are also presented and compared. The implementation of FPGA accelerators is also discussed by presenting parallelization opportunities in convolutional neural networks as well as possible accelerator architectures.

2.1 Convolutional Neural Networks

A convolutional neural network (CNN) is one of the many types of Deep Neural Network (DNN). Similarly to other types of neural networks, a CNN consists of multiple layers of interconnected nodes. However, the CNN is specifically designed to process grid-like structured data making it especially good when used for image analysis and pixel data processing.

Even though CNNs are applicable in a variety of tasks, their characteristics make them the model of choice for problems involving image processing since images are usually data represented as a grid-like arrangement of pixels, whose values represent their brightness. Some of the most popular applications of CNNs are image classification, object recognition, and image segmentation.

A CNN is composed of three key layers: convolutional, pooling, and fully connected. Apart from these, there are some other layers that can be added to help improve the model and the training in various ways such as batch normalization and upsampling layers. All of these layers can and are usually followed by an activation function.

2.1.1 Convolution Layers

The convolutional layer is the main building block of a CNN, it applies the convolution operation on the input and it is where most of the computations are performed. A convolution is a mathematical operation that when performed on two functions, creates a third one that expresses how the shape of one is modified by the other. This operation involves utilizing a smaller matrix of learnable parameters, known as kernel and is applied to a smaller area of the input known as the receptive field. The kernel

slides through the width and height of the input and the dot product between the input pixels and the kernel is calculated and fed to an output array. Once the process is complete and the kernel swept the entirety of the input map, an output map is generated. Both input and output maps are generically designated feature map (FM). The process just described is known as a 2D convolution.

In the case of a 3D convolution the input is no longer a simple 2D matrix but instead, a set of 2D maps, for example, an Red-Green-Blue (RGB) image that has 3 channels and so is comprised of 3 matrices of pixels, one for each channel. Each of these maps is then passed by a group of filters which are basically 3D tensors composed of one kernel for each of the channels of the image. Each kernel of the filter will perform a 2D convolution with each channel of the input and then all those maps are added together, along with an optional bias parameter, to produce the output map. There are as many output maps as there are filters involved in the convolution. An example of a 3D convolution is represented in Figure 2.1.

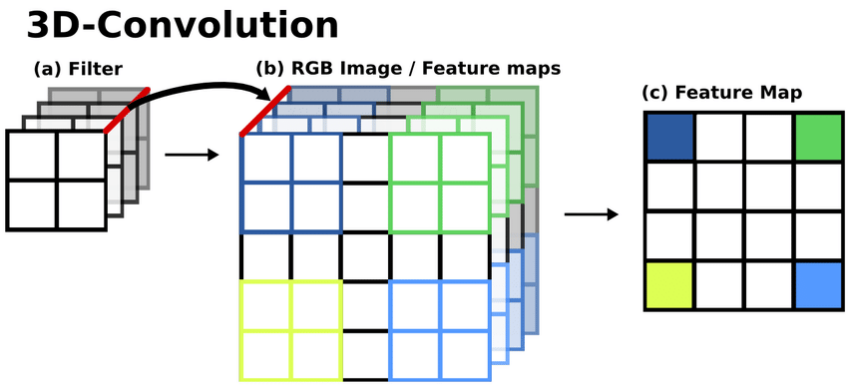

Figure 2.1: Example of a 3D convolution from [4].

The dimensions of the resulting FM depend on the dimensions of the filters and input. The depth of the FM is equal to the number of filters applied to the input. The width and height are calculated based on (2.1), where $in_{h,w}$ is the input's height/weight, $Out_{h,w}$ is the output's height/width, $k_{h,w}$, is the kernel height/width, s is the kernel stride and p represents the padding added (number of layers of 0 added to the edges of the input).

$$Out_{h,w} = \frac{in_{h,w} - k_{h,w} + 2 * p}{s} + 1 \tag{2.1}$$

Each operation performed in this layer can be described as a multiply–accumulate (MAC) operation between the values of the kernels and the input pixels. For a generic 3D convolution the number of MACs performed is given by

$$Total_{MAC} = Out_w \times Out_h \times N_{channels} \times k_h \times k_w \times N_{filters}, \tag{2.2}$$

where Out_w and Out_h are the output's width and height respectively, k_w and k_h are the kernel's width and height respectively, $N_{filters}$ is the number of filters and $N_{channels}$ is the number of channels.

2.1.2 Pooling Layers

A pooling layer reduces the spatial dimensions of the input data, reducing the number of parameters by applying a pooling operation to it. A pooling operation, much like the convolution operation, sweeps the input with a kernel and a given stride. However, the filter of the pooling operation has no weights and instead, produces the output FM by performing a statistical summary of all neighboring pixels within the receptive field. There are multiple pooling functions however, the most commonly used is max-pooling which stores in the output the maximum value within the evaluated vicinity. An example of max-pooling is represented in figure 2.2.

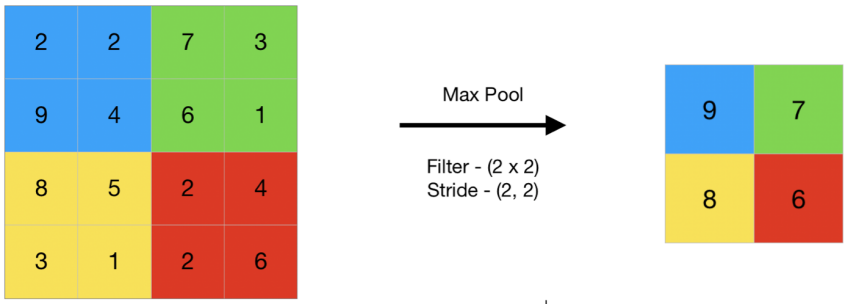

Figure 2.2: Example of a max-pooling layer with a window of 2×2.

The shape of the output FM of a pooling layer is determined by (2.3), where $in_{h,w}$ is the input's height/weight, $Out_{h,w}$ is the output's height/width, $k_{h,w}$, is the kernel height/width and s is the kernel stride.

$$Out_{h,w} = \frac{in_{h,w} - k_{h,w}}{s} + 1 \tag{2.3}$$

2.1.3 Fully Connected Layers

As the name indicates, in a fully connected layer every node of the input layer is connected to every output node and has a weight associated. Each output node is the result of the sum of the multiplication of every input and its respective weight and a bias term. The value y_k of an output neuron k of a fully connected layer is given by

$$y_k = \sum_{i=0}^{N_{input}-1} w_{ki} * x_i + b_k, \tag{2.4}$$

where $x_k i$ and $w_k i$ are vectors with all the inputs values and weights associated with k and b_k is the optional bias term.

2.1.4 Activation Functions

Activation functions are nonlinear functions that are applied to the input FM. The most commonly used activation functions are the sigmoid, the hyperbolic tangent (tanh), the rectified linear unit (ReLU), the leaky ReLU, the exponential linear unit (ELU), and the softmax. The aforementioned functions are represented in figure 2.3.

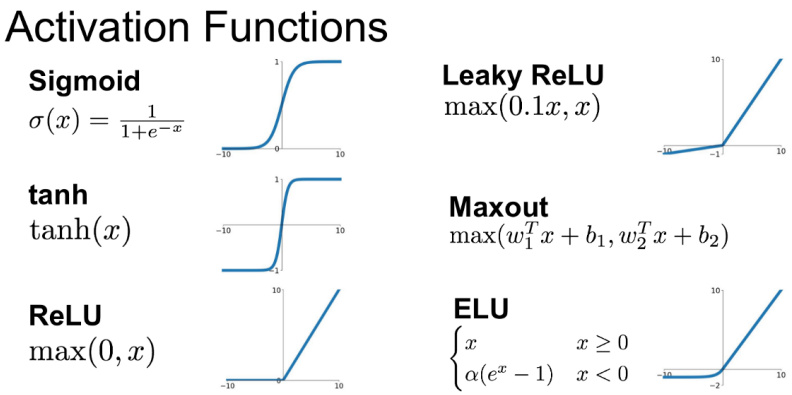

Figure 2.3: Examples of activation functions.

2.1.5 Shortcut Layers

This layer is used to mitigate the vanishing gradient problem, which happens in deeper DNN models, where the gradient gets smaller and smaller making it difficult for the network to learn. Shortcut connections skip one or more layers and then add their output to the output of a layer further ahead. This concept was proposed in [5] and a visual example is represented in Figure 2.4.

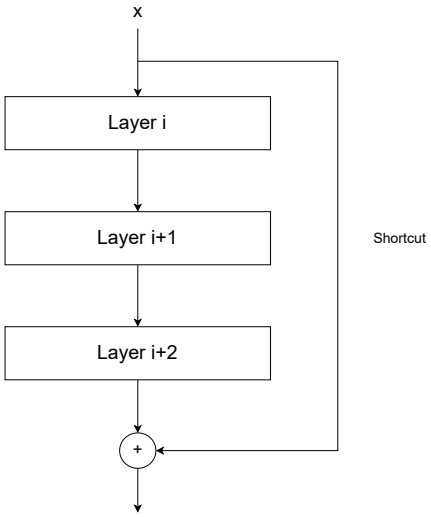

Figure 2.4: Example of a shortcut layer in addition.

The example in the figure includes a series of three convolutional layers in the backbone and a shortcut connect. The output of the last layer is added to the output of the last convolutional layer.

2.1.6 Routing Layers

Routing layers recover the output from one or more previous layers, by concatenating them and bringing them forward in the network while avoiding all the operations in between, bringing forward the finer-grained features, from earlier stages of the network.

2.1.7 Batch Normalization

Batch normalization is an optional layer that aims to normalize the inputs of each layer inputs to reduce internal covariate shift. This is achieved by initially dividing the data into mini-batches and then normalizing the values based on its average (μ) and standard deviation (σ), whose values are $\mu = 0$ and $\sigma = 1$. After training, the learned parameters ϵ and β are used to shift and scale the input values. This process was proposed by [6] and helps control each layer's input distribution, reducing internal covariate shift which in turn not only speeds up the training process but also improves accuracy. Each normalized output, z, for a given value x is given by

$$z = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta, \tag{2.5}$$

where μ is the average, σ is the standard deviation, β, and γ are trainable parameters, and $\epsilon$is a small constant that avoids numerical errors for denominators that are too close to zero.

2.1.8 Upsampling Layers

The upsampling layer increases the size of the FM to be fed to the next layer. A possible upsampling strategy is replicating each pixel into a 2 by 2 square, doubling the height and width of the input FM. Upsampling helps with the recovery of resolution due to all the downsampling done in convolutional and pooling layers.

2.2 CNN models for Object Detection

Object detection is a computer vision problem that aims to determine the location and identity of a given object in an image or video. This problem can be divided into three stages: region selection, feature extraction, and object classification [7]. There are a variety of deep learning-based solutions for this problem, including one-stage and two-stage detectors accompanied by various neural network architectures.

All object detectors take an image as an input and extract features from it using a convolutional neural network known as the backbone. The backbone is a CNN that on its own could be used to perform image classification. As a support for the backbone, recent deep learning-based object detectors introduced the neck section where features from different stages of the backbone are mixed, in order to help prepare the object detection step. The object detector's main function of classifying and localizing objects is performed at the head.

These models tend to detect objects by placing bounding boxes around them and classifying them within the correct category. These bounding boxes can be simple horizontal bounding boxes (HBBs) or oriented bounding boxes (OBBs). Both are valid representations, that are dependent on the applications. However, for aerial image object detection, OBB is more appropriate, since it allows the observer to more accurately distinguish objects in instances that are closely packed. A figurative example can be observed

in Figure 2.5.

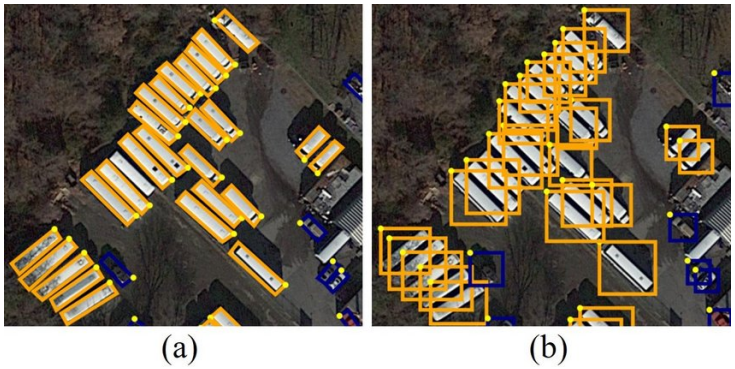

Figure 2.5: OBB(a) and HBB(b) representation comparison presented in [8].

As can be seen, the left image uses oriented boxes and so objects are easily identified In the image on the right, horizontal bounding boxes are used. In this case, boxes overlap and it is difficult to identify objects.

Object detection models are divided into two major groups: the more accurate two-stage detectors such as the R-CNN family of detectors [9–11]; and the faster but less precise one-stage detectors such as the YOLO [12–15], the SSD [16] and the RetinaNet [17]. These models' use cases depend on the application. Typically, two-stage object detectors are used when accuracy is the most important aspect. One-stage object detectors are used in embedded devices where some accuracy must be traded-off for performance.

Two-stage detectors divide the object detection problem into two steps. In the first step, a model for feature extraction and region proposals is used, while in the second step, each region previously proposed is fed to a classifier architecture to determine if and what objects are present in said region. These models tend to be more precise but are also slower and more computationally heavy than one-stage detectors. An example of this approach is the Regional Convolutional Neural Network (R-CNN) family of object detectors.

One-stage object detectors are designed to solve the same problem but in a single forward pass through a single network that is used for all stages of the object classification problem. This means using a single network that produces a set of bounding boxes and classifications for objects present in a given input image. These models, even if less precise than two-stage detectors, have faster inference times. Some examples of these models are the Single Shot Detector (SSD) and You Only Look Once (YOLO) models.

2.2.1 R-CNN: Regional Convolutional Neural Network

The R-CNN was initially proposed in 2014 and started by solving each stage of the object detection problem independently [9]. This architecture starts by extracting region proposals from the input image and then feeding these regions to a CNN model for feature extraction. Finally, a classification model known as Support Vector Machine (SVM) operates as a fully connected layer and is used to classify each

region. The last two steps can be seen as a normal CNN that operates on cropped region proposals of the original input. An illustrative representation of this model is depicted in Figure 2.6.

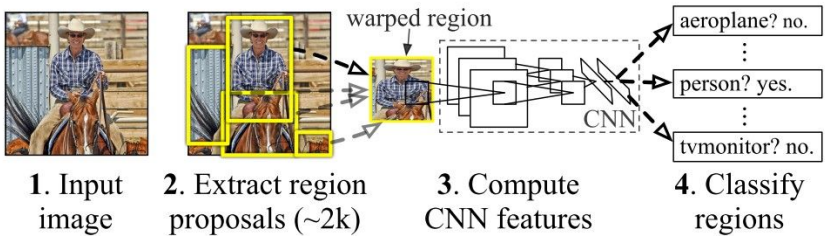

Figure 2.6: R-CNN architecture proposed in [9].

After the proposal of this model, several other variants were created, to improve the initial approach. These improved models include the Fast R-CNN [10] and the Faster R-CNN [11]. The Fast R-CNN introduced the Region of interest (ROI) pooling layer into the original architecture, which allows the use of a single feature map for the whole input image. The model receives an input map and a set of RoIs, the input is then sent to the CNN for feature extraction. The resulting FM is then divided into regions and reshaped into feature vectors of a fixed size to be fed to the fully-connected layer and classified. So, instead of having to pass each region through the CNN, the convolution operation is only done once for the entire input making the process faster. The Faster R-CNN is an even further improvement on the previous models, using a CNN to compute region proposals, ditching the slow selective search algorithms proposed by its two previous models.

2.2.2 SSD: Single Shot Detector

The SSD [16] is an object detection model designed to operate in real-time, composed of two main parts, the backbone, and the SSD head. A visual representation of the SSD architecture is present in 2.7.

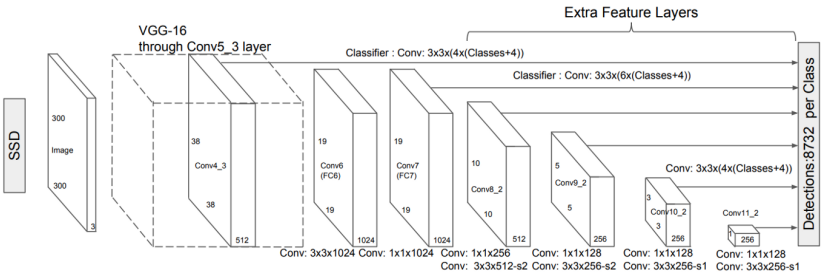

Figure 2.7: SSD architecture from [16].

The backbone as mentioned before is where high-level features are extracted from the input image and, is based on a standard image classification network with the final fully connected layers to convolutional layers, which in this case is the VGG-16 network but other models such as a variation of the ResNet could also work. The head adds a few auxiliary convolutional layers that gradually decrease the resolution of the FMs produced which will allow for multi-scale object detection. The SSD then divides

the FMs into grid maps and generates default bounding boxes for each grid cell of the FM. It then slides a small filter over each FM predicting a number of bounding boxes for each grid cell, relative to the default box. Moreover, all the confidence scores for all object categories are produced simultaneously with the generation of the predicted boxes, hence the name single shot detector.

2.2.3 YOLO: You Only Look Once

YOLO [12–15], much like the SSD, is a real-time object detection algorithm that uses a single CNN to localize and classify all the objects present in an image. This method divides the input image into a grid, where each grid cell detects objects if said object's center is present within the grid cell in question. This means that each grid cell is responsible for the prediction of conditional probabilities for each available class for a given object (C). Furthermore, each grid cell predicts several bounding boxes and confidence scores that —represent how confident the model is that an object is present in a given box, also known as the objectness score. Bounding boxes consist of 5 predictions, the coordinates of the center of the box (b_x, b_y), the width and height of the box (b_w, b_h), and the objectness score. The center coordinates are relative to the grid cell dimensions whereas the width and height of the box are relative to the whole image.

The predictions of the model, represented in Figure 2.8, for an input divided into a $G \times G$ grid, with B bounding boxes per grid cell, are encoded in a tensor of shape

$$G \times G \times (B * 5 + C). \tag{2.6}$$

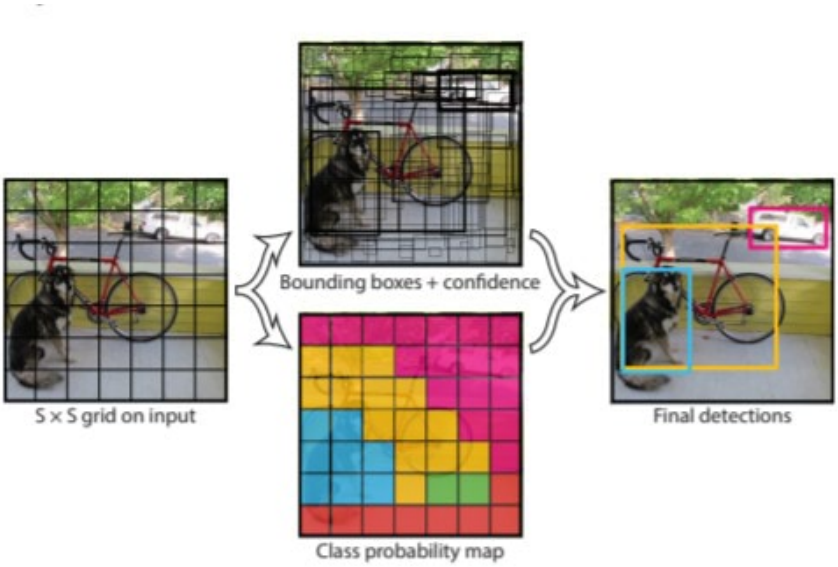

Figure 2.8: YOLO model proposed by [12].

In an image, a single object can have multiple candidate grid boxes which, when irrelevant need to be discarded, meaning that needs to be a metric to determine which boxes are accurately localizing an object. The iIntersection over Union (IoU) is a metric used to determine the accuracy of an object

detector by comparing the ground truth bounding box of an object to the predicted ones and, can have values between 0 and 1. The user defines an IoU threshold that when exceeded by a given bounding box means that the bounding box is a good prediction. Naturally, the higher the threshold the more accurate the model will be. A representation of how the IoU is calculated can be found in Figure 2.9.

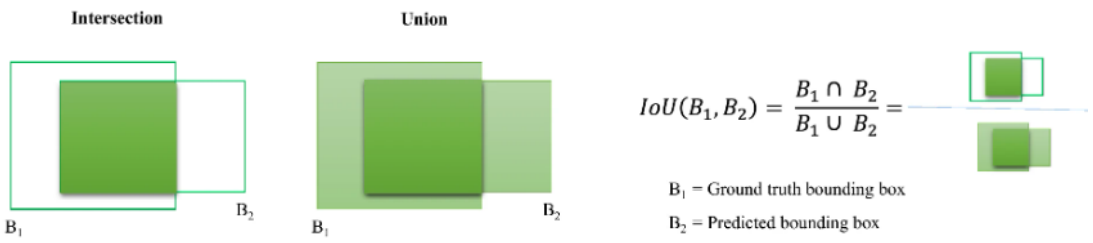

Figure 2.9: Diagram of explaining IoU calculation from [18].

As it can be seen in figure 2.9, the IoU represents the proportion between the area of the intersection of two boxes and the area of their union. The higher the value of IoU, the more overlap there is between the two boxes which means that, in the context of object detection, higher values of IoU represent better predictions.

Now that a metric has been established to test the correctness of a bounding box there is still the problem of multiple predictions exceeding the threshold which will lead to multiple bounding boxes per object. To solve this problem a technique known as Non-Maximum Surpression (NMS) is used to prune bounding boxes produced by suppressing (eliminating) the bounding boxes with too much overlap. This technique consists of selecting the bounding boxes with the highest IoU scores with the ground truth and then suppressing the ones with lower scores that overlap with it and repeating this process until all predicted bounding boxes are considered. The overlap between the boxes is determined using the IoU of the selected box and all other predicted bounding boxes, meaning ones over a certain IoU value with the selected bounding box are considered to be duplicated and eliminated. By the end of this process, all that remains are the bounding boxes with the highest probability score of containing an object and its classification, meaning the object detection task is concluded.

2.2.4 YOLOV4 and R-YOLOv4

The YOLO model has had many versions and one of its adaptations is the YOLOv4, which introduces some new key features and improvements over the previous versions. This version not only introduces a new backbone but also uses a variety of techniques that either improve performance without introducing further inference cost or accuracy but have an impact on inference time.

The backbone used for the YOLOv4 is the CSPDarknet53 [19] which, when compared to other models, is probed to be more suitable for the object detection problem. This network consists of convolutional layers but and Cross-Stage-Partial-connections (CSP) blocks that help combat the vanishing gradient problem very prominent in deep neural networks with considerable depth. This is accomplished by the use of routing and shortcut layers, as mentioned before.

YOLOv4's neck has two major blocks: Spatial Pyramid Pooling (SPP) and the Path Aggregation Network (PANet). Both blocks are used for feature extraction which helps to isolate the most relevant features extracted by the backbone and helps the detection process.

The main part of the algorithm is located at the head which, in this architecture, is the same used in YOLOv3 [14]. The YOLOv3 (and consequently the YOLOv4) makes predictions on three different stages of the convolution network to be able to detect objects of different sizes, in a similar fashion to the SSD. The YOLOv3 also uses predefined anchor boxes and predicts the offsets of said boxes instead of directly predicting the width and height of the bounding boxes. Thus calculations for bounding box predictions, according to [14] are given by

$$\begin{cases} b_x = \sigma(t_x) + c_x \\ b_y = \sigma(t_y) + c_y \\ b_w = p_w e^{t_w} \\ b_h = p_h e^{t_h} \end{cases}, \tag{2.7}$$

where t_x, t_y, t_w, t_h are the network's predictions for the bounding box coordinates, p_w and p_h are the anchor's predefined dimensions and c_x and c_y are the cell offset from the top left corner of the image.

Furthermore, YOLOv4 introduces a few data augmentation techniques, regularization methods, and even a few different activation functions (such as the mish activation) to further improve performance, which is all part of the bag of freebies and bag of specials proposed.

The R-YOLOv4 is a YOLOv4 modification proposed in [20] that allows the model to make accurate predictions with OBB instead of HBB. This is achieved by adding the rotation angle, θ, as an additional attribute to the bounding boxes and adding more anchor boxes at different rotating angles. The number of additional anchor boxes added is the result of multiplying the original three anchors per grid cell, per scale by the number of rotation angles considered. The angle of the predicted bounding box is given by

$$b_\theta = p_\theta + t_\theta, \tag{2.8}$$

where t_θ is the network prediction for the bounding box angle and p_θ is the predefined anchor's angle. Furthermore, the original bounding box loss had to be combined with the smooth-L1-IoU loss function proposed by [21].

2.2.5 Performance Metrics and Model Comparisons

To evaluate an object detection algorithm several metrics are used to characterize the capacity of said model to accurately and correctly localize and classify objects. The most commonly used metrics are Average Precision (AP) and mean Average Precision (mAP). To obtain these metrics there is a need to calculate the precision and recall of the model. The precision is used to measure the percentage of correct positive predictions among all positive predictions made, while the recall measures only the percentage of correct positive predictions among all actual positive cases. These metrics are calculated using (2.9) and (2.10) respectively where, t_p are the true positive predictions, f_p are false positive

predictions, t_n are correct negative predictions and f_n are false negative predictions.

$$Precision = \frac{t_p}{t_p + f_p} \tag{2.9}$$

$$Recall = \frac{t_p}{t_p + f_n} \tag{2.10}$$

The AP is then obtained by sorting the classifications produced by the model and then plotting the precision-recall curve and then calculating the area below the curve, meaning that AP will be high when both precision and recall are also high and low if either of them is low. Since both the precision and recall can only assume values between 0 and 1 the AP is given by

$$AP = \int_0^1 p(r)dr. \tag{2.11}$$

The mAP is the mean of the AP value of each class under consideration and is given by

$$mAP = \frac{1}{C}\sum_{k=0}^{C} AP_k, \tag{2.12}$$

where C is the total number of classes and AP_k is the AP for class k. Both mAP and AP can be calculated for different IoU threshold values and the different object scales considered.

With the proper metrics in hand to evaluate model correctness, a performance comparison between the different models mentioned can now be made, not only taking into account the precision of each model but also their inference times.

Table 2.1, shows the mAP_{50} (IoU threshold equal to 0.5) scores and inference times of known models referred to previously, on the Common Objects in Context (COCO) dataset [22], using the M40 and Titan X GPUs which are identical in performance.

Table 2.1: Inference times and mAP_{50} results on COCO dataset adapted from [14] and [17]

Model	mAP_{50}(%)	Inference Time (ms)
YOLOv3-320	51.5	22
YOLOv3-416	55.3	29
YOLOv3-608	57.9	51
SSD321	45.4	61
DSDSD321	46.1	85
R-FCN	51.9	85
SSD513	50.4	125
DSSD513	53.3	156
FPN FRCN	59.1	172
RetinaNet-50-500	50.9	73
RetinaNet-101-500	53.1	90
RetinaNet-101-800	57.5	198

These results lead to the conclusion that while having similar performance, the YOLOv3 exhibits a much superior speed when compared to the state-of-the-art models. Furthermore, when observing Table 2.2, adapted from [15], it is observable that the YOLOv4 improves on the YOLOv3 results for the same dataset. These results were obtained on the M40 GPU.

Table 2.2: AP_{50} and FPS results on COCO dataset adapted from [15]

Model	AP_{50}(%)	FPS
YOLOv3-320	51.5	45
YOLOv3-416	55.3	35
YOLOv3-608	57.9	20
YOLOv3-SPP-320	60.6	20
YOLOv4-320	62.8	38
YOLOv4-416	64.9	31
YOLOv4-608	65.7	23

Even so, these results are on the COCO dataset which is used for Object detection in natural images (ODNI). To better understand the performance of these models in aerial images table 2.3 shows the results of some of the models mentioned on the UCAS-AOD [23] dataset which is used for Object detection in aerial images (ODAI).

Table 2.3: mAP results on UCAS-AOD dataset adapted from [24] and [20]

Model	mAP(%)
Faster-RCNN	84.26
CNN-SOSF	86.52
YOLOv2	44.63
YOLOv3	91.09
CNN-AOOF	92.42
R-YOLOv4	95.05

As can be seen from the table, R-YOLOv4 is the most accurate model for oriented object detection among the compared models.

2.3 CNN implementations on FPGA

CNNs, although incredibly useful, are also computationally complex meaning they can take a long time to run and produce results. This happens because CNN architectures while reducing the number of parameters of the network, increase the number of operations performed. Most of these operations happen on the convolutional layer, as it can be seen in (2.2) and it is on these layers that 90% of inference time is spent[25]. Nevertheless, while computationally expensive, the convolution operation

is highly parallelizable. This is why CNNs are usually run on GPUs or implemented in reconfigurable devices such as CGRAs or FPGAs.

Hardware programmable FPGAs allow for the exploration of alternative accelarator architectures, and provide support for CNN acceleration by exploiting the different levels of parallelization associated with CNN computations. This way it is possible to optimize the hardware design to maximize performance. The pseudo-code for generic 3D convolution is represented in algorithm 1.

Algorithm 1 3D Convolution Loop

```
1: for i < N_Filters do                                   ▷ iterate through the filters
2:     for j < O_w do                                     ▷ iterate through output FM width
3:         for l < O_h do                                 ▷ iterate through output FM height
4:             OFM[i,j,l] = bias[i]                       ▷ Add bias term
5:             for c < N_channels do                      ▷ iterate through the channels
6:                 for k < k_w do                         ▷ iterate through kernel width
7:                     for n < k_h do                     ▷ iterate through kernel height
8:                         OFM[i,j,l] += IFM[j,l,c] * filter[i,k,n,c]   ▷ Perform MAC
```

By analyzing the pseudo-code in 1 it is noticeable that many operations within the loops do not have data dependencies meaning the loops could be parallelized. These parallelization opportunities are:

1. Operations within a single layer are independent meaning the last two for loops are parallelizable.(Intra-Convolution)

2. Each channel has its own FM and kernel meaning they can be processed in parallel (Inter-Convolution)

3. Multiple pixels from a single output FM can be processed in parallel (Intra-FM)

4. Different output channels can be computed simultaneously (Inter-FM)

These parallelization levels show that most operations performed on each of the loops are independent, allowing for loop unrolling. Loop unrolling is a technique that allows for multiple iterations of a loop to be performed in parallel significantly reducing the execution time, given a factor determined by how many iterations can be done simultaneously.

Another challenge of FPGA CNN acceleration is the lack of on-chip memory to store the entirety of the input and output FMs along with the kernels, meaning the data needs to be stored in an external memory, whose accesses create a significant bottleneck. One possible solution for this problem is the use of loop tilling where the kernels and input FMs are divided into blocks that can be stored in on-chip buffers. This technique along with a datapath that maximizes data reuse significantly reduces the overhead of accessing the external memory.

Furthermore, the reduction of precision of the fixed-point representation of the operands can also be used not only to reduce data transfer overhead but also to maximize the number of operations possible for the same resources. This technique known as quantization consists of reducing the number of bits of the operands which allows for more data to be stored in the on-chip memory, faster data transfers since more data can be transferred for the same bandwidth and also reduce the hardware complexity of

computations allowing for more MAC operations to be executed for the same number of DSPs. Nonetheless, this technique needs to be used carefully since reducing the number of bits results in the loss of precision of the model meaning a sweet spot for the trade-off of precision and performance needs to be found.

Quantization can be applied post-training or during training. In the first case, the network is trained and then the produced weights are converted from the original floating-point format into the desired fixed-point configuration. The second case results from converting the weights and activations during the forward pass of the training process resulting in already quantified values. While taking longer to train, the second choice leads to better results, since in this case the network is trained to take into account the quantization error.

Weight pruning can also be applied, by eliminating weights that do not surpass a certain threshold, reducing the overall number of MACs performed by the network. This technique shares some of the same risks of quantizations and also requires fine-tuning in order to avoid loss of precision. A representation of a generic architecture of a CNN accelerator is present in Figure 2.10.

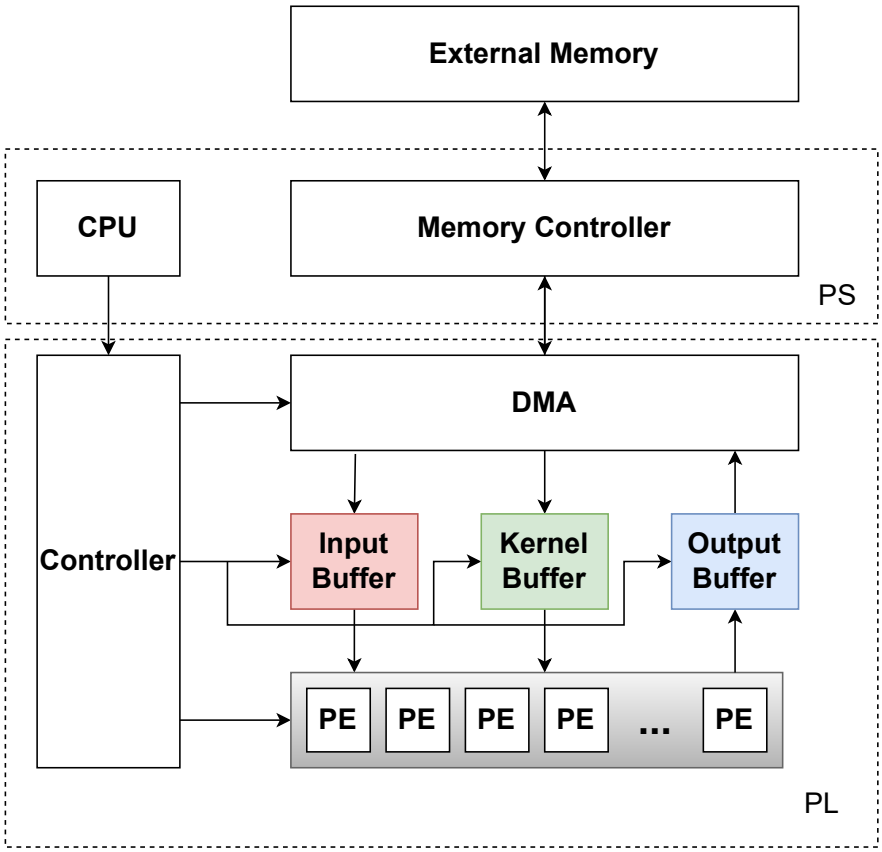

Figure 2.10: Generic CNN accelerator proposed by [25].

The generic architecture makes use of three on-chip buffers that allow for the reduction of data transfer overhead, by storing data locally. These buffers communicate with the main memory through Direct memory access (DMA) which not only fetches the input FMs and kernels into the buffers, for their

use in the PEs Processing element (PE) but also streams the outputs into the main memory for storage. Every element of the Programmable Logic (PL) is controlled by the controller, whose control signals are sent by the CPU.

Based on this generic architecture there are two main possible architectural approaches. The first involves mapping the entire CNN to the FPGA and all layers processed in a full dataflow. This approach is certainly the faster of the two since it minimizes memory transfers. However, it requires either small network configurations, an extremely large FPGA or even multiple FPGAs connected since mapping all operations performed in CNN requires a large number of resources. The second approach consists of processing each layer in sequence with a single engine. The engine is configured according to the characteristics of each layer (kernel size, sizes of the input and output maps, stride, etc.). This approach, known as Configurable Layer Processor (CLP) accelerator, requires more external memory accesses but supports models with different layer configurations, and bigger CNN architectures in a less expensive FPGA device, since the resource consumption of an individual layer is much smaller.

2.4 Conclusions

This chapter describes the CNN model and how it is used for the object detection problem. Different examples of state-of-the-art object detectors were presented and compared in terms of accuracy and inference speed. The YOLOv4 proved to have an accuracy that matches other state-of-the-art models, while having the fastest inference speed, making it suitable model for real-time object detection. Based on the YOLOv4 model, the R-YOLOv4 improves its ability to detect oriented objects.

Additionally, the opportunities for hardware acceleration on FPGA of CNN models were presented and two possible approaches were discussed. The full pipelined and the single engine implementations. The single engine accelerator while being slower, it is the most suitable approach for an edge computing application, since it uses fewer resources allowing the usage of a smaller FPGA.

Chapter 3

Design and Optimization of the Oriented Object Detection Model

In this chapter, the choice of network most suited to the proposed application is formalized, based on the conclusions of the last chapter, as well as a dataset to train the model. Then, optimizations applied to the model to reduce the hardware complexity and improve the throughput of the accelerator are described. Finally, accuracy results are determined for different configurations of the model.

3.1 Model Selection

The objective of this work is to implement an object detector accelerator in a SoC FPGA with the goal that it can be used for real-time oriented aerial object detection. With that objective in mind, an object detection model has to be chosen, and, based on the models presented and compared in the previous chapter the model chosen is an altered version of the R-YOLOv4. Based on the collected information, this is the most appropriate model for the task proposed. Not only is this model based on the YOLOv4, an object detection model with good accuracy for real-time applications, but its modifications also allow it to be more accurate when detecting oriented objects, which perfectly fits the objective of the work. Some modifications are made to the model at a cost of a minor precision loss to make it better suited for hardware acceleration. These modifications will be explained in section 3.3.

3.2 Dataset Selection

For the training of the model and testing of the application, a dataset has to be chosen. The dataset should be comprised of aerial images and support the detection of oriented objects by having an oriented bounding box (OBB) representation. To that end, the dataset chosen is the UCAS-AOD, which contains 1510 RGB images from Google Earth, having 2 total classes, cars and plains, with 14596 instances. Furthermore, this dataset supports OBB representation, hence matching all the dataset requirements.

In the figure 3.1 are a few examples of images of the UCAS-AOD dataset.

Figure 3.1: Example images from UCAS-AOD dataset.

3.3 Model Optimization

This work uses a modified version of R-YOLOv4 [20]. The original network contains 111 convolutional layers, each followed by one of three types of activation functions: Leaky ReLU, mish or linear (which is only applied to the output layers). Moreover, the network contains max pooling, upsampling, and shortcuts with concatenation.

To be more hardware-friendly, a few modifications were made to the original model at the cost of a small decrease in accuracy. These modifications resulted in a network that contains 66 convolutional layers all followed by the Leaky ReLU except for the output layers that have linear activation functions. Additionally, all concatenations were turned into sums, and the max pooling layers were removed.

Represented in table 3.1 is a description of the modified version of the RYOLOv4 network that contains all of its convolutional layers as well as their configurations.

Table 3.1: Modified RYOLOv4 network description table.

	Conv #	Layer	Input Size	Input Channels	Kernel Size	Stride	Padding	Output Channels	Output Size
	1	Conv0	608	3	3	1	1	32	608
	2	Conv1	608	32	3	2	1	64	304
CSP BLOCK 1	3	CSP conv1	304	64	1	1	0	32	304
	4	CSP conv2	304	64	1	1	0	32	304
	5	Bottleneck	304	32	1	1	0	32	304
	6	Bottleneck	304	32	3	1	1	32	304
	7	CSP conv3	304	32	1	1	0	32	304
	8	CSP conv4	304	32	1	1	0	64	304
	9	Conv2	304	64	3	2	1	128	152
CSP BLOCK 2	10	CSP conv1	152	128	1	1	0	64	152
	11	CSP conv2	152	128	1	1	0	64	152
	12	Bottleneck	152	64	1	1	0	64	152
	13	Bottleneck	152	64	3	1	1	64	152
	14	CSP conv3	152	64	1	1	0	64	152
	15	CSP conv4	152	64	1	1	0	128	152
	16	Conv3	152	128	3	2	1	256	76
CSP BLOCK 3	17	CSP conv1	76	256	1	1	0	128	76
	18	CSP conv2	76	256	1	1	0	128	76
	19	Bottleneck	76	128	1	1	0	128	76
	20	Bottleneck	76	128	3	1	1	128	76
	21	CSP conv3	76	128	1	1	0	128	76
	22	CSP conv4	76	128	1	1	0	256	76
	23	Conv4	76	256	3	2	1	512	38
CSP BLOCK 4	24	CSP conv1	38	512	1	1	0	256	38
	25	CSP conv2	38	512	1	1	0	256	38
	26	Bottleneck	38	256	1	1	0	256	38
	27	Bottleneck	38	256	3	1	1	256	38
	28	CSP conv3	38	256	1	1	0	256	38
	29	CSP conv4	38	256	1	1	0	512	38
	30	Conv5	38	512	3	2	1	384	19
CSP BLOCK 5	31	CSP conv1	19	384	1	1	0	192	19
	32	CSP conv2	19	384	1	1	0	192	19
	33	Bottleneck	19	192	1	1	0	192	19
	34	Bottleneck	19	192	3	1	1	192	19
	35	CSP conv3	19	192	1	1	0	192	19
	36	CSP conv4	19	192	1	1	0	512	19
SPP BLOCK	37	SPP conv1	19	512	1	1	0	256	19
	38	SPP conv2	19	256	3	1	1	512	19
	39	SPP conv3	19	512	1	1	0	256	19
	40	SPP conv4	19	256	1	1	0	256	19
	41	SPP conv5	19	256	3	1	1	512	19
	42	SPP conv6	19	512	1	1	0	512	19

	Conv #	Layer	Input Size	Input Channels	Kernel Size	Stride	Padding	Output Channels	Output Size
	43	Conv6	19	512	1	1	0	256	19
	44	Conv7	38	512	1	1	0	256	38
C5 BLOCK 1	45	C5 conv1	38	256	1	1	0	128	38
	46	C5 conv2	38	128	3	1	1	256	38
	47	C5 conv3	38	256	1	1	0	256	38
	48	Conv8	38	256	1	1	0	128	38
	49	Conv9	76	256	1	1	0	128	76
C5 BLOCK 2	50	C5 conv1	76	128	1	1	0	64	76
	51	C5 conv2	76	64	3	1	1	128	76
	52	C5 conv3	76	128	1	1	0	128	76
	53	Conv10	76	128	3	1	1	256	76
	54	Conv11	76	256	1	1	0	561	76
	55	Conv12	76	128	3	2	1	256	38
C5 BLOCK 3	56	C5 conv1	38	256	1	1	0	128	38
	57	C5 conv2	38	128	3	1	1	256	38
	58	C5 conv3	38	256	1	1	0	384	38
	59	Conv13	38	384	3	1	1	512	38
	60	Conv14	38	512	1	1	0	561	38
	61	Conv15	38	384	3	2	1	512	19
C5 BLOCK 3	62	C5 conv1	19	512	1	1	0	256	19
	63	C5 conv2	19	256	3	1	1	512	19
	64	C5 conv3	19	512	1	1	0	384	19
	65	Conv16	19	384	3	1	1	512	19
	66	Conv17	19	512	1	1	0	561	19

The reduction in the number of layers of the original model result from changes made to the CSP and C5 blocks. The C5 blocks were originally blocks of 5 consecutive convolutional layers, while in the modified version the last two layers of said block were removed resulting in the reduction of 2 layers per each C5 block in the network. The CSP block consists of a 4 convolutions and a bottleneck residual block, which an example of can be seen in figure 3.2. One of the parameters of the CSP block is a

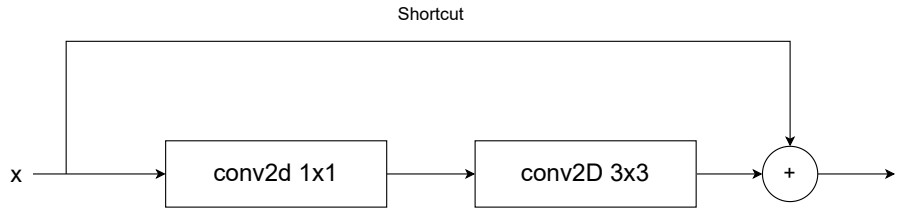

Figure 3.2: Residual Bottleneck block.

factor that indicates how many times the bottleneck is applied. In the original model for each of the 5 CSP blocks the said parameter assumes the values 1, 2, 8, 8 and 4, while in the modified model all CSP blocks have the factor set at 1 significantly reducing the number of convolutions of the network.

The substitution of concatenation layers with sums helped reduce the depth of various layers reduc-

ing the number of parameters and thus making the model consume less memory and have to perform less operations. Additionally some of the layers in the additional model had biases that were removed in the modified version, futher reducing the number of parameters of the model.

The accuracy of the original and modified versions of the RYOLOv4 are represented in table 3.2.

Table 3.2: Original and modified RYOLOv4 accuracy.

Model	MACS	Parameters	Car map_{50}%	Airplane map_{50}%	map_{50}%
Original RYOLOv4	55.8 G	56.7 M	78.7	96.5	87.6
Modified RYOLOv4	26.7 G	18.1 M	75.5	94.7	85.1

.

From the results, it can be observed that the total accuracy reduces 2.5%, but the complexity (MACS - Multiply-Accumulate) reduces about $2\times$ and the number of parameters also reduces about $3\times$.

3.4 Tools for Training and Quantization

After the selection of the model, it has to be trained, quantized and its parameters extracted to be used by the hardware accelerator. This section describes the tools necessary for training, quantization, and parameter extraction. The adopted training framework was PyTorch and the quantization framework was Brevitas. A few custom tools were developed to help export the model to hardware.

3.4.1 PyTorch

Pytorch is a python open-source machine-learning framework that allows for the development, training, and testing of machine-learning models. Even though it supports a wide variety of models, this framework is mostly used for deep learning and neural networks. These models can be defined in Pytorch as several functions, each representing a layer and where the arguments are the layer's parameters such as kernel size, stride, number of channels for the output and input, and padding. An example of a simple network developed in Pytorch is represented in 3.3.

```
 import torch.nn as nn

class ExampleNet(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride, padding):
        super(ExampleNet, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu1 = nn.ReLU()

    def foward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu1(x)

        return x
```

Figure 3.3: Simple example of a network done with Pytorch.

The example illustrated in the figure is a simple model with one convolutional layer, one batch normalization layer followed by a ReLU activation function.

3.4.2 Brevitas

Brevitas is a Python library that extends Pytorch, allowing for quantization-aware training to be applied to neural network models. There are quantized versions of the PyTorch layers that can be replaced in the original PyTorch model to help recreate hardware's low precision datapath during training, reducing the quantization error during inference in hardware. These layers are similar to the normal Pytorch layers, but require a quantization engine where all the quantization configuration parameters are declared.

The key parameter for quantization is the bitwidth, which defines the number of bits used to represent the data. Other parameters are also important to help define the exact representation of data. The quantization mode refers to the application of one of the supported quantization functions provided by Brevitas, as detailed in section 2.3. The quantization type indicates the data representation format. Additionally, the scaling type limits the range of values that the quantization scaling factor can assume during training. Furthermore, Brevitas allows for the activations and weights to be quantized separately in the same layer, meaning that weights and activations can be quantized with different sizes in the same layer.

The activations are quantized during the application of the activation functions while the weights are quantized during the backward step that occurs during training. The quantization engine used for the project can be seen in figure 3.4 and, an example of a simple network using Brevitas is represented in 3.5. This example is simply the result of the conversion of the the Pytorch model in 3.3 to a Brevitas quantized one.

As can be seen from the figure, the quantized version of the convolutional layer replaced the layer in the original model. The ReLu function was also replaced by its quantized version. The batch normalization layer is not quantized, since it will be merged with the convolutional layer and is not considered during inference.

```python
#Engine declaration
from brevitas.inject import ExtendedInjector
from brevitas.quant.solver import WeightQuantSolver, ActQuantSolver
from brevitas.core.bit_width import BitWidthImplType
from brevitas.core.quant import QuantType
from brevitas.core.restrict_val import RestrictValueType, FloatToIntImplType
from brevitas.core.scaling import ScalingImplType
from brevitas.core.zero_point import ZeroZeroPoint
from brevitas.inject.enum import ScalingImplType, StatsOp, RestrictValueType

class CustomQuant(ExtendedInjector):
    bit_width_impl_type = BitWidthImplType.CONST
    scaling_impl_type = ScalingImplType.CONST
    restrict_scaling_type = RestrictValueType.POWER_OF_TWO
    zero_point_impl = ZeroZeroPoint
    float_to_int_impl_type = FloatToIntImplType.ROUND
    scaling_impl_type = ScalingImplType.STATS
    scaling_stats_op = StatsOp.MAX
    scaling_per_output_channel = False
    bit_width = None
    narrow_range = True

    signed = True
    quant_tyoe = QuantType.INT

class CustomWeightQuant(CustomQuant,WeightQuantSolver):
        scaling_const = 1.0

class CustomActQuant(CustomQuant, ActQuantSolver):
   signed=True
    float_to_int_impl_type = FloatToIntImplType.FLOOR

class CustomSignedActQuant(CustomQuant, ActQuantSolver):
    signed=True
    float_to_int_impl_type = FloatToIntImplType.FLOOR
```

Figure 3.4: Quantization Engine.

```python
import torch.nn as nn
import brevitas.nn as qnn

weightBitWidth=8
activationBitWidth=8

class ExampleNet(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride, padding):
        super(ExampleNet, self).__init__()
        self.conv1 = qnn.QuantConv2d(in_channels, out_channels, kernel_size, stride, padding,
        ↪  weight_bit_width=weightBitWidth, weight_quant=CustomWeightQuant,return_quant_tensor=True)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = qnn.QuantReLU(bit_width=activationBitWidth, act_quant=CustomSignedActQuant,
        ↪  return_quant_tensor=True)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)

        return x
```

Figure 3.5: Simple example of a network done with Brevitas.

3.5 Custom Tools

To help the development of the accelerator, some extra tools were needed, not only to convert and integrate the original Pytorch model with a Brevitas quantized one, but also for parameter extraction and test generation. Most of these tools were originally developed in [26] and were modified to fit the requirements of this project. These tools are the Brevitas converter, the batch normalization merger, the weights extractor, and the bin file generator. All these tools were developed using the Python programming language.

3.5.1 Brevitas Converter

The Brevitas converter is a program that takes the original Pytorch model description and automatically converts it to a Brevitas quantized model description. This tool allows for a fast quantization of the original model.

The tool has two additional scripts that allow for the conversion of the weights trained from the original model into quantized, Brevitas-compatible weights, which allows for training with pre-trained weights, instead of training the model from scratch which speeds up the very time-consuming training process.

3.5.2 Bach Normalization Merger

The batch normalization merger is a program that iterates through every layer and merges the batch normalization layer with its respective convolutional layer. This is possible because since a batch normalization layer is effectively a 1x1 convolution and there are no non-linear layers between the convolutional and batch normalization layers. The processing of merging is completed by applying (3.1) to each layer of the model. The bias are irrelevant since they were removed from the model utilized in this project.

$$Output = W_{BN} \times (W_{Conv} \times FM_{Value} W_{BNMerged} = W_{BN} \times W_{Conv} Output_{BNMerged} = W_{BNMerged} \times FM_{Value} \tag{3.1}$$

This batch merging is useful because it reduces memory consumption and computational needs, further speeding up the training process. It also makes the extraction of the weights easier.

3.5.3 Weights Extractor

The hardware accelerator runs the inference of the model using the weights determined after training. Therefore, a program is needed to extract the weights produced during the quantization aware training, so that they can be used by the hardware accelerator.

The weights extractor tool is a program that extracts the weights from the brevitas quantized model and stores them into binary files in a fixed point format. These weights are stored z-wise, which is the format that allows for more parallelism. While in the original script from [26] these weights are packed in 64-bit words, in this project the weights are packed into 128-bit words. When there are not enough weight values to fill an entire 128-bit word, the remaining values are filled with zeroes.

3.5.4 Bin File Generator

The Bin File Generator developed in this work generates random arrays of any shape, determined by a configuration specification, and then stores them in a bin file in the format described in 3.5.3, z-wise, 128-bit words. The purpose of this script was to create tests to validate the functionality of the accelerator without having to run the whole network and wait for the training to be complete. With this tool, multiple tests for different convolution configurations can be generated and then used to validate the application.

3.6 Design Flow

Using Pytorch, Brevitas and the custom tools, the model quantization and export tasks to generate the quantized weights for the SoC FPGA are achieved follwing the design flow represented in figure 3.6.

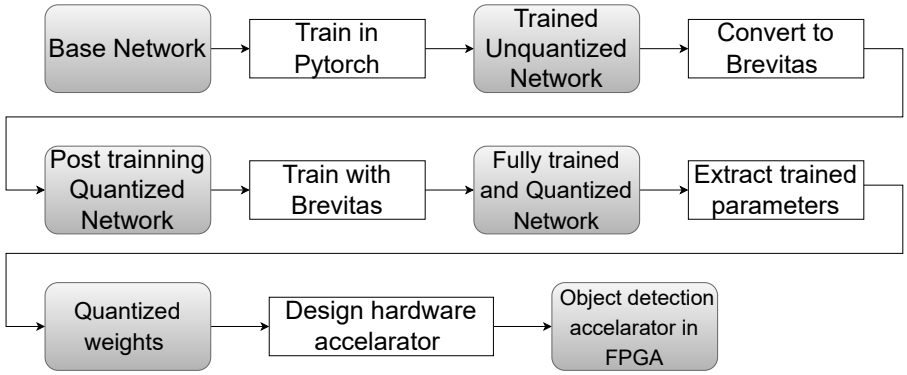

Figure 3.6: Proposed project design flow.

The base model is first trained using the PyTorch framework, producing trained model with weights and activations represented in single-precision floating-point. Then, using the Brevitas Converter tool, the model is quantized alongside its previously trained floating point weights and activations which generates a quantized Brevitas network and quantized weights and activations. Afterwards, Brevitas is used so that quantization-aware training can be applied to the network to reduce quantization error in inference. At this step, multiple quantization configurations are explored to find a good trade-off between computational complexity and accuracy. Typically, bitwidths of 2, 4, and 8 are considered for both weights and activations.

After quantization-aware training, the model is ready to have its quantized weights extracted, using the weights extractor tool, and used in the design, testing, and validation of the hardware accelerator. The hardware accelerator will be designed by developing an Intellectual Property (IP) core capable of running configurable layers. The design and development of the hardware accelerator will be discussed in Chapter 4.

3.7 Quantization Analysis of R-YOLOv4

Different quantization were tried with the reduced R-YOLOv4 model with varying weights and activation bitwidths. The mAP_{50} was annotated for each quantization.

The quantization process followed the design flow described in section 3.4. The models were trained for 100 epochs with the Adam optimizer. Table 3.3 reports the results for two different quantizations: 8×8 and 8×4.

Table 3.3: RYOLOv4 accuracy with and without quantization. Two different quantizations were considered: 8×8 and 8×4

Model	MACS	Parameters	Car map_{50}%	Airplane map_{50}%	All map_{50}%
Original RYOLOv4	55.8 G	56.7 M	78.7	96.5	87.6
Modified RYOLOv4	26.7 G	18.1 M	75.5	94.7	85.1
Quant. Modified RYOLOv4 8×8	26.7 G	18.1 M	73.5	92.3	82.9
Quant. Modified RYOLOv4 8×4	26.7 G	18.1 M	62.9	85.4	74.2

.

The accuracy of the modified model with quantization 8×8 reduced 2.2 percentual points compared to the same non-quantized model. Considering the more aggressive quantization, 8×4, the model accuracy dropped about 11 percentual points. Therefore, it was decided to consider the 8×8 quantization. Compared to the original model, the model implemented in hardware has an accuracy drop of 5.7 percentual points.

3.8 Conclusions

This chapter describes the design flow of the optimized model from the original floating-point model to the quantized model and finally the extraction of weights to be used in the hardware accelerator.

The original model was modified to reduce the computational complexity and memory requirements. The complexity reduces about $2\times$ and the number of parameters reduces about $3\times$ at the cost of 2.5% drop in accuracy.

Then, different quantitazions were tested to find a quantized model with a good trade-off between accuracy and computational complexity. The 8×8 quantization was chosen with an accuracy reduction of 5.7 percentual points compared to the original floating-point model.

Chapter 4

Hardware Accelerator Design

In this chapter, the design of the hardware accelerator for the model discussed in the previous section is presented.

The accelerator was developed using Xilinx Vitis High level synthesis (HLS) version 2022.1, and the target FPGA was an AMD/Xilinx Zynq Ultrascale+ MultiProcessor System-On-Chip (MPSoc) XCZU7EV from Xilinx integrated in a ZCU104 board. Besides the FPGA, the board also includes 2GB of DDR4 memory and interfaces to peripherals.

Vitis HLS is a high-level synthesis tool where hardware descriptions can be done using the C/C++ programming languages as well as in built pragmas for optimization. The development process consists of producing the logic for the operations meant to be performed by the accelerator and then testing and validating their functionality through simulation. Afterwards, the design is synthesized and into an Register-transferlevel (RTL) design and its implementation is validated. Finally the RTL IP is exported and added to the Vivado's block design in order to finalize the implementation of the accelerator in the device.

4.1 IP Core Design

The objective of the accelerator is to make the inference of the model described in the previous chapter faster. This is done by creating an IP capable of performing all the convolutional layers of the RYOLOv4 model described in the previous chapter, since this is where most of the operations are performed and where there is more parallelism to be exploited.

The core was designed to execute the layers in sequence, that is, a single engine is considered that can be configured according to the characteristics of each layer of the network model. The IP core is therefore configurable to support the execution of any convolutional layer present in the RYOLOv4 model. More specifically, the IP supports convolutions with kernels of size 3×3 and 1×1, and strides of 1 or 2. It also only supports the leaky ReLU activation function.

The IP explores two types of parallelism: inter-layer parallelism (where multiple output maps are generated in parallel) and intra-kernel parallelism (where multiple MAC units are used to run a single

convolution in parallel). The developed architecture has 16 PEs to explore inter-layer parallelism each with 16 parallel MAC units. A larger number of PEs and MACs can be considered at the cost of more hardware resources. Considering that the activations are quantized with 8 bits, 16 output activations are produced in parallel and packed in a 128-bit word output to be sent to the output interface. Since each PE runs 16 MACs operations, it receives 16 activations and weights in parallel packed also in 128 bits. The architecture of the IP is represented in figure 4.1.

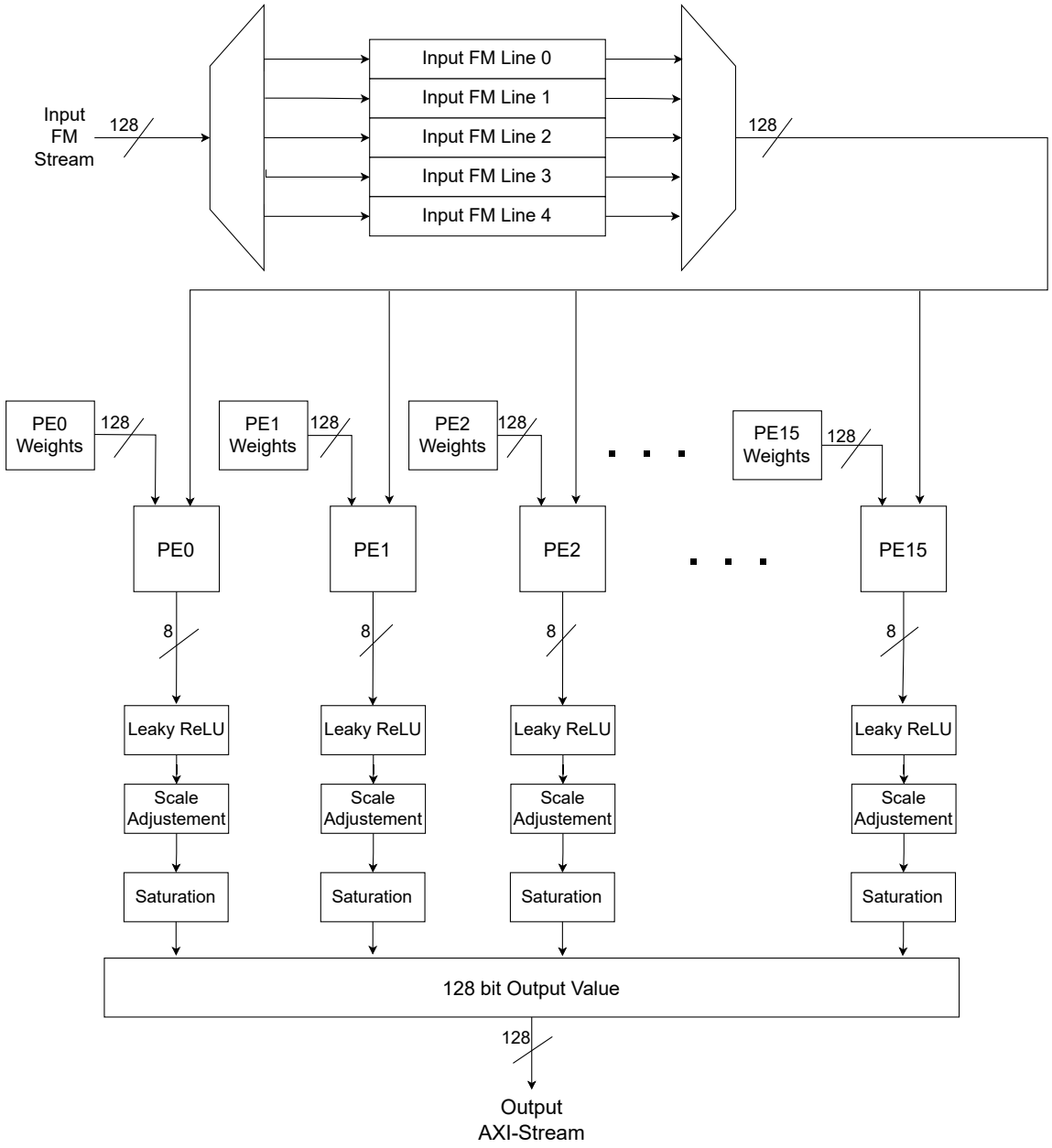

Figure 4.1: Convolution IP Diagram.

Before running a layer, all weights are stored in distributed on-chip memory. The higher number of weights among all layers is considered to size the local memories of the IP allocated to store the

weights. These are stored in the Ultra random access memory (URAM) of the FPGA. To guarantee a single transfer of weights for each execution of a convolutional layer, the size of the local weight memories is set according to the layer with the larger number of weights. Input and output maps are only stored partially in on-chip memory since these are processed in stream. Only 5 lines from the input FM are stored locally in Block random access memory (BRAM) and the output activations are packed and sent immediately to external memory to be read by the next layer. This on-chip storage allows the IP to process any convolution configuration while reducing the number of data transfers with the external memory which would affect performance.

Furthermore, the memory for the filters is partitioned into 16 sections, one for each PE for local PE access, allowing for simultaneous access to different sections of the filters memory. In order to ensure a PE workload balance, the filters are evenly distributed through the partitions, meaning that for a convolution with 32 filters, each PE stores locally two filters.

The IP is configured to receive both the maps and the weights in a z-wise configuration, meaning that they are sorted through their Z-axis. The easiest way to understand this data configuration is to imagine a map with $5 \times 5 \times 24$, which means that this map has 24 input channels. This map is organized so that, in the first place, the IP receives all 24 channels at position (0,0), then all the 24 channels of position (1,0), and so on until all the 25 positions are sent. This is done by first iterating through X and then Y, meaning, all the 24 channels of a FM line are sent before going onto the next. The values are packed into 128-words and each of the map values is 8 bits meaning each bus has 16 values. For the previous example since there are 24 channels, two 128 bit words are needed to represent one pixel of the map, with the first 126-bit word being the first 16 channels and the second being the remainder 8. Since 8 activations are not enough to fill the 128-bit bus, the remainder free positions are filled with zeros, so that these useless positions will not affect the computations. In figure 4.2 there is a visual representation of this data organization. This representation is true for both maps and filters.

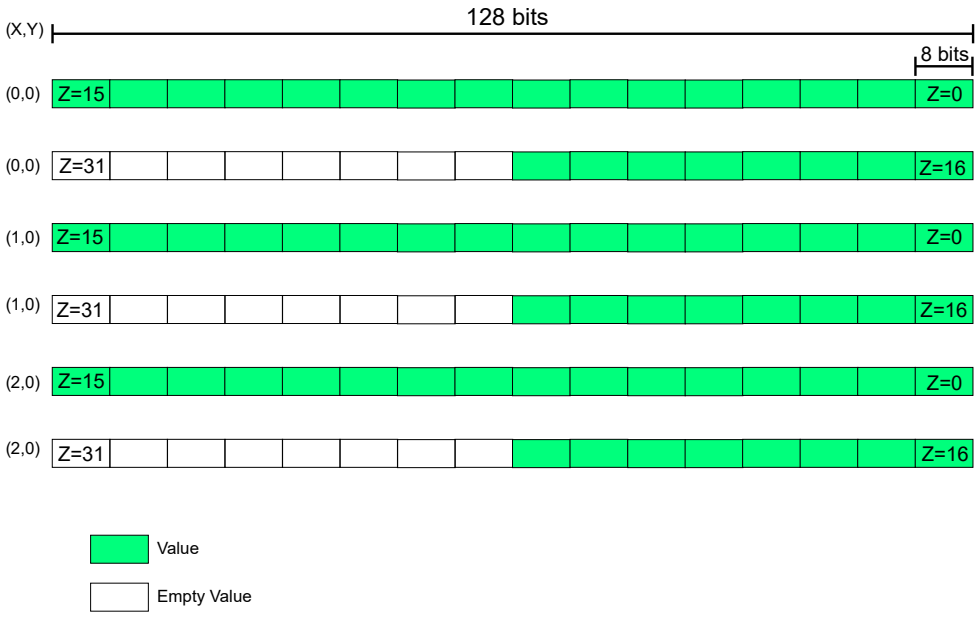

Figure 4.2: Z-wise data organization.

The figure illustrates an example of packing activations of a map with 24 channels. Since the number of channels is not multiple of 16, the data is padded to 16 positions.

The order in which the input activations are read depends on the kernel size. The neural network model considered in this work includes kernels of size 3×3 and 1×1. The z-lines of pixels must be read to follow the window of the kernel. With a kernel of size 3×3, the PEs receive three vectors of three sequential lines. For a kernel of size 1×1 only a single vector is read. The sequence of reading addresses of the feature map lines is generated by an address generator unit that shares the output address with all PEs (not represented in the figure). The filters in each PE are read in sequence since the order in which they are stored in memory matches the sequence with which the activations are read.

To execute one layer, the IP starts by receiving the configuration parameters of said layer: map size, kernel size, padding, number of input channels, number of filters, stride, activations, via an AXI4-Lite port. Then, it reads through the AXI-Stream port all the weight values and stores them in the URAMs, and also reads the initial three or four liens from the input FM (depending on the kernel size) to enable the execution of the first convolutions with any value of stride. The remaining lines of the input map are read in parallel with the calculation of the convolutions over the lines present in the input map memory.

The processing flow for the convolution operation begins by resetting all PE accumulators. Next a 128-bit word, which contains the first 16 channels of the pixel of the FM, is sent to the PE along with another 128-bit word containing the 16 corresponding weights for the convolution operation. Inside the PE, the dot product between the activations and the weights is calculated by performing 16 MAC operations. In the subsequent iteration, the next 16 channels of the first input activation are sent, repeating the previous step until all channels of the activation are processed. Once all channels of the first input activation are processed, the convolution moves to the next activation, performing all z-wise operations and continuing this process until all activations required for the output are iterated over.

A visual representation of a PE is represented in 4.3.

The sixteen parallel multipliers are followed by an adder tree and an accumulator. Once the accumulator of each of the PEs has an output activation ready, these values are passed through the Leaky ReLU function, scaled according to the scale of the fixed point representation, and then saturated within the 8-bit signed representation range (-127 to 127) to avoid over and underflow. Finally, the accumulator values are concatenated into a 128-bit word to be streamed out through the output AXI-Stream port.

Since each PE accesses different filters, the output pixels produced will correspond to different channels of the output, which will preserve the z-wise organization in the output feature map.

Simultaneously, at every iteration of the processing elements, a 128-bit word composed of 8-bit activations is read through the AXI-Stream port and stored in local memory. This allows reading the next input FM lines to be processed while processing the previous ones, masking the data transfer overhead. This process needs to be controlled cautiously to prevent the newly read lines to overwrite the yet to be processed lines read before. The IP is entirely pipelined in order to maximize throughput.

A modified version of the Leaky ReLU function was considered to simplify its hardware implementation. The original Leaky ReLU function is defined as $max(0.1 \times x, x)$. This requires a multiplication by 0.1. The modified Leaky ReLU function is defined as $max(0.09375 \times x, x)$ to simplify this operation.

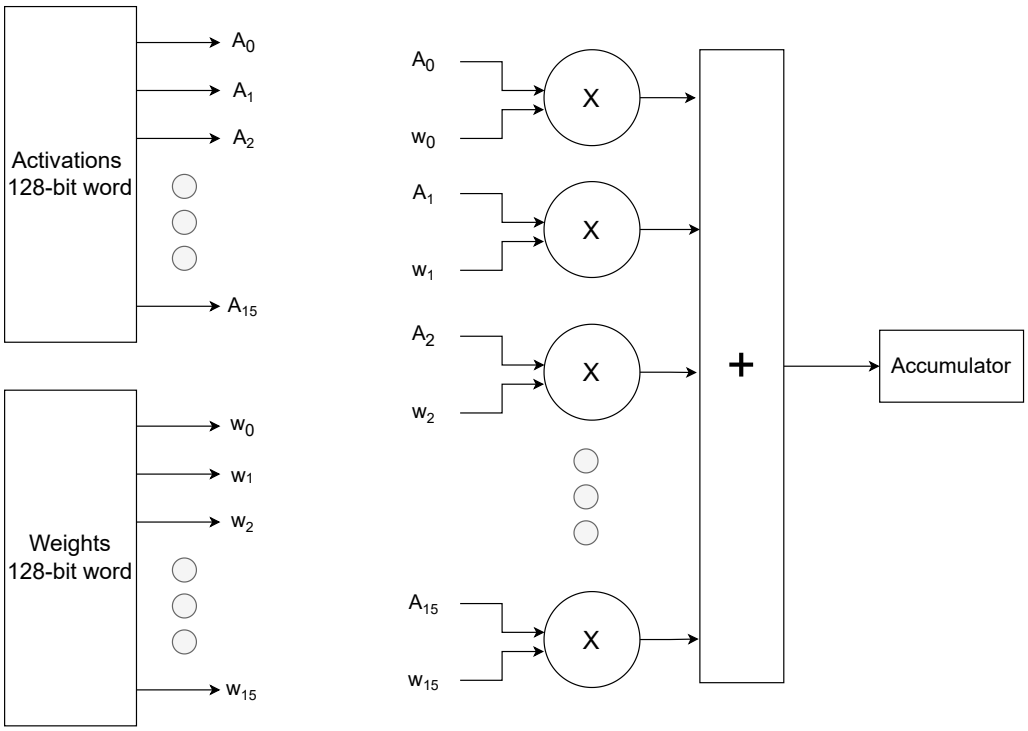

Figure 4.3: Processing Element Diagram.

Knowing that $0.09375 = 2^{-4} + 2^{-5}$, a multiplication by this constant is implemented as an addition of two shifted values. The modified Leaky ReLU was considered during the training of the model.

4.2 Vits HLS Results

The Vitis HLS tool generates reports detailing the pipeline characteristics and utilization estimates of the IP. These pipeline characteristics are obtained after synthesis, allowing for the observation of the iteration interval and iteration latency for each loop. The iteration latency represents the number of clock cycles required to complete one full iteration of a loop from start to finish, while iteration interval, also known as initiation interval is the number of clock cycles needed to start consecutive iterations of a loop. While the impact of iteration latency on performance becomes less significant as the number of iterations in a loop increases, the iteration interval is crucial for the overall performance of the loop. A lower iteration interval allows iterations to start more frequently, resulting in more effective pipeline utilization.

The pipeline characteristics of the IP are represented in the table 4.1. It is important to mention that the clock period considered is $10\,\mathrm{ns}$.

Table 4.1: Pipeline Characteristics of the Convolution IP core.

	Iteration Interval (clk cycles)	Iteration Latency (clk cycles)
Read Weights Loop	1	1
Read Initial FM Loop	1	2
Convolution Loop	1	43

As it can be observed in 4.1 the iteration interval for all of the loops is 1 which very important for the performance of the IP. An iteration interval greater than 1 would have a multiplicative effect on the time needed to run the loop, as it reduces the frequency at which new iterations start, thereby decreasing throughput and increasing total execution time.

The utilization estimates from HLS can also be obtained post-synthesis, however, these are very conservative estimations. A much more accurate resource utilization estimate is obtained from the post-implementation report. The resources estimate from the post-implementation report of Vitis HLS is in table 4.2.

Table 4.2: Resource utilization estimates from Vitis HLS.

Resource	Used	Available	Utilization (%)	Guideline(%)
LUT	18293	230400	7.9	70
FF	9461	460800	2.1	50
DSP	281	1728	16.3	80
URAM	64	96	66.7	80
BRAM (36K)	21.5	312	6.9	80

As intended the resource consumption of the accelerator does not exceed the device hardware limitations. Most of the DSPs used are bound to the PEs to execute the MAC operations. However, there are a few more that are used for execute calculations and other control operations.

The local memory of the IP was divided between BRAMs and URAMs. The BRAMs are used to store the feature map lines and were dimensioned to be able to support the biggest possible line. This happens in the second layer of the model when the input map dimensions are $608 \times 608 \times 32$. The map values are organized z-wise and the IP only stores 5 lines of the map, that is, $97\,\mathrm{KB}$ of memory are needed to store the lines. Since each BRAM has $36\,\mathrm{Kb}$, the amount of BRAms needed to store the FM lines can be calculated, as it is shown in (4.1).

$$\#\text{BRAMs} = \frac{95KB}{36Kb} = 21.1 \approx 21,5 \text{ BRAMs} \tag{4.1}$$

The URAMs is used to store the weights and were dimensioned to support the largest layer of the model. The largest layer features $512 \times 384 \times 3 \times 3$ weights which is equivalent to $1,69$MB of weights. Since the memory is partitioned into 16 sections, each partition needs to be able to store $110.6\,\mathrm{KB}$. Knowing that each URAM has a capacity of $288\,\mathrm{Kb}$, the ammount of URAMs needed for each partition

can be calculated, as it is shown in (4.2).

$$\#\text{URAMs} = \frac{110,6KB}{288Kb} = 3.1 \approx 4 \text{ URAMs} \tag{4.2}$$

Since there are 16 partitions and each uses 4 URAMs, the total number of URAMs used is 64.

4.3 Conclusions

A single computing engine was developed to execute the convolutional layers in sequence. The engine is configurable and supports the execution of convolutions with kernels of size 3×3 and 1×1, strides of 1 or 2, and the modified leaky ReLU activation function.

The accelerator has 16 processing elements, each running the convolution with a different filter in parallel, and each PE has 16 MACs in parallel. Therefore, the engine efficiently runs 256 MACs in parallel.

The reports from HLS show that all loops within the IP have an iteration interval of 1, which means the IP will have optimal throughput and the design does not exceed the hardware limitations of the FPGA, nor the recommended limits of Vitis HLS.

Chapter 5

HW/SW System Implementation and Results

In this chapter, the development and results of the HW/SW system to implement the RYOLOv4 are presented.

5.1 Hardware/Software Architecture

The hardware/software architecture of the system is comprised of the accelerator and the processor.

Both the Processing System (PS) and the PL of the target FPGA (Zynq UltraScale+ XCZU7EV) are used to implement the system. The PL block implements the IP explained in the previous section which was exported as an IP core and integrated into the FPGA. The PS block equipped with a quad-core Arm Cortex-A53 Application Processing Unit (APU) runs the software of the system.

For the interface between the PS and the PL blocks, the Advanced eXtensible interface (AXI) communication standard is used, which provides high bandwidth, low latency connections, resulting in efficient data transfers.

The system requires the transfer of the FM and filters from the external memory to the hardware accelerator. The access to the external memory is done through one high-performance port of the PS-PL interface and with the use of DMA blocks, which are set up and configured by the PS to send the FM and filters as well as receive the outputs produced by the accelerator.

The hardware/software architecture was elaborated in Vivado, whose block design is represented in figure 5.1.

As it can be seen in 5.1, other then the PL and PS, a DMA is included to transfer the weights and input FM values from external memory into the IP and transfer the output FM values produced by the IP to the external memory via the AXI-Stream protocol.

The DMA connects to the external memory controller through HP0 (High-Performance) port of the ZYNQ Ultrascale+ PS. This connection is done by the AXI SmartConnect block.

There is also an AXI Interconnect block, that connects the AXI-Lite interface of the PS to both the

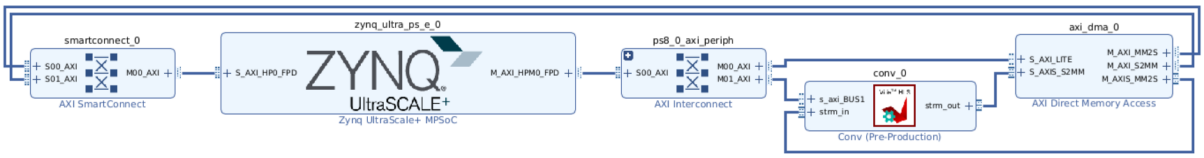

Figure 5.1: Vivado Block Design.

DMA and the PL block. This allows the PS to send the configuration parameters of the convolutional layer via the AXI-Lite protocol, as well as to control the scheduling of data being transferred between the IP and the external memory through the DMA.

The model is executed by running the convolutional layers one at a time until all layers are executed.

The software platform created in the Vitis IDE runs in the ARM processor of the ZYNQ FPGA. It iteratively configures the convolutional engine and instructs the accelerator to start executing the convolution. The configuration parameters of the engine are sent through the AXI-Lite interface. Then, a start signal through the same interface instructs the core to start operating. The accelerator first reads all weights and stores them in the local memories of the PEs. These are read in the predefined order from the external memory. This transfer is executed by the DMA, which are configured to transfer the right number of weights, using the AXI-Stream interface. The output transfer is done by the same DMA, which is configured before starting the stream of input FM values so that the outputs can be streamed out of the IP and into the external memory as soon as they are ready.

The processor then waits for the end of the transfer of output activations to external memory, meaning that the core has finished the execution. Once the output stream finishes and the signal done of the control register of the AXI-Lite is read, the convolution is completed. The process described must be repeated and chained together by feeding the output of a previous layer as the output of the next. The upsampling and FM sum layers are not supported, so these must be executed in software.

5.2 System Results

This section presents the experimental results of the project. First, the Vivado results from the model are introduced. Following that, the performance results of the model HW/SW implementation are presented and compared with other solutions.

5.2.1 Resource Utilization

The resource utilization of the hardware architecture illustrated in Figure 5.1 after place and route is found in table 5.1.

Table 5.1: Vivado's Resource Utilization.

	LUT (230 400)		FF (460 800)		DSP (1728)		BRAM (312)		URAM (96)	
	Used	%	Used	%	Used	%	Used	%	Used	%
Convolution IP	25 577	11.1	10 720	2.3	163	9.4	21.5	6.9	64	66.7
DMA	2077	0.9	3217	0,7	0	0	5	1.6	0	0
ps8_axi_periph	1321	0.6	1446	0.2	0	0	0	0	0	0
Smartconnect	2151	0.9	3910	0.8	0	0	0	0	0	0
Total	31 135	13.5	19 326	4.2	163	9.4	26.5	6.9	64	66.7

As shown in Table 5.1, the majority of the FPGA resources are utilized by the developed convolution IP and the DMA. Both the BRAM and URAM usage estimates from the HLS (Table 4.2) are accurate. The remaining components have a minimal impact on FPGA resource usage.

The most used resource is memory, in particular, URAM to store the weights of the model. The architecture can be designed with less memory. However, this would mean that the weights of the larger layers would not fit into internal memory at the same time. Therefore, the input map would have to be read more than once and the output maps would be produced in chunks which would then have to be concatenated. Besides, reducing data transfers with external memory also reduces the energy consumption.

Another important aspect is that only around 13% of LUTs and 10% of DSPs were used. Since the proposed accelerator is scalable in terms of parallelism, the system's performance could be improved by increasing parallelism at the cost of more resources.

5.2.2 Performance Results

Each convolutional layer from the model was run individually on the ZCU104. The output of each layer was validated as correct and its timings were registered. Furthermore, a software-only convolution was also developed using simple C, with which the outputs were validated and the execution times were also collected to be compared with those obtained with the accelerator. It is relevant to mention that the C code for the software code is not optimized and thus the software execution results could be potentially better.

The processing time of each layer was calculated, including the data transfers in and out of the FPGA, with the accelerator running with an operating frequency of 100 MHz. The theoretical time the accelerator would need to complete each layer was also calculated by multiplying the number of cycles each layer takes to complete by the clock period of $10\,\mathrm{ns}$. The number of cycles the accelerator takes to complete a layer is calculated by dividing the number of MAC operation of a layer by the number of MACS the accelerator performs per cycle, which in this case is 256. The timing results for each layer can be found in table 5.2.

Table 5.2: Performance results per layer.

Conv #	Input Size	Input Channels	Kernel Size	Stride	Padding	Output Channels	Output Size	#MACs (M)	#Cycles	Theoretical time (ms)	Real Time (ms)	SW Time (ms)
1	608	16	3	1	1	32	608	1703.41	6 653 952	66.540	69.29	320 160
2	608	32	3	2	1	64	304	1703.41	6 653 952	66.540	74.115	319 028
3	304	64	1	1	0	32	304	189.27	739 328	7.393	11.437	38 087
4	304	64	1	1	0	32	304	189.27	739 328	7.393	10.896	38 081
5	304	32	1	1	0	32	304	94.63	369 664	3.697	5.087	19 139
6	304	32	3	1	1	32	304	851.71	3 326 976	33.270	34.562	159 924
7	304	32	1	1	0	32	304	94.63	369 664	3.697	5.087	19 141
8	304	32	1	1	0	64	304	189.27	739 328	7.393	8.828	38 233
9	304	64	3	2	1	128	152	1703.41	6 653 952	66.540	70.209	318 745
10	152	128	1	1	0	64	152	189.27	739 328	7.393	9.17	38 007
11	152	128	1	1	0	64	152	189.27	739 328	7.393	9.172	38 005
12	152	64	1	1	0	64	152	94.63	369 664	3.697	4.432	19 071
13	152	64	3	1	1	64	152	851.71	3 326 976	33.270	34.013	1 569 683
14	152	64	1	1	0	64	152	94.63	369 664	3.697	4.418	19 063
15	152	64	1	1	0	128	152	189.27	739 328	7.393	8.13	38 080
16	152	128	3	2	1	256	76	1703.41	6 653 952	66.540	68.827	318 764
17	76	256	1	1	0	128	76	189.27	739 328	7.393	8.35	37 963
18	76	256	1	1	0	128	76	189.27	739 328	7.393	8.22	37 965
19	76	128	1	1	0	128	76	94.63	369 664	3.697	4.098	19 023
20	76	128	3	1	1	128	76	851.71	3 326 976	33.270	33.829	159 314
21	76	128	1	1	0	128	76	94.63	369 664	3.697	4.097	19 025
22	76	128	1	1	0	256	76	189.27	739 328	7.393	7.814	37 999
23	76	256	3	2	1	512	38	1703.41	6 653 952	66.540	69.055	317 584
24	38	512	1	1	0	256	38	189.27	739 328	7.393	8.054	37 934
25	38	512	1	1	0	256	38	189.27	739 328	7.393	8.055	37 934
26	38	256	1	1	0	256	38	94.63	369 664	3.697	3.991	19 004
27	38	256	3	1	1	256	38	851.71	3 326 976	33.270	34.247	158 665
28	38	256	1	1	0	256	38	94.63	369 664	3.697	3.991	19 002
29	38	256	1	1	0	512	38	189.27	739 328	7.393	7.775	37 955
30	38	512	3	2	1	384	19	638.78	2 495 232	24.952	27.711	118 455
31	19	384	1	1	0	192	19	26.62	103 968	1.040	1.226	5 379
32	19	384	1	1	0	192	19	26.62	103 968	1.040	1.226	5 364
33	19	192	1	1	0	192	19	13.31	51 984	0.520	0.601	2 723
34	19	192	3	1	1	192	19	119.77	467 856	4.679	5.143	22 180
35	19	192	1	1	0	192	19	13.31	51 984	0.520	0.6	2 724
36	19	192	1	1	0	512	19	35.49	138 624	1.386	1.549	716 233
37	19	512	1	1	0	256	19	47.32	184 832	1.848	2.147	9 525
38	19	256	3	1	1	512	19	425.85	1 663 488	16.635	18.233	78 720
39	19	512	1	1	0	256	19	47.32	184 832	1.848	2.147	9 523
40	19	256	1	1	0	256	19	23.66	92 416	0.924	1.061	479 397
41	19	256	3	1	1	512	19	425.85	1 663 488	16.635	18.233	78 716
42	19	512	1	1	0	512	19	94.63	369 664	3.697	4.168	18 994

Conv #	Input Size	Input Channels	Kernel Size	Stride	Padding	Output Channels	Output Size	#MACs (M)	#Cycles	Theoretical time (ms)	Real Time (ms)	SW Time (ms)
43	19	512	1	1	0	256	19	47.32	184 832	1.848	2.147	9 524
44	38	512	1	1	0	256	38	189.27	739 328	7.393	8.055	37 929
45	38	256	1	1	0	128	38	47.32	184 832	1.848	2.103	9 534
46	38	128	3	1	1	256	38	425.85	1 663 488	16.635	17.089	79 360
47	38	256	1	1	0	256	38	94.63	369 664	3.697	3.991	19 006
48	38	256	1	1	0	128	38	47.32	184 832	1.848	2.102	9 537
49	76	256	1	1	0	128	76	189.27	739 328	7.393	8.349	37 965
50	76	128	1	1	0	64	76	47.32	184 832	1.848	2.239	9 543
51	76	64	3	1	1	128	76	425.85	1 663 488	16.635	16.838	79 695
52	76	128	1	1	0	128	76	94.63	369 664	3.697	4.097	19 025
53	76	128	3	1	1	256	76	1703.41	6 653 952	66.540	67.295	318 571
54	76	256	1	1	0	561	76	829.53	3 240 336	32.403	33.449	166 194
55	76	128	3	2	1	256	38	425.85	1 663 488	16.635	17.463	79 394
56	38	256	1	1	0	128	38	47.32	184 832	1.848	2.103	9 528
57	38	128	3	1	1	256	38	425.85	1 663 488	16.635	17.089	79 359
58	38	256	1	1	0	384	38	141.95	554 496	5.545	5.883	28 475
59	38	384	3	1	1	512	38	2555.12	9 980 928	99.809	102.48	475 871
60	38	512	1	1	0	561	38	414.76	1 620 168	16.202	17.04	83 058
61	38	384	3	2	1	512	19	638.78	2 495 232	24.952	27.645	118 505
62	19	512	1	1	0	256	19	47.32	184 832	1.848	2.147	9 521
63	19	256	3	1	1	512	19	425.85	1 663 488	16.635	18.233	79 558
64	19	512	1	1	0	384	19	70.98	277 248	2.772	3.158	14 261
65	19	384	3	1	1	512	19	638.78	2 495 232	24.952	27.361	118 046
66	19	512	1	1	0	561	19	103.69	405 042	4.050	4.547	20 805

As shown in table 5.2, each layer the accelerator timings have a slight discrepancy with the theoretical ones. This is due to the data transfer times not being accounted for in the theoretical times calculations.

The total execution time of the model, based on the run times of each layer, can be found on the table 5.3. It is noteworthy that this execution time is only an estimate of the real network total runtime, since the upsampling and FM layers were not implemented and, if implemented solely on software, these layers would have a significant impact on the models' performance.

Table 5.3: Total execution times.

Theoretical HW Time (s)	Real HW Time (s)	SW Only Time (s)
1.05	1.13	7710.77

To further compare the system results, the network model was ran and tested in other platforms. In addition to the software-only solution run in the Arm Cortex-A53 present on the ZCU04 development kit, the model was also tested in with a GPU-only execution, using a NVIDIA GeForce RTX 3090, and a CPU-only execution using an Intel(R) Core(TM) i7-11700F. These tests were done using the Pytorch floating point model and measuring its overall execution time. The execution of the model in the GPU and CPU was done by running the model in inference mode for 453 images from the UCAS-AOD dataset. The performance comparison alongside power consumption of these different systems can be found in table 5.4. The power consumption values for the ZCU104 were obtaned via the Vivado power report, with an additional estimated $3\,\mathrm{W}$ accounted for external memory and component access. The power consumption figures for the CPU and GPU were based on the manufacturer's maximum specified power usage

Table 5.4: Performance and efficiency of different platforms

Device	Time (s)	FPS	Power (W)
ZCU104	1.13	0.88	7.27
ARM Cortex-A53	7710.77	1.2×10^{-3}	5.63
Intel(R) Core(TM) i7-11700F	0.15	6.67	65
NVIDIA GeForce RTX 3090	0.0082	122.46	350

The highest throughput is achieved with the GPU which achieves 122 FPS. Both the GPU and the Intel processor are faster but require high power. The proposed solution in hardware achieves a frame rate close to 1 FPS with only 7.3 W. As stated above, the parallelism of the architecture using the same FPGA can be increased by a factor of 6 or 7, permitting increasing the throughput of the accelerator to close to 6 FPS.

Almost 50% of the power consumed by the FPGA is by the ARM core. With the complete model implemented in hardware, the processor is only used to configure the accelerator and the DMA. Using a small soft processor for these tasks implemented in the programmable logic would permit using an FPGA with only programmable logic and, thus, reduce the power.

This makes the hardware/software solution a far superior choice for real-time, on-site object detection in aerial images. Although it may be less performant, it boasts significantly lower power consumption compared to GPU and CPU platforms.

5.3 Conclusions

The accelerator was mapped in the programmable logic of the ZYNQ FPGA and integrated with ARM processor. The transfer of weights and feature maps between the external memory and the core is done through one DMA configured in the programmable logic.

The results reveal that the most used resource is memory. The free available DSPs and LUTs permit to increase the parallelism of the accelerator and consequently increase the throughput of the system. With a throughput of around 1 FPS and 7.3 W, the developed hardware/software system is slower than a GPU but uses $48\times$ lower power, making it a more suitable solution for edge computing appilications.

Chapter 6

Conclusion

This thesis presents the development and implementation of a hardware/software solution for real-time object detection in aerial images using a Convolutional Neural Network (CNN).

The R-YOLOv4 model was selected as the most suitable for this project due to its balance between accuracy and performance, as well as its support for oriented object detection. The model was modified to improve its suitability for hardware deployment. These modifications decreased the depth of the network and reduced the number of parameters, thereby lowering the computational and memory requirements at the cost of $2.5\,\%$ of accuracy. The model was quantized with 8 bits for both activations and weights, with a final mAP50 of 82.9%, 5.7% lower than the original model.

A custom hardware accelerator was designed and implemented in a Zynq Ultrascale+ MPSoc FPGA XCZU7E, with a total of 256 MACs that run in parallel.

Experimental results demonstrated that the proposed solution achieves a frame rate of nearly 1 FPS with a power consumption of only 7.3 W. Although this performance is lower compared to high-end GPU and CPU platforms, the significant reduction in power consumption makes it an attractive solution for edge computing applications. It is worth noting that the upsampling layers and feature map sum layers were not implemented in hardware.

The implementation details and performance evaluations indicate that the hardware/software solution offers a viable alternative for embedded object detection.

6.1 Future Work

As future work, the model can be fully implemented in hardware, including the upsampling and the sum layers. The sum layer requires an additional adder in each core, while the upsampling layer just requires additional memory to temporarily store the output activations before being upsampled. Therefore, the impact on the number of resources is minimal.

The sum layers can be implemented including a second AXI-Stream interface in the accelerator to receive the map to be summed to the output of the convolutional layer before being streamed out. The second AXI-Stream would be connected to another DMA and High-Performance port of the FPGA to

access the external memory controller.

The upsampling operation replicates each activation of the map into a 2×2 square. To implement it in hardware, an additional buffer should be added to the IP to store a fully processed output line from the IP. While processing the next line, the buffered line will be streamed out twice, with each activation in the line also being streamed twice. Additionally, configurations with more PEs and more MACs per PE can be implemented and tested to utilize the FPGAs available resources to achieve a higher throughput.

Bibliography

[1] V. Reilly, H. Idrees, and M. Shah, "Detection and Tracking of Large Number of Targets in Wide Area Surveillance," in *Computer Vision – ECCV 2010*, ser. Lecture Notes in Computer Science, K. Daniilidis, P. Maragos, and N. Paragios, Eds. Berlin, Heidelberg: Springer, 2010, pp. 186–199.

[2] 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*, 2020, vol. 943, arXiv:1910.13796 [cs]. [Online]. Available: http://arxiv.org/abs/1910.13796

[3] U. Nepal and H. Eslamiat, "Comparing YOLOv3, YOLOv4 and YOLOv5 for Autonomous Landing Spot Detection in Faulty UAVs," *Sensors*, vol. 22, no. 2, p. 464, Jan. 2022. [Online]. Available: https://www.mdpi.com/1424-8220/22/2/464

[4] Y. Tamaazousti, "On The Universality of Visual and Multimodal Representations," Ph.D. dissertation, Jun. 2018.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," Dec. 2015, arXiv:1512.03385 [cs]. [Online]. Available: http://arxiv.org/abs/1512.03385

[6] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," Mar. 2015, arXiv:1502.03167 [cs]. [Online]. Available: http://arxiv.org/abs/1502.03167

[7] Z.-Q. Zhao, P. Zheng, S.-t. Xu, and X. Wu, "Object Detection with Deep Learning: A Review," Apr. 2019, arXiv:1807.05511 [cs]. [Online]. Available: http://arxiv.org/abs/1807.05511

[8] G. Cheng, X. Yuan, X. Yao, K. Yan, Q. Zeng, and J. Han, "Towards Large-Scale Small Object Detection: Survey and Benchmarks," Dec. 2022, arXiv:2207.14096 [cs]. [Online]. Available: http://arxiv.org/abs/2207.14096

[9] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," Nov. 2013. [Online]. Available: https://arxiv.org/abs/1311.2524v5

[10] R. Girshick, "Fast R-CNN," Sep. 2015, arXiv:1504.08083 [cs]. [Online]. Available: http://arxiv.org/abs/1504.08083

[11] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” Jan. 2016, arXiv:1506.01497 [cs]. [Online]. Available: http://arxiv.org/abs/1506.01497

[12] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” May 2016, arXiv:1506.02640 [cs]. [Online]. Available: http://arxiv.org/abs/1506.02640

[13] J. Redmon and A. Farhadi, “YOLO9000: Better, Faster, Stronger,” in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Honolulu, HI: IEEE, Jul. 2017, pp. 6517–6525. [Online]. Available: http://ieeexplore.ieee.org/document/8100173/

[14] ——, “YOLOv3: An Incremental Improvement,” Apr. 2018, arXiv:1804.02767 [cs]. [Online]. Available: http://arxiv.org/abs/1804.02767

[15] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, “YOLOv4: Optimal Speed and Accuracy of Object Detection,” Apr. 2020, arXiv:2004.10934 [cs, eess]. [Online]. Available: http://arxiv.org/abs/2004.10934

[16] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, “SSD: Single Shot MultiBox Detector,” 2016, vol. 9905, pp. 21–37, arXiv:1512.02325 [cs]. [Online]. Available: http://arxiv.org/abs/1512.02325

[17] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, “Focal Loss for Dense Object Detection,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 2, pp. 318–327, Feb. 2020, conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[18] M. G. Selvaraj, A. Vergara, H. Ruiz, N. Safari, S. Elayabalan, W. Ocimati, and G. Blomme, “AI-powered banana diseases and pest detection,” *Plant Methods*, vol. 15, no. 1, p. 92, Dec. 2019. [Online]. Available: https://plantmethods.biomedcentral.com/articles/10.1186/s13007-019-0475-z

[19] C.-Y. Wang, H.-Y. M. Liao, I.-H. Yeh, Y.-H. Wu, P.-Y. Chen, and J.-W. Hsieh, “CSPNet: A New Backbone that can Enhance Learning Capability of CNN,” Nov. 2019, arXiv:1911.11929 [cs]. [Online]. Available: http://arxiv.org/abs/1911.11929

[20] “kunnnnethan/R-YOLOv4.” [Online]. Available: https://github.com/kunnnnethan/R-YOLOv4

[21] X. Yang, J. Yan, Z. Feng, and T. He, “R3Det: Refined Single-Stage Detector with Feature Refinement for Rotating Object,” Dec. 2020, arXiv:1908.05612 [cs, eess]. [Online]. Available: http://arxiv.org/abs/1908.05612

[22] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, “Microsoft COCO: Common Objects in Context,” Feb. 2015, arXiv:1405.0312 [cs]. [Online]. Available: http://arxiv.org/abs/1405.0312

[23] H. Zhu, X. Chen, W. Dai, K. Fu, Q. Ye, and J. Jiao, “Orientation robust object detection in aerial images using deep convolutional neural network,” in *2015 IEEE International Conference on Image Processing (ICIP)*, Sep. 2015, pp. 3735–3739.

[24] Z. Dong, M. Wang, Y. Wang, Y. Liu, Y. Feng, and W. Xu, “Multi-Oriented Object Detection in High-Resolution Remote Sensing Imagery Based on Convolutional Neural Networks with Adaptive Object Orientation Features,” *Remote Sensing*, vol. 14, no. 4, p. 950, Jan. 2022, number: 4 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2072-4292/14/4/950

[25] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, “Accelerating CNN inference on FPGAs: A Survey,” May 2018. [Online]. Available: https://arxiv.org/abs/1806.01683v1

[26] M. Reis, “Hardware acceleration of cnn-based image segmentation for fire detection,” Nov. 2022.