Hardware Acceleration of CNN-Based Image Segmentation for Fire Detection

Miguel Ângelo Ferreira da Paixão dos Reis

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: 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. Paulo Ferreira Godinho Flores

November 2022

Declaration

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

Acknowledgments

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

I would also like to thank INESC-ID and Fundação para a Ciência e a Tecnologia (FCT) for allocating a research grant, with reference BI| 2022/246, for me to be able to study the development of hardware design for image recognition using neural networks. I also acknowledge project IPL/2021/smartSPACE ISEL through Instituto Politécnico de Lisboa for providing training and design equipment.

I would like to offer my special thanks to Catarina Silva for always being present and for her patience throughout this journey.

I would also like to thank my family for their support in life, for giving everything that I need, for believing in my capabilities and helping me achieve even my most ambitious goals. Finally, I would like to thank my friends for sharing this journey with me, and the encouragement that they gave me.

Abstract

Image segmentation is an important task for many applications, like surveillance, security, medical, among others. Many of these applications need to be executed on edge devices to take real-time decisions. Hence, low cost and fast solutions are needed for image segmentation. One such application is automatic fire detection systems to facilitate the intervention and reduce the cost of firefighters travel in case of false occurrences.

This work optimizes an image segmentation algorithm for fire detection based on a deep learning model (Deeplabv3+ with a Modified Aligned Xception), designs a hardware accelerator for the algorithm and implements it in a System-on-Chip Field Programmable Gate Array (Xilinx Zynq UltraScale+ ZU3EG SoC).

Fixed-point quantization of both weights and activations, and batch normalization folding were applied to the original floating-point model followed by fine-tuning to improve the hardware performance without sacrificing accuracy.

The accelerator with a quantization of 4-bits was implemented in the FPGA and integrated with the processor of the SoC FPGA. The results show that the system has a mIoU accuracy of 92.76%, better than the accuracy of the original model (91.53%), and a throughput of 1 frame per second with a power of 5.8 W, which fulfills the throughput requirements of 0.1 FPS by 10 times, with a low power consumption.

Keywords

Convolutional Neural Network; Image Segmentation; Fire Detection; Quantization; Hardware Accelerator; High-Level Synthesis.

Resumo

A segmentação de imagens é uma tarefa importante para muitas aplicações, como vigilância, segurança, médica, entre outras. Muitas dessas aplicações precisam de ser executadas em dispositivos on-the-edge para tomar decisões em tempo real. Assim, soluções rápidas e de baixo custo são necessárias para segmentação de imagens. Uma dessas aplicações são os sistemas automáticos de detecção de incêndio para facilitar a intervenção e reduzir o custo de deslocamento dos bombeiros em caso de ocorrências falsas.

Este trabalho otimiza um algoritmo de segmentação de imagens para deteção de incêndios baseado em um modelo de aprendizagem profunda (Deeplabv3+ com Modified Aligned Xception), projeta um acelerador de hardware para executar o algoritmo e implementa-o em um dispositivo de lógica reconfigurável (Xilinx Zynq UltraScale+ ZU3EG SoC FPGA).

Os pesos e as ativações do modelo original foram quantizados com representações em vírgula fixa e os parâmetros de normalização foram integrados com os pesos. O modelo transformado foi de novo treinado. Este processo melhorar o desempenho do hardware sem sacrificar a precisão.

O acelerador com uma quantização de 4 bits foi implementado na FPGA e integrado com o processador da FPGA. Os resultados mostram que o sistema possui uma precisão mIoU de 92,76%, melhor que a precisão do modelo original (91,53%) e uma taxa de processamento de 1 imagem por segundo com uma potência de 5,8 W, cumprindo os requisitos de 0,1 FPS por 10 vezes com um baixo consumo de potência.

Palavras Chave

Rede Neuronal Convolucional; Segmentação de Imagem; Deteção de Incêndios; Quantização; Acelerador de Hardware; Síntese de Alto-Nivel;

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Objectives . . . 1

1.3 Outline . . . 2

2 State of the Art **3**

2.1 Convolutional Neural Networks . . . 3

2.1.1 Convolutional Layer . . . 4

2.1.2 Depthwise Separable Convolution . . . 6

2.1.3 Atrous Convolution . . . 7

2.1.4 Pooling Layer . . . 8

2.1.5 Activation Functions . . . 9

2.2 Image Classification . . . 9

2.2.1 LeNet . . . 9

2.2.2 AlexNet . . . 10

2.2.3 ResNet . . . 10

2.2.4 Xception . . . 11

2.2.5 Model Comparison . . . 12

2.3 Semantic Segmentation . . . 12

2.3.1 DeepLabV3+ . . . 14

2.4 CNNs in Field Programmable Gate Arrays . . . 14

2.4.1 Generic Convolution Algorithm . . . 15

2.5 Processing Flow Approaches . . . 17

2.6 Conclusion . . . 19

3 Design and Optimization of the Network Model **21**

3.1 Design Flow from Model to FPGA . . . 21

3.2 Tools for Training and Quantization . . . 22

3.2.1 Pytorch . . . 22

3.2.2 Brevitas . . . 23
3.3 Helper Tools . . . 23
3.3.1 Brevitas Converter . . . 24
3.3.2 Batch Normalization Merger . . . 26
3.3.3 Weights Extractor . . . 27
3.4 Network Description . . . 28
3.4.1 Corsican Dataset . . . 34
3.5 Work flow . . . 35

4 Hardware Accelerator for DeepLabV3+ 37
4.1 Hardware Design . . . 38
4.1.1 Convolution IP . . . 40
4.1.2 Depthwise Convolution IP . . . 41
4.1.3 Pre-Processing IP . . . 42
4.1.4 Post-Processing IP . . . 42
4.2 FPGA Image Build . . . 44
4.3 Baremetal Deployment . . . 47

5 Results 51
5.1 Pytorch Training . . . 51
5.2 Quantization . . . 52
5.3 Vitis HLS Results . . . 53
5.3.1 Pipeline Characteristics . . . 53
5.3.2 HLS Resource Estimate . . . 54
5.3.3 BRAM Sizing . . . 54
5.4 FPGA Build . . . 56
5.5 System Results . . . 56

6 Conclusion 63
6.1 Future Work . . . 63

Bibliography 65

List of Figures

2.1 LeNet Network from [1] . . . 4
2.2 Example of a 2D Convolution . . . 5
2.3 Example of a single iteration of a 3D Convolution . . . 6
2.4 Example of a Depthwise Separable Convolution . . . 7
2.5 Example of a Atrous Convolution . . . 8
2.6 Example of a Pooling computation . . . 8
2.7 Activation Functions . . . 9
2.8 AlexNet network from [2] . . . 10
2.9 Resnet-34 Network from [3] . . . 11
2.10 Xception Network from [4] . . . 11
2.11 Example of an output of an Image Segmentation CNN [5] . . . 13
2.12 DeepLabV3+ Network . . . 14
2.13 Typical FPGA-based CNN accelerator based on [6] . . . 15
2.14 Configurable Layer Processor Accelerator . . . 17
2.15 Pipelined Accelerator . . . 17

3.1 Design Flow . . . 22
3.2 Convolution, Batch Normalization and ReLu layers definition in Pytorch . . . 23
3.3 Quantization Engine declaration . . . 24
3.4 setBitWitdth Function . . . 25
3.5 Input LeNet Network . . . 25
3.6 Output of Brevitas Converter on the Lenet Network . . . 26
3.7 Example of variable bitwidth weights packed into a 64-Bit Word . . . 27
3.8 DeepLabV3+ . . . 28
3.9 Modified Aligned Xception . . . 29
3.10 Example images of the Corsican Dataset . . . 34

4.1 IP Topology 38
4.2 Filter and Bias data example 39
4.3 Feature Map data example 39
4.4 Convolution IP 40
4.5 Depthwise IP 41
4.6 Post Processing IP Flow Chart 44
4.7 Vivado Block Design 46
4.8 Atrous Convolution with dilation rate of 2 with partitioned FMs 48

5.1 Example of input (left) and corresponding output (right) of the neural network 52

List of Tables

2.1 Comparison between models on the ImageNet dataset with input size of 299×299 12
2.2 Error for different fixed point data sizes 17
2.3 Hardware resources and Performance Comparison between Angel-Eye and MALOC from [7] and [8] 18

3.1 DeepLabV3+ with Xception Backbone Network Description 31

5.1 Test mIoU(%) of Quantized Network with varying bitwidths 52
5.2 Pipeline characteristics of the Convolution IP 53
5.3 Pipeline characteristics of the Depthwise Convolution IP 53
5.4 Pipeline characteristics of the Pre-Processing IP 53
5.5 Pipeline characteristics of the Post-Processing IP 54
5.6 Vitis HLS Resource Estimates 54
5.7 Biggest FM, Weights and Bias values of the network 55
5.8 Number of BRAMs for FM for each IP 55
5.9 Number of BRAMs for Bias for each IP 55
5.10 Number of BRAMs for Weights for each IP 55
5.11 Total Number of Brams Needed for each IP 56
5.12 Vivado Implementation Resources 56
5.13 Timing Results per Layer 58
5.14 Timing Summary for various types of convolutions flows 61
5.15 Throughput, Power and Cost Comparison of several Solutions 62

List of Algorithms

2.1 3D Convolution 16

Acronyms

ASPP	Atrous Spatial Pyramid Pooling
CLP	Configurable Layer Processor
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DMA	Direct Memory Access
DNN	Deep Neural Network
FM	Feature Map
FPS	Frames Per Second
FC	Fully Connected
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
IFM	Input Feature Map
IoT	Internet-of-Things
IP	Intellectual Property
LSB	Least Significant Byte
MAC	Multiply and Accumulate
mIoU	mean Intersection over Union
MSB	Most Significant Byte
OFM	Output Feature Map
PE	Processing Element
PL	Programmable Logic
PS	Programming System
ReLU	Rectified Linear Unit

RGB Red-Green-Blue

SoC System-on-Chip

1 Introduction

The goal of this work is to develop a fire detection deep neural network embedded system using a low-cost System-on-Chip (SoC) Field Programmable Gate Array (FPGA) to be used on Internet-of-Things (IoT) Devices. This chapter describes the motivation, the main objectives for this work and the report outline.

1.1 Motivation

Due to the increase of forest fires in the last decade, the purpose chosen for this work is the early detection of fires in order to decrease the damage of these disasters. Image processing in deep neural networks has been continuously evolving throughout the years. Due to the computation-intensive nature of deep neural networks, they are predominantly executed with the assistance of a Central Processing Unit (CPU) or a Graphics Processing Unit (GPU) in real-world applications, which are costly in power consumption and money wise. Running them on edge devices is an improvement regarding cost and power consumption but it is a challenge due to the limited resources attached to them. FPGAs can be used as on-the-edge devices due to their low cost and possible low-power consumption but several tradeoffs must be done in order to run CNNs on them. Speed, accuracy and throughput are some of the compromises that must be made in order for the system to run in a low-cost FPGA. For detecting fires a minimal throughput of 0.1 Frames Per Second (FPS) with a low power consumption should be achieved.

1.2 Objectives

The main objective of this work is to propose a methodology to develop a real-time fire-detecting FPGA-based Convolutional Neural Network. This implies studying the concept of Convolutional Neural Net-

works and the state of the art of FPGA-based Convolutional Neural Networks. Afterwards, the training of the Convolutional Neural Network is performed on a separate machine with a powerful GPU since the training takes a considerable amount of time while the inference must be done on-site by an edge device like an FPGA in order to reduce communication latency. The quantization of the trained model will then be performed in order to reach the smallest data size without compromising the accuracy of the model. The hardware acceleration will also be designed along with the software intended for network inference. The embedded system will then be tested and validated and any fine-tuning needed will also be performed.

1.3 Outline

This report is organized as follows: Chapter 2 reviews the State of the Art where convolutional neural networks are introduced along with the most relevant examples of known networks and describes the most common implementation methods on the FPGAs. Two types of architectures are also introduced, the all-purpose accelerator architecture where all convolutional layers can be computed in the same convolutional layer processor and the pipelined accelerator architecture where the whole Convolutional Neural Network (CNN) model is mapped into the FPGAs hardware. Chapter 3 details the design and optimization of the deep learning model, as well as the application to be developedand analyzes the initial results with training and quantization. Chapter 4 presents the hardware accelerator to run the model. Chapter 5 presents the results of both model optimization, hardware design and full-system execution. Chapter 6 concludes with the developed work and proposes future work.

2

State of the Art

This chapter introduces the concept of Convolutional Neural Networks (CNN) and their use in image classification and image segmentation along with the most relevant models that have been proposed. Implementation methods on FPGAs are also introduced along with two architecture types.

2.1 Convolutional Neural Networks

CNNs are a category of Deep Neural Network (DNN) mostly suited for image-focused tasks such as image classification or image recognition. CNNs work by processing input images in the form of matrices with pre-defined dimensions, for example, an Red-Green-Blue (RGB) image would have three channels and a black and white image would only have one, and getting an output that can be later turned into a prediction. In order for a CNN to be able to output a desirable prediction it needs to be trained over several iterations with the help of already classified datasets. These datasets consist of an agglomerate of images, with several classes, each with its corresponding truth label. A well-trained CNN will output correct predictions of the labels associated with the input images that are processed.

CNNs are made up of several layers, where each one gets its input Feature Map (FM) from the previous layer, and outputs its resulting FM to the next one. Feature maps are the input and output of each layer, composed of two-dimensional arrays of pre-determined sizes. The number of filters used in a layer will result in the same number of output FMs. There are four types of layers which together make up a model of a CNN: the convolutional layers, the pooling layers, the activation function layers, and the fully connected layers. Figure 2.1 shows an example of a basic CNN LeNet proposed in [1] where the mentioned layers can be seen.

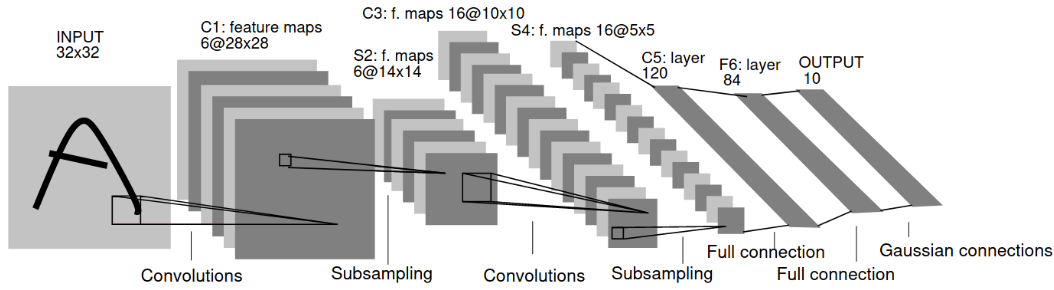

Figure 2.1: LeNet Network from [1]

Some CNNs also include a Batch Normalization layer to normalize the average and the standard deviation of the layers output FM to zero and one, respectively [9]. This improves overall accuracy and speeds up the training process.

2.1.1 Convolutional Layer

The convolutional layer is the most important component of a DNN, it is responsible for extracting features out of its input FM. This is done by convoluting the input FM with a predefined kernel. A kernel is a small two-dimensional array composed of weights that are trained in order to achieve the best predictions possible. A 2D convolution is done by processing a single kernel with its input FM with also only one two-dimensional array. The kernel is overlaid with the input FM array and the dot-product is performed: the multiplications of each overlapped value are added together and then stored on a position of the output array. A bias value is also added to the result which is also a trained parameter. The process is repeated one position to the right until the output array is filled. There are two parameters which alter the way the convolution is done: the padding and the stride. The stride dictates how many positions the kernel shifts after each dot-product and the padding determines how many 0-valued positions are added to the edges of the input array. The size of the output FM is calculated with Equation (2.1) [10] where i_h and i_w are the input FM array height and width, the k_h and k_w are the kernel height and width, the s is the stride and the p is the padding.

$$Output\ FM\ array\ size = (\frac{i_h - k_h + 2p}{s} + 1) * (\frac{i_w - k_w + 2p}{s} + 1) = OFM_h * OFM_w \tag{2.1}$$

The number of total multiplications in a 2D convolution can be calculated using equation eq. (2.2) where Output Feature Map (OFM) is the output FM and k is the kernel.

$$\#ofMultiplications = OFM_h * OFM_w * k_h * k_w \tag{2.2}$$

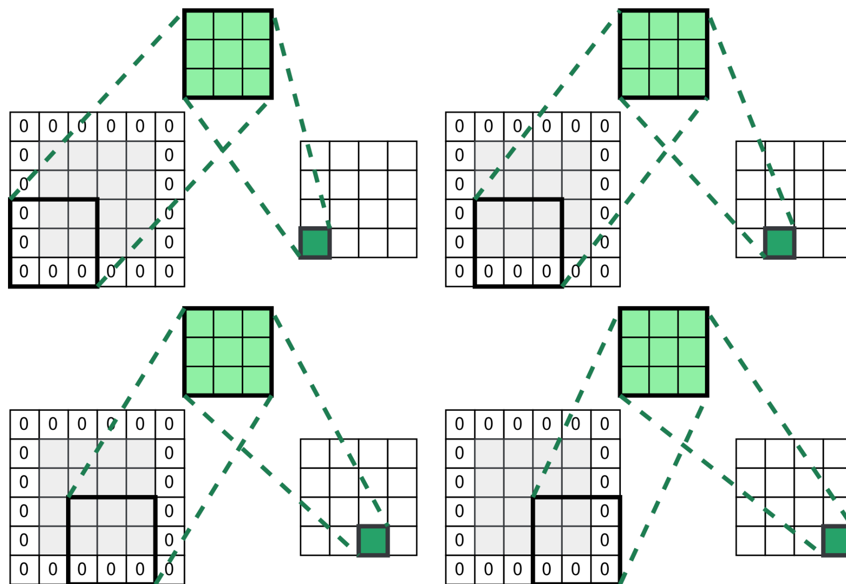

Figure 2.2: Example of a 2D Convolution

Figure 2.2 shows an example of a 2D convolution, showing only 4 dot-products for simplicity, with an input FM array size of 4×4, a kernel with a size of 3×3 and a stride and padding of 1. The kernel is represented by the light-green box with several weights represented by its internal squares. The input and output FMs are also represented on the left and right, respectively, of each dot-product. In each iteration, the dot-product result is represented by the dark-green squares on the output FM. After each dot-product, except the last one, a one-position shift to the right is shown. The output FM array has a size of 4×4 that is confirmed by equation eq. (2.1).

For a 3D convolution, the output FM is comprised of several two-dimensional arrays, or channels, the same number as the number of the filter used. A filter is composed of several channels, comparable to a 3D kernel. As in the 2D convolution, several dot-products are performed with the same channels (e.g the first filter channel is only computed with the first input FM channel, the second filter channel with the second input FM channel, and so forth) and the results of these dot-products are added together along with the bias and stored in a position of a channel of the OFM. Each filter used will result in a different channel on the OFM.

Figure 2.3 shows an example of a 3D convolution of a 3 channel input FM with 2 filters.

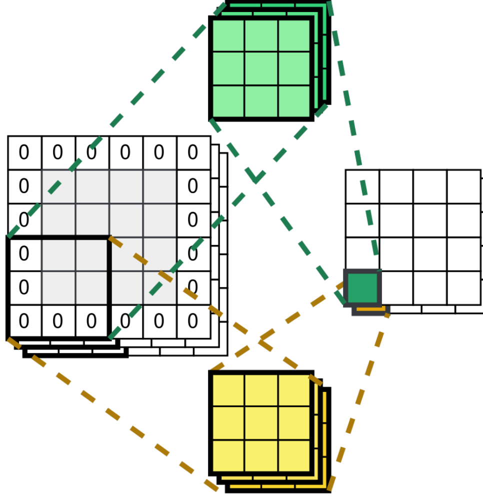

Figure 2.3: Example of a single iteration of a 3D Convolution

An input FM with 3 channels can be seen with two filters, shown with two different colours, with also 3 channels and an output FM of 2 channels, the same number of channels as the number of filters. Only one iteration of the convolution is shown, for simplicity, where each filter is related to a single channel of the output FM.

The number of total multiplications in a 3D convolution can be calculated using equation eq. (2.3) where OFM is the output FM, $N_{filters}$ is the number of filters and F is the Filter.

$$\#of Multiplications = OFM_h * OFM_w * N_{filters} * F_{channels} * F_h * F_w \tag{2.3}$$

2.1.2 Depthwise Separable Convolution

Depthwise separable convolutions are a less computationally expensive way of doing 3D convolutions with a large number of filters [11]. The depthwise separable convolution works by using channel-independent dot-products, much like several 2D convolutions, resulting in an intermediate FM with the same number of channels as the filter used, and then doing a 3D convolution with several filters with 1×1 sized channels with the intermediate FM of the previous convolution. Like the 3D convolution, the output FM has the same number of channels as the number of filters used in the depthwise convolution.

The number of total multiplications in a depthwise separable convolution can be calculated using equation eq. (2.4) where OFM is the output FM, IFM is the output FM.

$$\#of Multiplications = OFM_h * OFM_w * F_h * F_w * IFM_{channels} + N_{filters} * IFM_{channels} * OFM_h * OFM_w \tag{2.4}$$

While a 3D convolution of a 3-channel 12×12 Input Feature Map (IFM) with 256 filters sized 5×5 that

results in a 256 channel 8×8 output FM requires a total of 1,228,800 multiplications, a depthwise separable convolution needs only 4,800 multiplications for the first convolution and then 49,152 multiplications for the depthwise convolution for a total of 53,952 multiplications.

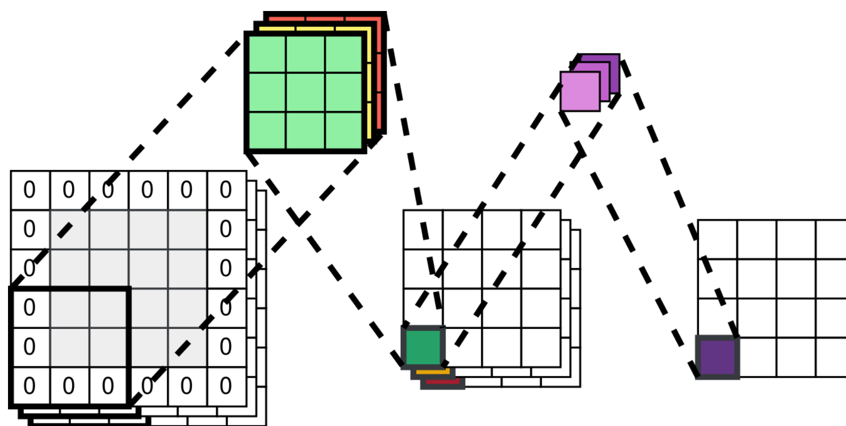

Figure 2.4: Example of a Depthwise Separable Convolution

Figure 2.4 shows an example of a depthwise separable convolution of a 3 channel 4×4 image with one filter with 3 channels for the channel independent convolutions and one 1×1 3 channel filter for the depthwise convolutions. Figure 2.4 shows one iteration of a depthwise separable convolution, such that the intermediate FM, in the middle, has 3 channels where each coloured value is computed with a 2D convolution of each channel of the input FM with its respective filter channel. Afterwards, the output FM is calculated as a 3D convolution with a single 3 channel filter sized 1×1.

2.1.3 Atrous Convolution

Atrous convolutions allow the control of the resolution of the convolutions by adding a new parameter, the rate. The rate defines the number of spaces between the values of the kernel. This allows for the convolutions to have a wider field of view with the same kernel size.

The size of this convolution output is calculated with Equation (2.5) where i_h and i_w are the input height and width, the k_h and k_w are the kernel height and width, the s is stride, the p is the padding and r is the rate.

$$OutputSize = (\frac{i_h - (k_h * r - (r-1)) + 2p}{s} + 1) * (\frac{i_w - (k_h * r - (r-1)) + 2p}{s} + 1) = O_h * O_w \quad (2.5)$$

The number of multiplications of an Atrous Convolution is also given by eq. (2.3).

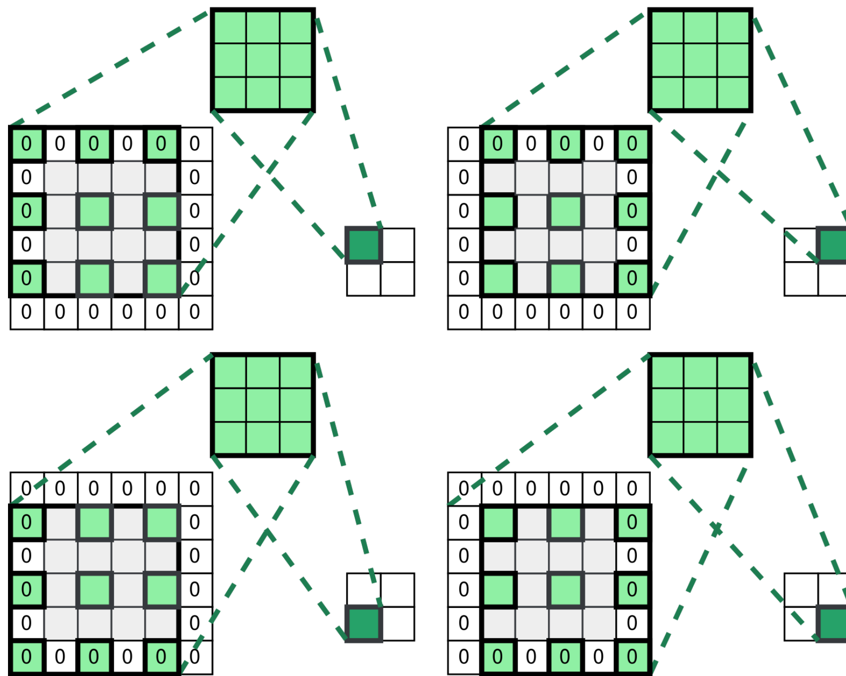

Figure 2.5: Example of a Atrous Convolution

Figure 2.5 shows an example of a full atrous convolution of a single channel 4×4 input FM, a kernel of 3x3 with a stride and padding of 1 and rate of 2 where four dot-products are performed where the values of the input FM are separated by one value, defined by the rate parameter of 2. The shift after each dot-product, except the last one, is defined by the stride parameter of 1.

2.1.4 Pooling Layer

Pooling layers are used to reduce the size of the input FM and by doing so they make the detected features more prevalent and reduce memory consumption. The most used type of pooling layer is max pooling which gets the maximum value of a window, with a predefined size, and stores it in an output FM with a reduced size.

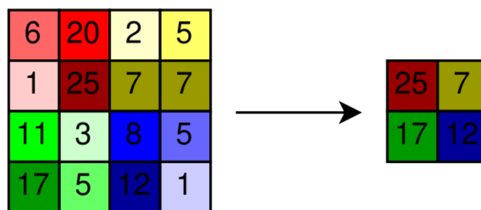

Figure 2.6: Example of a Pooling computation

Figure 2.6 shows a max pooling operation of 2 where the output FM is comprised of the maximum values of each 2×2 block of the input FM.

2.1.5 Activation Functions

The function layer simply passes each value of the input FM through a function. The most common activation functions are the Sigmoid Function and the Rectified Linear Unit (ReLU). The Sigmoid Function smooths the output using a logistic function while the ReLU function turns all negative values to 0 and keeps the positive values the same.

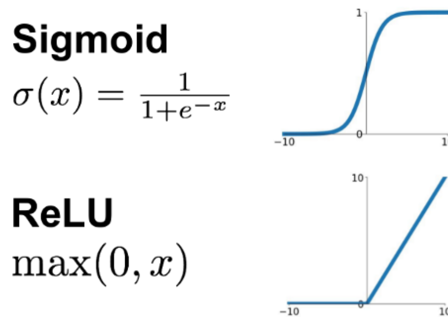

Figure 2.7: Activation Functions

Sketched in Figure 2.7 are both these activation functions where it can be seen that the ReLU only outputs positive values while the sigmoid, through its logistic function, allows for negative values.

2.2 Image Classification

Image classification works by attributing a percentage of certainty of prediction to several classes, the more probable one being the one with a higher value. The metrics to evaluate these models are the top-1 error and top-5 error, which indicate how many times the network has predicted the correct label with the highest probability and how many times the correct label appears in the network's top five predicted classes, respectively.

2.2.1 LeNet

The LeNet model [1], proposed in 1989, was one of the earliest CNN models used for image classification. This model, compared to newer models, is a simpler network containing a total of 6 layers: 2 convolutional layers and 2 average pooling layers, followed by 2 Fully Connected (FC) layers. It uses the Sigmoid function as the activation after each convolutional layer and the first fully connected layer. It was designed to classify 32×32 single-channel images of handwritten digits. Figure 2.1 shows the LeNet network.

2.2.2 AlexNet

The AlexNet [2] model added more layers, in comparison to the LeNet layer, uses ReLu activations layers instead of Sigmoid functions and uses max pooling layers instead of the average pooling layers. It is comprised of 11 layers in total: 5 convolutional, 3 max pooling layers and 3 fully connected layers. It takes as inputs three channel RGB images of size 227×227 and uses the ReLU as an activation function in the first, second and fifth convolutional layers. The AlexNet model won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012 [2].

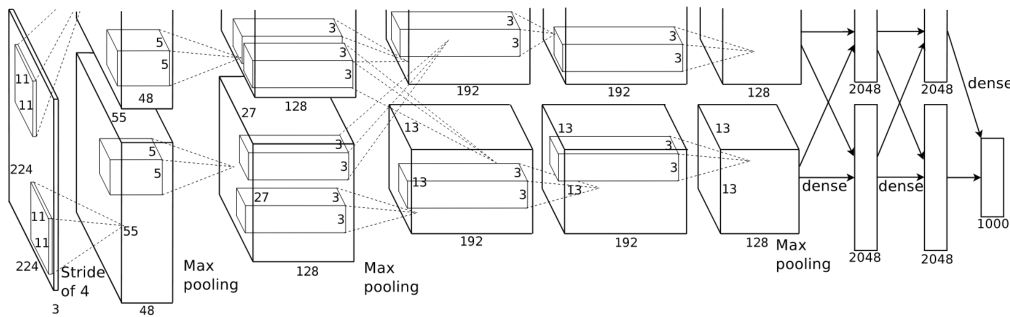

Figure 2.8: AlexNet network from [2]

Figure 2.8 shows the AlexNet network where the size of each FM can be visualized along with the filters size.

2.2.3 ResNet

Deeper networks tend to lose accuracy compared to more shallow networks due to the vanishing gradient problem. The vanishing gradient problem occurs when a DNN is unable to propagate useful gradient information to the initial layers, limiting the training of those layers and thus reducing their effect in inference. ResNet [3] solves this by implementing shortcut connections. Every few layers the output is added with the input of an earlier layer, diminishing the impact of having a large number of layers and thus solving the vanishing gradient problem. This network uses batch normalizations followed by ReLU after every convolution and ends with an average pooling layer followed by a fully connected layer. The ResNet model [3] won the ILSVRC in 2015 with its 152 layers variant.

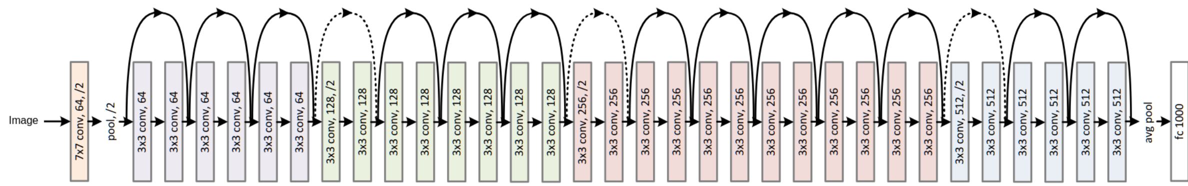

Figure 2.9: Resnet-34 Network from [3]

Figure 2.9 shows the Resnet-34 network, a 34 layers variation of the ResNet network, where, for most of the model, a shortcut connection is added in between 2 layers.

2.2.4 Xception

The Xception model [4] uses mostly separable depthwise convolutions, instead of the 3D convolutions used by the previous networks, followed by batch normalizations, ReLU Functions and/or max pooling layers. The use of depthwise separable convolutions was inspired by the use of Inception modules on the Inception V3 network. The typical Inception modules can be simplified as large 1×1 convolutions followed by convolutions that operate on non-overlapping segments of the output FM. It was found that the depthwise separable convolutions performed similarly with a similar number of weights. The Xception model also uses shortcut connections similar to ResNet. This model was able to achieve a top-1 accuracy of 79% on the ImageNet dataset, surpassing the 77% accuracy achieved by ResNet-152.

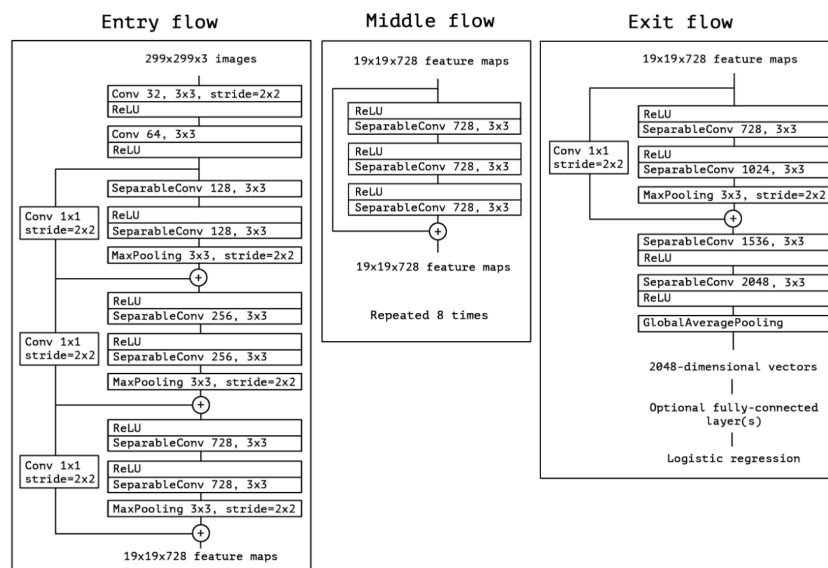

Figure 2.10: Xception Network from [4]

Figure 2.10 shows the Xception network where no 2D or 3D convolutions are used, except at the

beginning of the Entry flow, but instead, depthwise separable convolutions are used. Like the Resnet network, it also employs the use of shortcut connections.

2.2.5 Model Comparison

Table 2.1 shows a comparison between these models, showing the number of convolutional layers, their top-1 percentages on the ImageNet dataset and the number of Multiply-Accumulate (MAC) operations.

Table 2.1: Comparison between models on the ImageNet dataset with input size of 299×299

	LeNet	AlexNet	ResNet-34	ResNet-152	Xception
# Convolutional Layers	2	5	34	152	36
Top-1 Accuracy	-	63%	76%	77%	79%
# MAC Operations	424k	770M	3.68G	11.6G	4.58G

The table shows that with an increase of layers and consequently the increase of MAC operations the accuracy of a network can increase. However, as shown with the Xception network, that is not always true since it has only 36 layers.

2.3 Semantic Segmentation

Semantic segmentation is the process of assigning a label to each pixel of an input image. This is useful for object detection where several objects can be identified. A common architecture for semantic segmentation is the Encoder-Decoder structure [12, 13]. The Encoder-Decoder structure is a two-stage network where the encoder compresses its input through several convolutions, which captures the semantic information, while the decoder predicts the output from the encoder's output FM. The encoder is usually a network such as ResNet or Xception that captures the semantic information, through its several convolutions, while the decoder assigns each output pixels label using the encoder's semantic information.

Figure 2.11 shows an example of an output of a Semantic Segmentation CNN where it is shown that the motorcycle and the driver are separately identified with different colours, as they are different classes.

Figure 2.11: Example of an output of an Image Segmentation CNN [5]

The metrics to evaluate these networks are the accuracy which is the ratio of correct pixels over the total number of pixels of the image (eq. 2.6), the mean Intersection over Union (mIoU) which is the average of the correctly predicted pixels (TP) divided by the total number of pixels of the input image of each class (eq. 2.7), and the Dice coefficient which is the average of the ratio of the double of correctly predicted pixels (TP) over the sum of the double of the correctly predicted pixels with the rest of the pixels for each class (TP is True Positives, FN is False Negatives, FP is False Positives and N is the number of classes) shown in (eq. 2.8).

$$Accuracy = \frac{TP}{TP + FP + FN} \tag{2.6}$$

$$mIoU = \frac{1}{N}\Sigma\frac{TP}{TP + FP + FN} \tag{2.7}$$

$$DiceScore = \frac{1}{N}\Sigma\frac{2TP}{2TP + FP + FN} \tag{2.8}$$

For example, if the prediction of the network is 100 pixels of class B and the input image of 100 pixels has 5 pixels of class A and 95 pixels of class B, the calculated metrics would be an accuracy of $95/100 = 95\%$, a mIoU of $(0\% + 95\%)/2 = 47\%$ and a Dice coefficient of $(0\% + 97\%)/2 = 48.5\%$. For an image with a large background, like in figure 2.11 the accuracy alone is not a great indicator of whether the network is performing well or not due to the difference in density of the labels present i.e. 5 pixels of class A and 95 pixels of class B, so the mIoU and Dice coefficient may be used as better metrics.

DeConvNet [14], SegNet [15] and DeepLavV3+ [16] are some of the example models [12] used for Image Segmentation. DeepLabV3+ combines the use of Atrous Spatial Pyramid Pooling (ASPP) and the use of Deep CNNs such as ResNet or Xception to improve the localization of object boundaries, as explained in the following section.

2.3.1 DeepLabV3+

DeepLabV3+ [16] is an architecture that employs the Encoder-Decoder structure. A simple decoder is added to its predecessor DeepLabV3 which makes use of the ASPP. ASPP applies several atrous convolutions with different rates in parallel in order to compute features in multiple scales. ASPP can help with detecting objects that might be too far away or too close to the camera due to the use of atrous convolutions with different rates. DeepLabV3+ is the successor of DeepLabV3, which does not have a decoder. By adding a simple yet effective decoder, DeepLabV3+ is able to obtain sharper segmentations. The use of other networks as its encoder, or backbone, is possible. Using a Modified Xception as a backbone this model was able to obtain a mIoU of 89% while its predecessor DeepLabV3 was only able to achieve 86% on the JFT-300M dataset. The JFT-300M dataset is a proprietary dataset by Google that contains 300M Images with a total of over one billion labels.

Shown in figure 2.12 is a representation of the DeepLabV3+ network.

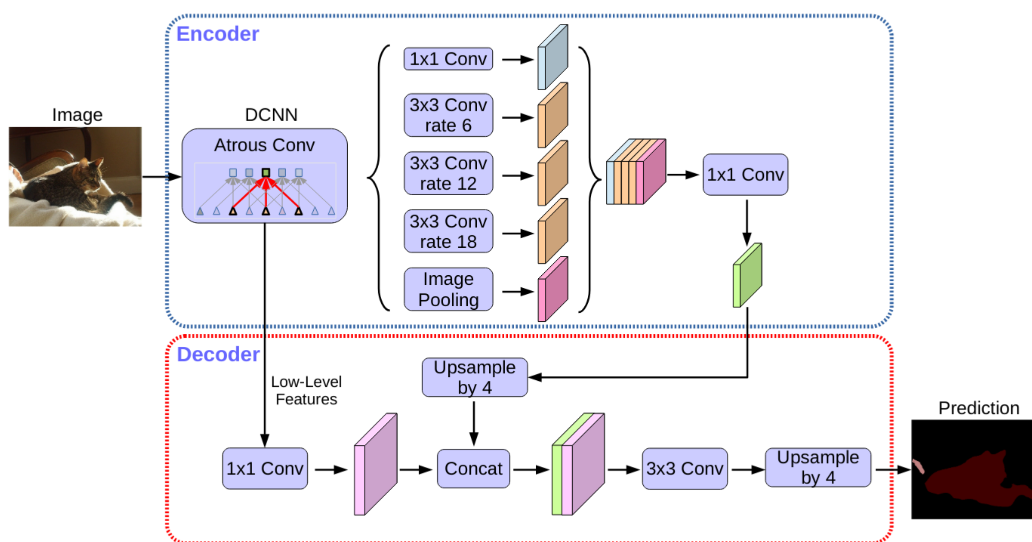

Figure 2.12: DeepLabV3+ Network

The above Figure is a representation of the V3+ network with two main blocks: the Encoder and the Decoder. The ASPP is visible on the right side of the Encoder.

2.4 CNNs in Field Programmable Gate Arrays

The computations involved in CNNs are highly intensive due to the high number of Multiply and Accumulate (MAC) operations (tab. 2.1). As CNN models get deeper the number of operations and the storage requirements increase. In order to efficiently use FPGAs, the CNNs can be compressed in order to

reduce the use of memory, communications and operations while taking into account the minimization of accuracy loss. Compression of a CNN can be done with quantization and pruning techniques.

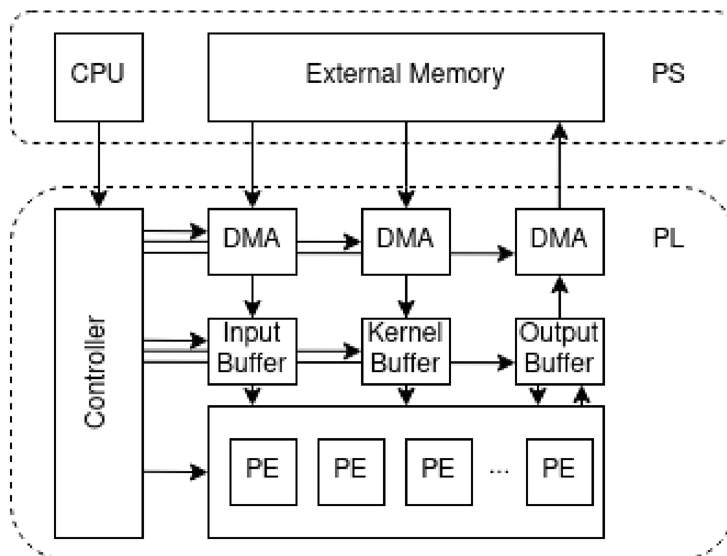

Figure 2.13: Typical FPGA-based CNN accelerator based on [6]

In typical FPGA-based CNN accelerators [6], as shown in Figure 2.13, there are several levels of memory hierarchy. These systems are composed of input buffers that fetch the input FMs and the kernel values from the external memory through Direct Memory Access (DMA). The input FMs are streamed into Processing Element (PE) in the Programmable Logic (PL) and on-chip memories are used for their capabilities of storing data locally. There are also on-chip output buffers to store the output FMs to be transferred back to the external memory.

The workload is sent to the controller from the CPU in the Programming System (PS) which then sends the control signals to the other modules. In order to decrease the critical path of the circuit and increase the throughput the computations are usually pipelined.

2.4.1 Generic Convolution Algorithm

Usually, more than 90% of the execution time of a CNN is spent processing the convolutional layers [6], therefore speedup is achieved by parallelizing these layers.

Algorithm 2.1 is a generic algorithm for a simple 3D convolution with N filters, each with FN kernels. OH and OW represent the size of the output FM in height and width respectively, and KW and KH represent the width and height of the kernel. The convolution process iterates through each Filter, processing each kernel with its corresponding input FM, applying the necessary MAC operations and storing the output FM in an output vector Y.

Algorithm 2.1: 3D Convolution

```
begin
    //Go through N Filters
    LOOP 1:for i in {0,...,N-1} do
        //Go through the width of the input FM
        LOOP 2:for j in {0,...,OW-1} do
            //Go through the height of the input FM
            LOOP 3:for k in {0,...,OH-1} do
                Y[i,j,k] = bias[i] //set initial output pixel to bias
                //Go through FN kernels of the filter
                LOOP 4:for v in {0,...,FN-1} do
                    //Go through the width of the kernel
                    LOOP 5:for u in {0,...,KW-1} do
                        //Go through the height of the kernel
                        LOOP 6:for c in {0,...,KH-1} do
                            Y[i,k,j] += X[v,k+c,j+u] * K[i,v,c,u]
```

Some of these loops may be unrolled in order to achieve parallelism, thus decreasing execution time. Unrolling a loop allows two or more iterations of a loop to be performed at the same time (an associated unroll factor determines how many iterations are executed in parallel). The following are the different kinds of achievable parallelism by loop unrolling, in algorithm 2.1.

Intra-Convolution Parallelism: multiplications in single-layer convolutions are implemented in parallel (LOOPS 5 and 6).

Inter-Convolution Parallelism: multiple layers convolutions implemented in parallel (LOOP 4).

Intra-Feature-Map Parallelism: multiple pixels of a single output channel are computed in parallel (LOOPS 2 and 3).

Inter-Feature-Map Parallelism: multiple output channels are computed in parallel (LOOP 1).

The overall number of computations can be reduced without significant accuracy losses by pruning. Pruning simply removes or zeroes the weights below certain thresholds. Three types of pruning are [17]: channel level pruning which removes all outgoing filters if the incoming kernels are also filters, kernel level pruning where a kernel is deleted from a filter and intra-kernel sparsity pruning which forces some weights into zero values ones. Further fine-tuning can be performed through successive iterations until a specified trade-off between accuracy and pruning objective is achieved. AlexNet is reduced by a factor of 9x with no significant accuracy losses using intra-kernel sparsity pruning [18].

Data can be quantized to any number of bits reducing the complexity of the hardware necessary for the computations and the storage capacity. It also can increase the throughput of the system but it may also decrease the overall accuracy of the model, being necessary to choose the best trade-off between accuracy and throughput and/or resource consumption. A fixed point number representation is usually chosen due to the less computationally expensive operations associated with it, in comparison

to floating point representation, and is characterized by its Bitwidth (BW), the number of bits used to store the number, and the Fractional Offset (FO) which is the number of fractional bits. Table 2.2 shows representations with different data sizes and their relative truncation error.

Table 2.2: Error for different fixed point data sizes

Original Value	BW	FO	Binary	True Value	Relative Error
3.838	3	0	011	3	21.8%
	4	1	011.1	3.5	08.8%
	5	2	011.11	3.75	02.3%
	6	3	011.110	3.75	02.3%
	7	4	011.1101	3.8125	00.7%
	8	5	011.11010	3.8125	00.7%
	9	6	011.110101	3.828125	00.3%

Table 2.2 shows that, for an example value of 3.838, the error decreases with the increase of the fractional bits,but the total BW increases. A minimum of 3 integer bits is necessary to represent the integer value with the sign bit.

Quantization can be performed after the model is trained by simply converting the floating point values into the chosen data size, or while training where the weights and activations are converted (during the forward pass), the latter gives better results due to the implicit fine-tuning.

2.5 Processing Flow Approaches

Two types of processing flow approaches can be used in order to run CNNs on FPGAs, one in which one or several Configurable Layer Processor (CLP) computes each layer iteratively, and another where the whole model computation is mapped on the FPGA and each layer is computed in sequence in a pipelined fashion.

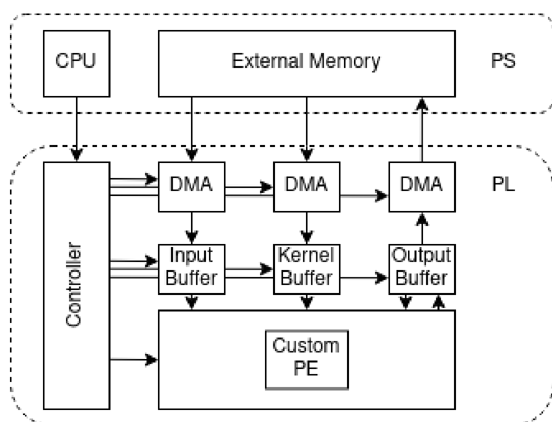

Figure 2.14: Configurable Layer Processor Accelerator

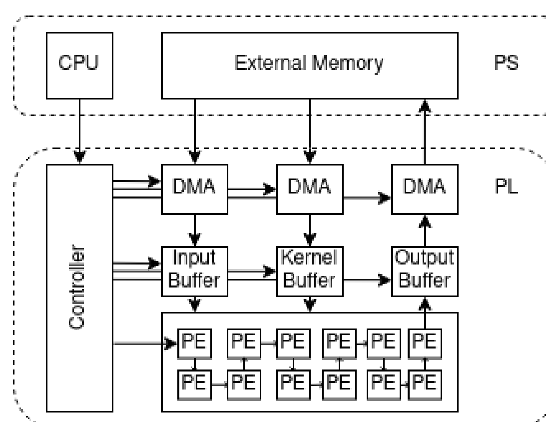

Figure 2.15: Pipelined Accelerator

Figures 2.14 and 2.15 show representations of accelerators following these two processing flow approaches. For the CLP accelerator, there is only one customizable CLP, where the convolution parameters, such as sizes, padding and stride, are sent from the CPU, and in the pipelined accelerator there is a chain of hardwired layer processor with fixed convolution parameters tied to each layer processor.

While pipelined accelerators might be faster due to the low memory transfers, they are constrained to a specific CNN model and they can be more hardware expensive than all-purpose CLPs because all the individual layers must be simultaneously mapped in the hardware.

Two examples of relevant works that use these two processing flows are Angel-Eye [7] which uses a CLP Processing flow and MALOC [8] which uses a pipelined layer processor processing flow.

The Angel-Eye [7] is a design flow that uses a run-time configurable accelerator on the VGG models, it uses instructions to describe each layer's parameters and the data transfers it has to perform. Angel-Eye employs Inter-Feature-Map parallelism, Intra-Convolution parallelism and Inter-Convolution parallelism. By fusing layers [19] several can be processed in the same iteration of the CLP thus decreasing the number of intermediate memory accesses. Higher efficiency is also achievable [20] with the use of two smaller CLPs than one bigger one due to possible unused hardware on the latter.

The MALOC [8] is a pipelined architecture where all intermediate results are passed through each PE resulting in less memory transfer. For the quantization, MALOC uses 16-bit fixed-point representation and it uses loop tiling as well as fused layers to decrease the amount of buffered data. It also prunes FC layers to decrease the total number of weights used without decreasing the accuracy. Another architecture is a systolic array [21] where a 2D array of PEs is laid out and the data is shifted between adjacent PEs removing the need for global interconnects. The model is mapped out into a feasible PE array configuration making use of the exploitation of data reuse.

Table 2.3: Hardware resources and Performance Comparison between Angel-Eye and MALOC from [7] and [8]

	LUTs	FFs	BRAMs	DSPs	Clock	Performance
Angel-Eye XC7Z045	182616	127653	486	780	150 MHz	187.8 GOP/s
MALOC VX690T	320203	309227	2365	2688	150 MHz	829.84 GOP/s

Since both of these works implement a VGG-16 network, a comparison can be done on hardware resources and performance. The Angel-Eye implementation was done on Xilinx Zynq-7045 FPGA while the MALOC implementation was done on Xilinx Virtex-690T FPGA. Both of these implementations used an 8-bit implementation. Table 2.3 shows this comparison and it is clear that the MALOC implementation requires more hardware resources than the Angel-Eye implementation but comes with a significantly higher performance than Angel-Eye.

2.6 Conclusion

This chapter showed how CNNs work along with an example of known CNNs and their distinct features. The methods of implementing CNNs in FPGAs along with examples of two different approaches on how to implement the accelerators was also shown. Quantization will be used in the next chapter along with a CLP processing flow to be used with a DeepLabV3+ network.

3

Design and Optimization of the Network Model

3.1 Design Flow from Model to FPGA

In order to deploy this neural network into an SoC FPGA several tasks must be performed, according to the design flow shown in Figure 3.1.

The network model has to be chosen along with the dataset to train the model with known tools such as Pytorch and Brevitas. After training, the trained model is evaluated to check the accuracy and to obtain checkpoints useful to verify the correctness of the outputs produced by the developed system.

A tool was developed, named Brevitas Converter, that converts the Pytorch network into a quantized Brevitas network along with the trained weights. Afterwards, a study on quantization of data of this network, using Brevitas, has to be done taking into account the tradeoffs between area and performance from reducing the data bitwidth. This study must be diverse regarding the size in bitwidth of the data used throughout the network in order for the best bitwidth configuration to be chosen.

The accuracy and efficiency of inference of the network are then improved by merging the convolutional layers with the batch normalization layers with a tool developed in Python named Batch Normalization Merger, thus reducing the number of overall layers.

Fine-tuning is done to improve the accuracy further. The network will then be modelled by developing Intellectual Property (IP) capable of running configurable layers where each of the network's layers will be processed. The drivers for these IPs will also be developed in order to configure the IPs for each layer correctly. The weights will then be exported into the FPGA to be used with the developed system using another developed tool in Python named Weights Extractor. These tools are described in the following sections.

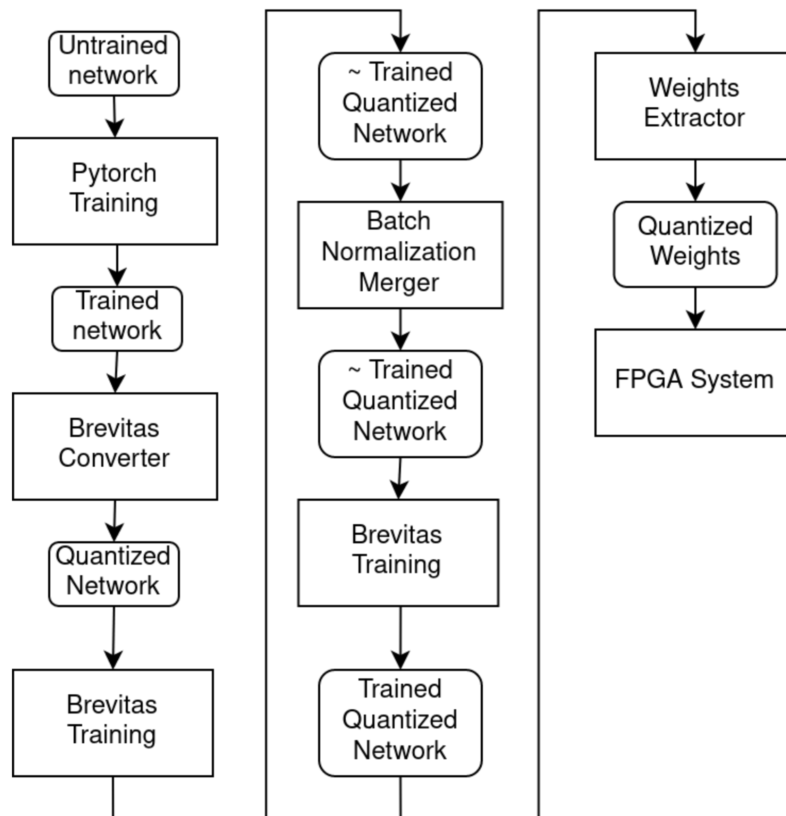

Figure 3.1: Design Flow

3.2 Tools for Training and Quantization

For this work the chosen Network is DeepLabV3+ with a Modified Aligned Xception Backbone, starting from the work done by Houda [22]. The network was trained with Pytorch[1] while the quantization training was performed with Brevitas[2].

3.2.1 Pytorch

PyTorch is an open-source machine learning framework implemented in Python optimized for both CPUs and GPUs. The training of networks with this framework is done by feeding the training script, where several training parameters are defined such as learning rates, metrics and loss functions, and a description of the neural network to be used. The network description consists of several layers, where each one is a function with its parameters passed as arguments. These include the number of input channels, number of output channels, kernel size, stride and padding (see example in figure 3.2).

[1] https://pytorch.org/
[2] https://github.com/Xilinx/brevitas

```python
self.conv1 = nn.Conv2d(3, 32, 3, stride=2, padding=1, bias=False)
self.bn1 = nn.BatchNorm2d(32)
self.relu1 = nn.ReLU(inplace=True)
```

Figure 3.2: Convolution, Batch Normalization and ReLu layers definition in Pytorch

An already developed Pytorch model[3] of the DeepLabV3+ model with several backbones was available which made the whole development process easier. The dataset was then ported to the already developed model and the network training was done.

3.2.2 Brevitas

Brevitas is a PyTorch library used for quantization-aware training that implements a set of building blocks at different levels of abstraction to model a reduced precision hardware data path at training time. This library provides several quantized versions of the standard PyTorch layers that can be replaced with the original PyTorch model. In order to perform the quantization of layers, their definition on the network description must be replaced by their quantized version of Brevitas e.g. nn.Conv2d is replaced by qnn.QuantConv2d and nn.ReLU is replaced by qnn.QuantReLU. These new layer functions include additional parameters to define the quantization characteristics. The quantization happens on the weights and also on the outputs of the layers. The quantization of the convolution outputs is done in the activation functions, while the convolution weights are quantized at the end of the backpropagation of the training process. A Brevitas activation function must be inserted after each layer in order for the next layer to receive a map with the right bitwidth.

3.3 Helper Tools

To better integrate Pytorch, Brevitas and FPGA deployment three python-based tools were developed. These tools are:

- **Brevitas Converter:** a python script that takes the description file of a Pytorch network and generates a description file for the Brevitas framework and converts the already trained weights to the generated Brevitas Network.
- **Batch Normalization Merger:** a tool that merges Batch Normalization layers with the previous convolution layer.
- **Weights Extractor:** a tool that extracts the weights and bias into binary files in order to be loaded into the SoC DDR memory.

[3] https://github.com/jfzhang95/pytorch-deeplab-xception

3.3.1 Brevitas Converter

Brevitas Converter is a Python3 script which takes a Pytorch network description with all its layers and converts it into a Brevitas compatible quantized network description along with the conversion of the weights. Brevitas has the ability to model a reduced precision hardware data path at training time. This works by replacing the layers in the Pytorch network description with their quantized selves. In Brevitas, quantization happens in the weights in the Convolution layers and it also happens during inference on the output of activations layers. For this work, the quantization was done with fixed point numbers and a scaling of values that are a power of two in order to get a faster processing time and cheaper resource consumption in hardware.

```
class CustomQuant(ExtendedInjector):
    bit_width_impl_type = BitWidthImplType.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 = False
    signed = True

    @value
    def quant_type():
        global weightBitWidth
        if weightBitWidth == 1:
            return QuantType.BINARY
        #elif  weightBitWidth ==2:
        #   return QuantType.TERNARY
        else:
            return QuantType.INT

class CustomWeightQuant(CustomQuant,WeightQuantSolver):
    scaling_const = 1.0

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

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

Figure 3.3: Quantization Engine declaration

For Brevitas it is necessary to declare the quantization engines for convolutions and weights, to be used in the Brevitas layer functions and they can be seen in Figure 3.3 where a global quantization engine called CustomQuant is declared along with quantizations engines for the weights(CustomWeightQuant) and for the activations(CustomActQuant and CustomSignedQuant). CustomQuant contains parameters that specify that all quantizations are fixed point(quant_type = QuantType.INT) or binary(quant_type = QuantType.BINARY) if the bitwidth is 1. It also sets the scaling factor to be a power of two(restrict_scaling_type = RestrictValueType.POWER_OF_TWO) in order to take advantage of the bit shift divisions and multiplications. It is also set that each channel has the same scaling throughout all the channels per

FM(scaling_per_output_channel = False) so that the computation and memory requirements are smaller at a possible cost of accuracy. All weights are set to be signed(signed = True) in order to have positive and negative values for them. For the Activation Engines the scaling is done by flooring(float_to_int_impl_type = FloatToIntImplType.FLOOR) the value so that only a bitshift is done in hardware. The only difference between the CustomActQuant and CustomSignedActQuant is that the latter is signed.

The Brevitas Converter replaces the convolutions and activation function with their quantized functions from Brevitas and sets the quantization engines as the ones stated above according to the type of layer. It is also added a function that changes the size of the bitwidths to be called before running the network as seen in figure 3.4.

```
def setBitWidths(weight,activation):
    global weightBitWidth
    global activationBitWidth
    weightBitWidth=weight
    activationBitWidth=activation
```

Figure 3.4: setBitWitdth Function

It is also added as the first layer of the quantized network an Identity Layer that quantizes the input to the chosen bitwidth. Seen in figure 3.5 and 3.6 is an example of an input LeNet Network with the output of the Brevitas Converter respectively.

```
'''LeNet in PyTorch.'''
import torch.nn as nn
import torch.nn.functional as F

class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.relu1  = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.relu2  = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2)
        self.fc1   = nn.Linear(16*5*5, 120)
        self.relu3  = nn.ReLU()
        self.fc2   = nn.Linear(120, 84)
        self.relu4  = nn.ReLU()
        self.fc3   = nn.Linear(84, 10)


    def forward(self, x):
        out = self.relu1(self.conv1(x))
        out = self.pool1(out)
        out = self.relu2(self.conv2(out))
        out = self.pool2(out)
        out = out.view(out.size(0), -1)
        out = self.relu3(self.fc1(out))
        out = self.relu4(self.fc2(out))
        out = self.fc3(out)
        return out
```

Figure 3.5: Input LeNet Network

```python
'''LeNetQuant in PyTorch.'''
import torch.nn as nn
import torch.nn.functional as F

class LeNetQuant(nn.Module):
    def __init__(self):
        super(LeNetQuant, self).__init__()
        global weightBitWidth
        global activationBitWidth

        self.imageQuant = qnn.QuantIdentity(bit_width=8, act_quant=CustomActQuant, return_quant_tensor=True)
        self.conv1 = qnn.QuantConv2d(3, 6, 5, weight_bit_width=weightBitWidth, bias_quant=BiasQuant, weight_quant=CustomWeightQuant, return_quant_ten
        self.relu1  = qnn.QuantReLU(bit_width=activationBitWidth, return_quant_tensor=True, act_quant=CustomActQuant) if weightBitWidth not in (1,2)
        self.pool1 = qnn.QuantMaxPool2d(2, return_quant_tensor=True)
        self.conv2 = qnn.QuantConv2d(6, 16, 5, weight_bit_width=weightBitWidth, bias_quant=BiasQuant, weight_quant=CustomWeightQuant, return_quant_te
        self.relu2  = qnn.QuantReLU(bit_width=activationBitWidth, return_quant_tensor=True, act_quant=CustomActQuant) if weightBitWidth not in (1,2)
        self.pool2 = qnn.QuantMaxPool2d(2, return_quant_tensor=True)
        self.fc1   = qnn.QuantLinear(16*5*5, 120, bias=True, weight_bit_width=weightBitWidth, bias_quant=BiasQuant, weight_quant=CustomWeightQuant, r
        self.relu3  = qnn.QuantReLU(bit_width=activationBitWidth, return_quant_tensor=True, act_quant=CustomActQuant) if weightBitWidth not in (1,2)
        self.fc2   = qnn.QuantLinear(120, 84, bias=True, weight_bit_width=weightBitWidth, bias_quant=BiasQuant, weight_quant=CustomWeightQuant, retur
        self.relu4  = qnn.QuantReLU(bit_width=activationBitWidth, return_quant_tensor=True, act_quant=CustomActQuant) if weightBitWidth not in (1,2)
        self.fc3   = qnn.QuantLinear(84, 10, bias=True, weight_bit_width=weightBitWidth, bias_quant=BiasQuant, weight_quant=CustomWeightQuant, return


    def setBitWidths(weight,activation):
        global weightBitWidth
        global activationBitWidth
        weightBitWidth=weight
        activationBitWidth=activation

    def forward(self, x):
        out = self.relu1(self.conv1(self.imageQuant(x)))
        out = self.pool1(out)
        out = self.relu2(self.conv2(out))
        out = self.pool2(out)
        out = out.view(out.size(0), -1)
        out = self.relu3(self.fc1(out))
        out = self.relu4(self.fc2(out))
        out = self.fc3(out)
        return out
```

Figure 3.6: Output of Brevitas Converter on the Lenet Network

As seen in the above Figures, each layer was replaced e.g. Conv2D by QuantConv2d and ReLU by QuantReLU, and the necessary quantization engines were added to each function.

Once the Brevitas Network is generated, the Brevitas Converter, if needed, takes the weights of the Pytorch Networks and converts them to one that can be used with the generated work. It does this by generating a blank file of the new network and transferring the weights to the new weights file. This only converts the weights and other parameters such as the activation scale need to be trained. Converting the weights speeds up the process by a considerable amount since the weights are trained before the conversion and the training process takes a lot more time with the quantized layers due to the increased number of features between Pytorch and Brevitas layers.

3.3.2 Batch Normalization Merger

Batch Normalization during inference time works like a 1x1 convolution i.e. each FM value is multiplied by the Batch Normalization Weight and added with a Batch Normalization Bias as seen in formula 3.1.

$$Out_{Value} = W_{BN} * FM_{Value} + Bias_{BN} \tag{3.1}$$

Since there isn't a non-linear layer between the convolutions layer and the batch normalization layers, the weight and bias can be fused together in order to reduce run time and memory requirements, as shown in formula 3.2.

$$Out_{Value} = W_{BN} * (W_{Conv} * FM_{Value} + Bias_{Conv}) + Bias_{BN} \tag{3.2}$$
$$W = W_{BN} * W_{Conv}$$
$$Bias = W_{BN} * Bias_{Conv} + Bias_{BN}$$
$$Out_{Value} = W * FM_{Value} + Bias$$

As seen in the formula above the 4 parameters are turned into only 2 thus reducing run time and memory requirements. The Batch Normalization Merger is a script that iterates every layer and fuses grouped convolutions and batch normalization layers automatically and outputs a new weights file. Due to the nature of quantization, there needs to be further training in order to reach higher values of accuracy.

3.3.3 Weights Extractor

Weights extractor is a Python3 script that extracts all the weights of a trained Brevitas quantized network into binary files in a format that can be read easily. The weights are stored z-wise in a fixed point format into 64-bit words with the weight's bitwidth being variable, as shown in Figure 3.7.

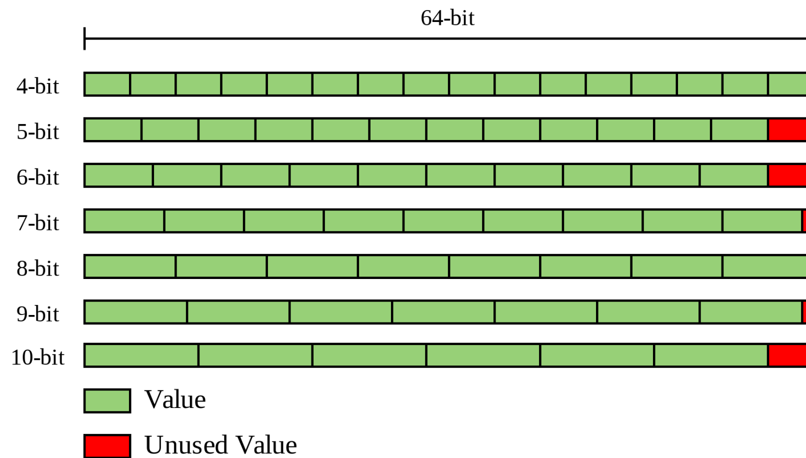

Figure 3.7: Example of variable bitwidth weights packed into a 64-Bit Word

As shown in the above Figure, there is some memory waste on bitwidths that are not a power of 2 numbers due to the last value not being able to fit into the remaining bits of the 64-bit word. The weight values are stored in order from the most significant bits to the least significant bits.

3.4 Network Description

This work uses the DeepLabV3+ network with a Modified Aligned Xception as a backbone. This network consists of 142 Convolutions and ReLu as activation functions. This network contains FM modifying features such as FM sums, average pooling and interpolations.

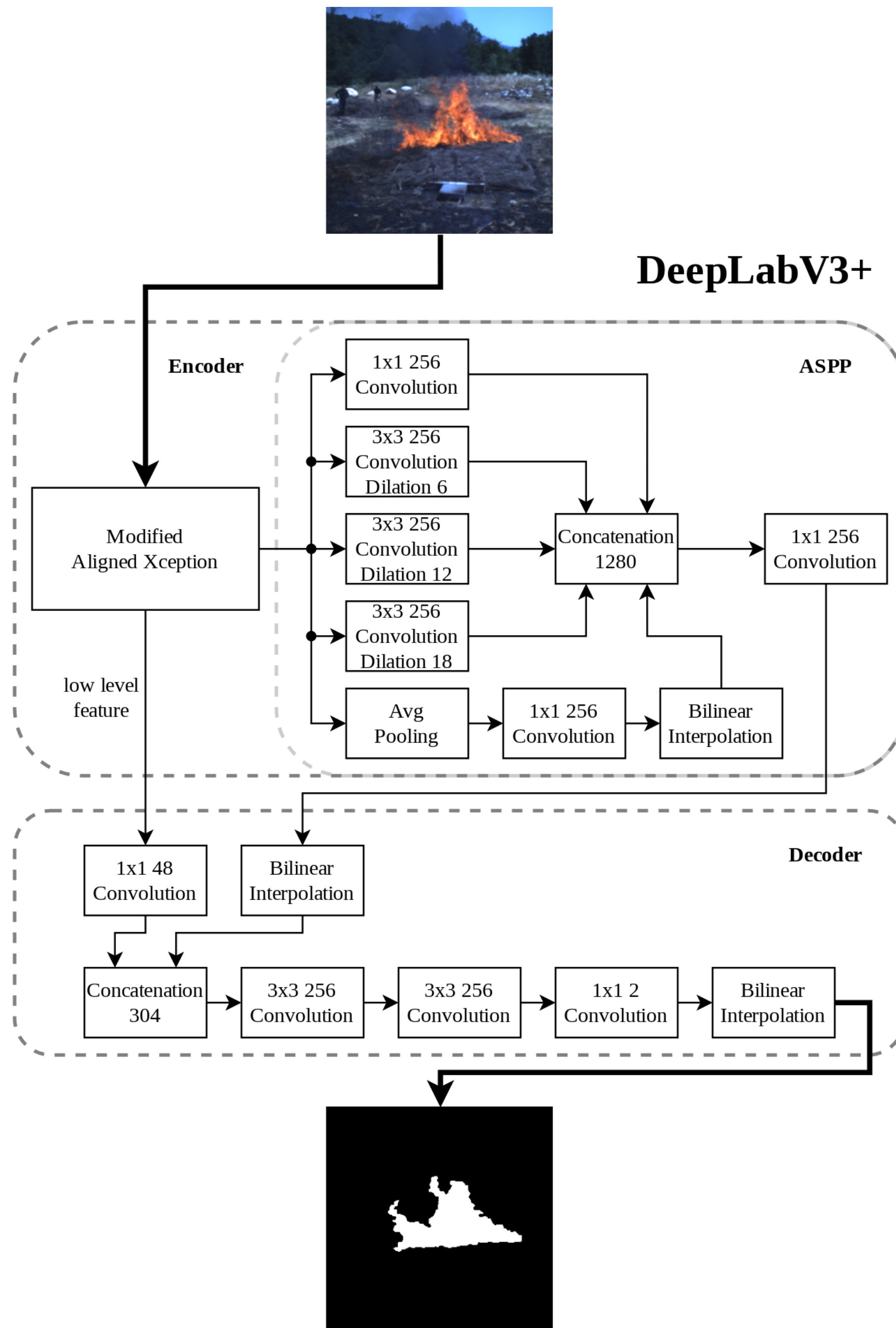

Figure 3.8: DeepLabV3+

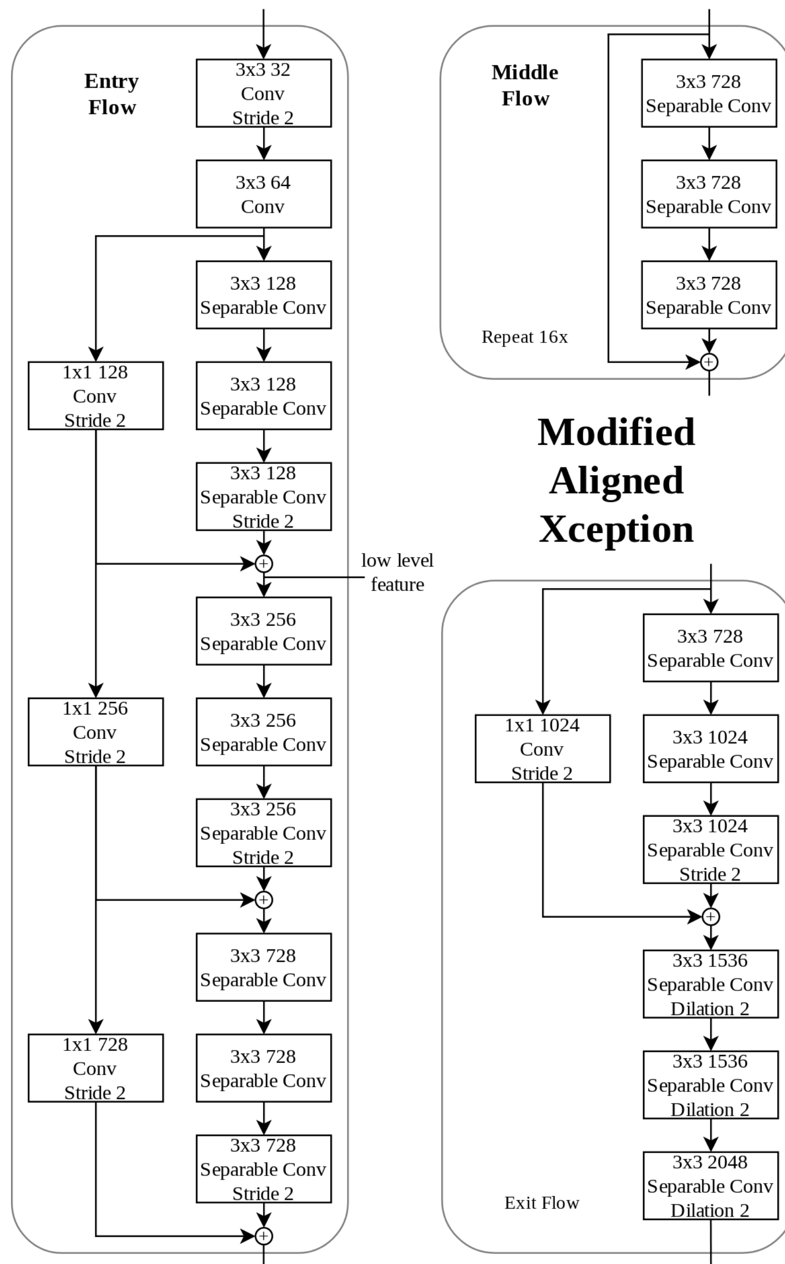

Figure 3.9: Modified Aligned Xception

Figures 3.8 and 3.9 show the Modified Aligned Xception and the DeepLabV3+ Networks, respectively, with all the convolutional layers stated in order of execution with their most important parameters as well. This network features Separable Convolutions, Atrous Convolutions, FM Sums, Average Pooling, Bilinear Interpolations and Concatenations. Each Separable Convolution block is composed of a depthwise convolution, a batch normalization layer, a pointwise convolution, another batch normalization layer and a ReLU layer. A convolution block is composed of a Convolution, a batch normalization

layer and ReLU Layer, except for the last convolution where there is no ReLU layer. The Network is divided into several parts the Encoder which is composed of a backbone network, in this case, the Modified Aligned Xception which could be replaced with another one, and the ASPP block which gathers features in multiple scales with the use of atrous convolutions. Next to the encoder is the decoder block which is responsible for decoding the features gathered by the encoder. As for the Modified Aligned Xception it is divided into three sections: the Entry flow which consists of several Separable Convolutions, some with a stride of 2 in order to reduce the size of the FMs. The Middle Flow is repeated 16 times in sequence and the Exit flow contains some convolutions with dilation characteristics.

Shown in Table 3.1 is a more detailed description of the network with all its layers and their parameters.

Table 3.1: DeepLabV3+ with Xception Backbone Network Description

		Conv #	Layer	Input Channels	Input Size	Kernel	Padding	Dilation	Stride	Output Channels	Output Size
XCEPTION		1	Conv1	3	300	3	1	1	2	32	150
		2	Conv2	32	150	3	1	1	1	64	150
	BLOCK1	3	DepthWise	64	150	3	1	1	1	64	150
		4	PointWise	64	150	1	0	1	1	128	150
		5	DepthWise	128	150	3	1	1	1	128	150
		6	PointWise	128	150	1	0	1	1	128	150
		7	DepthWise	128	150	3	1	1	2	128	75
		8	PointWise	128	75	1	0	1	1	128	75
		9	Conv	64	150	1	0	1	2	128	75
	BLOCK2	10	DepthWise	128	75	3	1	1	1	128	75
		11	PointWise	128	75	1	0	1	1	256	75
		12	DepthWise	256	75	3	1	1	1	256	75
		13	PointWise	256	75	1	0	1	1	256	75
		14	DepthWise	256	75	3	1	1	2	256	38
		15	PointWise	256	38	1	0	1	1	256	38
		16	Conv	128	75	1	0	1	2	256	38
	BLOCK3	17	DepthWise	256	38	3	1	1	1	256	38
		18	PointWise	256	38	1	0	1	1	728	38
		19	DepthWise	728	38	3	1	1	1	728	38
		20	PointWise	728	38	1	0	1	1	728	38
		21	DepthWise	728	38	3	1	1	2	728	19
		22	PointWise	728	19	1	0	1	1	728	19
		23	Conv	256	38	1	0	1	2	728	19
	BLOCK4	24	DepthWise	728	19	3	1	1	1	728	19
		25	PointWise	728	19	1	0	1	1	728	19
		26	DepthWise	728	19	3	1	1	1	728	19
		27	PointWise	728	19	1	0	1	1	728	19
		28	DepthWise	728	19	3	1	1	1	728	19
		29	PointWise	728	19	1	0	1	1	728	19
	BLOCK5	30	DepthWise	728	19	3	1	1	1	728	19
		31	PointWise	728	19	1	0	1	1	728	19
		32	DepthWise	728	19	3	1	1	1	728	19
		33	PointWise	728	19	1	0	1	1	728	19
		34	DepthWise	728	19	3	1	1	1	728	19
		35	PointWise	728	19	1	0	1	1	728	19
	BLOCK6	36	DepthWise	728	19	3	1	1	1	728	19
		37	PointWise	728	19	1	0	1	1	728	19
		38	DepthWise	728	19	3	1	1	1	728	19
		39	PointWise	728	19	1	0	1	1	728	19
		40	DepthWise	728	19	3	1	1	1	728	19
		41	PointWise	728	19	1	0	1	1	728	19
	BLOCK7	42	DepthWise	728	19	3	1	1	1	728	19
		43	PointWise	728	19	1	0	1	1	728	19
		44	DepthWise	728	19	3	1	1	1	728	19
		45	PointWise	728	19	1	0	1	1	728	19
		46	DepthWise	728	19	3	1	1	1	728	19
		47	PointWise	728	19	1	0	1	1	728	19

		Conv #	Layer	Input Channels	Input Size	Kernel	Padding	Dilation	Stride	Output Channels	Output Size
XCEPTION	BLOCK8	48	DepthWise	728	19	3	1	1	1	728	19
		49	PointWise	728	19	1	0	1	1	728	19
		50	DepthWise	728	19	3	1	1	1	728	19
		51	PointWise	728	19	1	0	1	1	728	19
		52	DepthWise	728	19	3	1	1	1	728	19
		53	PointWise	728	19	1	0	1	1	728	19
	BLOCK9	54	DepthWise	728	19	3	1	1	1	728	19
		55	PointWise	728	19	1	0	1	1	728	19
		56	DepthWise	728	19	3	1	1	1	728	19
		57	PointWise	728	19	1	0	1	1	728	19
		58	DepthWise	728	19	3	1	1	1	728	19
		59	PointWise	728	19	1	0	1	1	728	19
	BLOCK10	60	DepthWise	728	19	3	1	1	1	728	19
		61	PointWise	728	19	1	0	1	1	728	19
		62	DepthWise	728	19	3	1	1	1	728	19
		63	PointWise	728	19	1	0	1	1	728	19
		64	DepthWise	728	19	3	1	1	1	728	19
		65	PointWise	728	19	1	0	1	1	728	19
	BLOCK11	66	DepthWise	728	19	3	1	1	1	728	19
		67	PointWise	728	19	1	0	1	1	728	19
		68	DepthWise	728	19	3	1	1	1	728	19
		69	PointWise	728	19	1	0	1	1	728	19
		70	DepthWise	728	19	3	1	1	1	728	19
		71	PointWise	728	19	1	0	1	1	728	19
	BLOCK12	72	DepthWise	728	19	3	1	1	1	728	19
		73	PointWise	728	19	1	0	1	1	728	19
		74	DepthWise	728	19	3	1	1	1	728	19
		75	PointWise	728	19	1	0	1	1	728	19
		76	DepthWise	728	19	3	1	1	1	728	19
		77	PointWise	728	19	1	0	1	1	728	19
	BLOCK13	78	DepthWise	728	19	3	1	1	1	728	19
		79	PointWise	728	19	1	0	1	1	728	19
		80	DepthWise	728	19	3	1	1	1	728	19
		81	PointWise	728	19	1	0	1	1	728	19
		82	DepthWise	728	19	3	1	1	1	728	19
		83	PointWise	728	19	1	0	1	1	728	19
	BLOCK14	84	DepthWise	728	19	3	1	1	1	728	19
		85	PointWise	728	19	1	0	1	1	728	19
		86	DepthWise	728	19	3	1	1	1	728	19
		87	PointWise	728	19	1	0	1	1	728	19
		88	DepthWise	728	19	3	1	1	1	728	19
		89	PointWise	728	19	1	0	1	1	728	19

		Conv #	Layer	Input Channels	Input Size	Kernel	Padding	Dilation	Stride	Output Channels	Output Size
XCEPTION	BLOCK15	90	DepthWise	728	19	3	1	1	1	728	19
		91	PointWise	728	19	1	0	1	1	728	19
		92	DepthWise	728	19	3	1	1	1	728	19
		93	PointWise	728	19	1	0	1	1	728	19
		94	DepthWise	728	19	3	1	1	1	728	19
		95	PointWise	728	19	1	0	1	1	728	19
	BLOCK16	96	DepthWise	728	19	3	1	1	1	728	19
		97	PointWise	728	19	1	0	1	1	728	19
		98	DepthWise	728	19	3	1	1	1	728	19
		99	PointWise	728	19	1	0	1	1	728	19
		100	DepthWise	728	19	3	1	1	1	728	19
		101	PointWise	728	19	1	0	1	1	728	19
	BLOCK17	102	DepthWise	728	19	3	1	1	1	728	19
		103	PointWise	728	19	1	0	1	1	728	19
		104	DepthWise	728	19	3	1	1	1	728	19
		105	PointWise	728	19	1	0	1	1	728	19
		106	DepthWise	728	19	3	1	1	1	728	19
		107	PointWise	728	19	1	0	1	1	728	19
	BLOCK18	108	DepthWise	728	19	3	1	1	1	728	19
		109	PointWise	728	19	1	0	1	1	728	19
		110	DepthWise	728	19	3	1	1	1	728	19
		111	PointWise	728	19	1	0	1	1	728	19
		112	DepthWise	728	19	3	1	1	1	728	19
		113	PointWise	728	19	1	0	1	1	728	19
	BLOCK19	114	DepthWise	728	19	3	1	1	1	728	19
		115	PointWise	728	19	1	0	1	1	728	19
		116	DepthWise	728	19	3	1	1	1	728	19
		117	PointWise	728	19	1	0	1	1	728	19
		118	DepthWise	728	19	3	1	1	1	728	19
		119	PointWise	728	19	1	0	1	1	728	19
	BLOCK20	120	DepthWise	728	19	3	1	1	1	728	19
		121	PointWise	728	19	1	0	1	1	728	19
		122	DepthWise	728	19	3	1	1	1	728	19
		123	PointWise	728	19	1	0	1	1	1024	19
		124	DepthWise	1024	19	3	1	1	1	1024	19
		125	PointWise	1024	19	1	0	1	1	1024	19
		126	Conv	728	19	1	0	1	1	1024	19
		127	DepthWise	1024	19	3	2	2	1	1024	19
		128	PointWise	1024	19	1	0	1	1	1536	19
		129	DepthWise	1536	19	3	2	2	1	1536	19
		130	PointWise	1536	19	1	0	1	1	1536	19
		131	DepthWise	1536	19	3	2	2	1	1536	19
		132	PointWise	1536	19	1	0	1	1	2048	19

		Conv #	Layer	Input Channels	Input Size	Kernel	Padding	Dilation	Stride	Output Channels	Output Size
ASPP		133	AtrousConv1	2048	19	1	0	1	1	256	19
		134	AtrousConv2	2048	19	3	6	6	1	256	19
		135	AtrousConv3	2048	19	3	12	12	1	256	19
		136	AtrousConv4	2048	19	3	18	18	1	256	19
		137	Conv	2048	1	1	0	1	1	256	1
		138	Conv	1280	19	1	0	1	1	256	19
Decoder		139	Conv	128	75	1	0	1	1	48	75
		140	Conv	304	75	3	1	1	1	256	75
		141	Conv	256	75	3	1	1	1	256	75
		142	Conv	256	75	1	0	1	1	2	75

As can be seen in the table above the network is composed of 142 Convolutions, where 4 are normal Convolutions, 63 are Depthwise Convolutions (3 of which also have a dilation parameter of 2), 72 are Pointwise Convolutions and 3 are Dilated Convolutions. Every Convolution(except Convolution #142), Pointwise Convolution and Dilated Convolution are followed by a ReLU Layer and a Batch Normalization layer, while Depthwise Convolutions only have a Batch Normalization layer afterwards. All Blocks on the backbone have 3 Consecutive Separable Convolutions which the resulting FM is added to the blocks input FM while blocks #1, #2, #3 and #20 have the output of the Separable Convolutions added to the output of a Pointwise Convolution. There is also an average pooling operation done after convolution #132 where the resulting 2048x19x19 FM turns into a 2048x1x1 FM. There are 3 instances of bilinear interpolations of FMs in this network: the first being the interpolation after convolution #137 where a 256x1x1 FM turns into a 256x19x19 FM, the second being after convolution #138 where a 256x19x19 FM turns into a 256x75x75 FM and the third being after the last convolution (#142) where the 2x75x75 FM is turned into a 2x300x300 FM. There are also 2 instances of FM concatenation on the z-axis: the first being between the outputs of the Atrous Convolutions (#133-#136) and the first bilinear interpolation and the second being between the resulting FM of convolution #139 and the second bilinear interpolation.

3.4.1 Corsican Dataset

The Corsican dataset[4] is composed of a total of 1135 images of fires acquired in the visual range (i.e RGB values) and in the near-infrared (i.e single channel image) under various conditions of positioning, weather, vegetation, distance to the fire and brightness as seen in Figure 3.10.

Figure 3.10: Example images of the Corsican Dataset

Along with each of these images are also the truth images which consist of black and white images where the black pixels represent the background i.e. no fire and the white pixels represent fire. The truth images are used in the training process in order to evaluate the accuracy of the network between

[4] http://cfdb.univ-corse.fr/

training epochs and at the end of training where the final network accuracy is measured.

3.5 Work flow

The dataflow regarding the network will be the following: the Network will be trained using only non-quantized Pytorch Functions where the maximum accuracy will be recorded. Afterwards, the network description file along with the trained weights will be converted to a quantized version using the developed Tool Brevitas Converter, followed by a small fine-tuning. Then the batch normalization layers will be merged with the previous convolutions in order to reduce the total number of layers during inference. Then the quantized network will be submitted to a final stage of fine-tuning where the optimal accuracy will be achieved.

4

Hardware Accelerator for DeepLabV3+

The hardware accelerator was designed in order for the network to run from start to end without software intervention as it would prove to be a bottleneck for the whole system. All features of the network such as the convolutions, sum of maps, average pooling and interpolation were kept in mind when developing the accelerator.

The hardware accelerator was developed using Xilinx Vitis 2022.1 High Level Synthesis. The accelerator development targeted an implementation in the PL of Xilinx Zynq UltraScale+ SoC ZU3EG FPGA, more specifically part number xczu3eg-sbva484-1-i, present in the Ultra96-v2 development board. The accelerator was developed with configurable bitwidth for the weights and activations.

The Zynq UltraScale+ Architecture combines a dual-core ARM Cortex-A53 PS along with an FPGA. The interface between these two elements is based on the Advanced eXtensible Interface (AXI) standard, which provides for high bandwidth and low latency connections. Using an SoC with a PL and a PS combined has the advantage over using a softcore processor of reducing PL Hardware resources and having significant performance improvements.

Vitis HLS is a high-level Synthesis tool that allows C and C++ functions to be compiled into RTL components that can be implemented in the PL fabric. The design flow of Vitis HLS consists of compiling and simulating the algorithm, analysing and optimising the design, synthesising it into an RTL design, verifying the RTL implementation and then exporting the RTL IP to Vivado to be implemented on the device.

The Hardware architecture for this work is composed of Processing Blocks which compute each and every layer iteratively. The PS will send via DMA the input FMs and weights of each layer and then receive, also through DMA, the output of those layers.

4.1 Hardware Design

For the development of the hardware, a few choices had to be made beforehand. The bitwidth of the weights, bias and activations were chosen to be statically configurable (at compiling time), and the data structure was chosen to be stored z-wise i.e. in the same pixel of a map the values of channels are stored consecutively.

The architecture is split into 2 different convolution types: Regular Convolution and Depthwise Convolution. The Regular Convolution block also computes the pointwise convolutions. For the FM modifying features present in the network such as FM Sums, Interpolation and Averaging, separate blocks were developed depending on whether they are done after or before the Convolutions. The data will be transferred to and from the respective processing Blocks using the DMAs. Since the pre and post-processing elements are only relevant to the regular convolution block, the components developed for these elements will only be connected to the Convolution IP, as shown in figure 4.1.

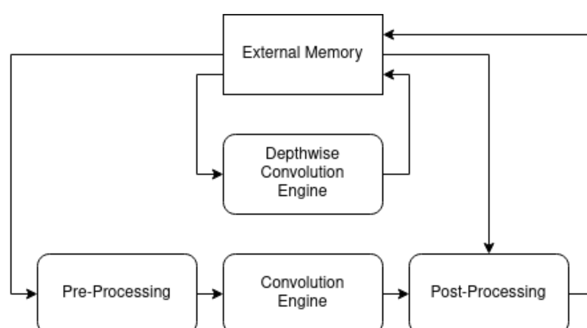

Figure 4.1: IP Topology

Figure 4.2 shows an example of the data structure for the 4-bit filter and 8-bit bias values for a convolution featuring 2 filters with 23 1x1 Kernels. Since each 64-bit value can hold 16 4-bit values, each filter is represented in only 2 64-bit values, the second one is partially filled with zeros. The order of the values is from the Most Significant Byte (MSB) to the Least Significant Byte (LSB) and so the unused values are kept in the LSB part and are filled with zeros. The same concept applies with the Biases values but they are instead 8-bit values and for this example, there are only 2 bias values and can be represented with only one 64-bit value.

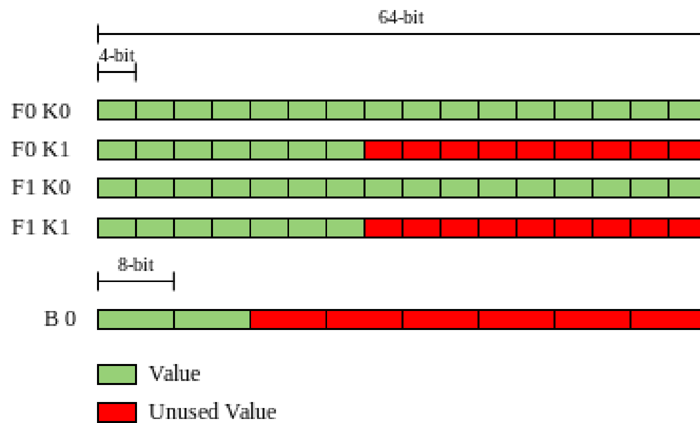

Figure 4.2: Filter and Bias data example

Figure 4.3 shows an example of the data structure for 2 pixels of a 4-bit FM with 30 z-wise channels. As in the filter data structure, the unused values are filled with zeros.

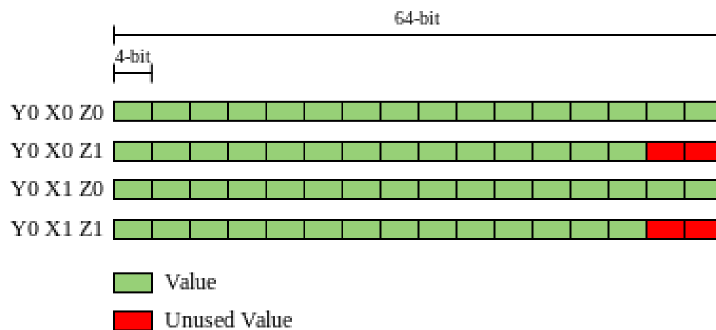

Figure 4.3: Feature Map data example

All developed components use the AXI4-Lite protocol for parameter configuration, while the data inputs and outputs are transferred to and from the external memory with the use of the AXI4-Stream protocol. The AXI4-Lite protocol is a point-to-point bidirectional interface between an IP core and the processor while the AXI4-Stream protocol is designed for transporting arbitrary unidirectional data, one value per clock cycle.

The bit-width chosen for the AXI4-Stream data was 64-bit, which for an example of 4-bit weights and 8-bit bias, results in 16 values of weights streamed per clock cycle and 8 values of bias per clock cycle. As such, for 4-bit weights the representation range is 0 to 15 for unsigned values and -8 to 7 for signed values and -128 to 127 for 8-bit bias values.

4.1.1 Convolution IP

The Convolution IP features enough PEs to fill a 64-bit output word, for an example of 4-bit activations there would be 16 PEs and each PE would also perform 16 MAC operations. In order to keep local memory usage as low as possible while reducing the number of data transfers, there are only enough BRAMs to store a pre-determined number of biases and weights and 4 lines of the biggest input map of the network, since the biggest kernel size is also 3 and the extra line is used to store the next line of the feature map.

In the beginning, the IP receives, through the AXI4-Lite port, the parameters of the convolution and then begins to store into the BRAMs the bias and weights values from the AXI-Stream input port and then receives 3 lines of the map. Then, for each output pixel, the PEs accumulators are reset to the values of the corresponding filter bias. Afterwards, for each output pixel, the dot products of the weights with the corresponding map values are calculated. When the PEs computations are done, each value of the accumulators is scaled and concatenated into a 64-bit value which is then streamed out through the output AXI-Stream port. While the PEs are working the next pixel of the input FM is being read into the extra line of BRAMs. This process repeats until the convolution is done. The entire IP is pipelined in order to maximize the throughput.

Figure 4.4 shows the block diagram of the convolution IP, where the behaviour described above is illustrated.

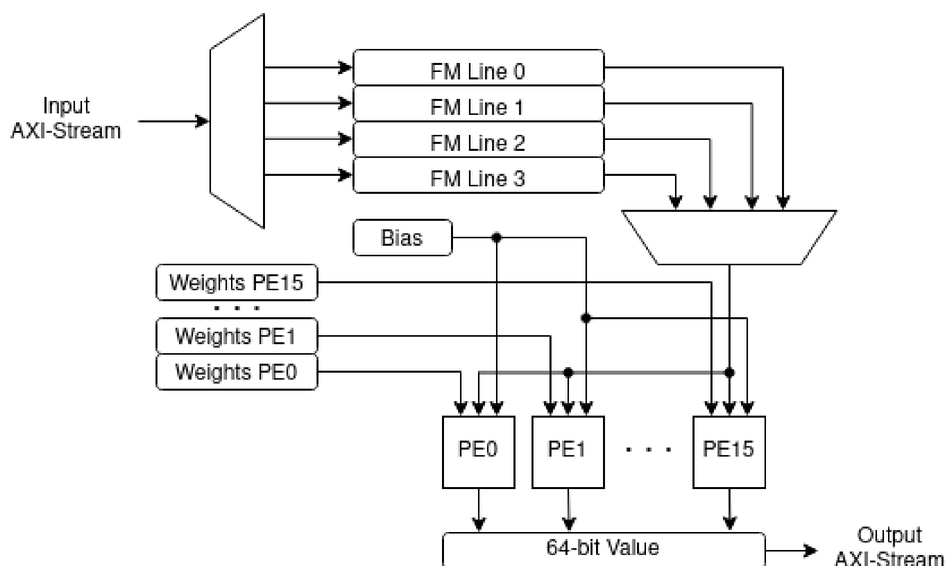

Figure 4.4: Convolution IP

The accumulators bitwidth must be big enough to hold all accumulations and can be determined with the formula 4.1, where $NumAdds$ is the number of additions per output pixel and A_{Width} and W_{Width} are the activations and weights bitwidth.

$$AccumBitWidth = Ceiling(log_2(NumAdds * (2^{A_{Width}+W_{Width}}) - 1)) \quad (4.1)$$

Since $NumAdds$ is the biggest number of additions in the network per output pixel, which is $KernelSize^2 \times NumZChannels = 3^2 \times 2048$, and for a A_{Width} and W_{Width} of 4, the accumulator bitwidth would be 23.

4.1.2 Depthwise Convolution IP

The Depthwise Convolution IP works similarly to the Convolution IP, the difference being that each PE does 9 MAC operations, corresponding to a 3×3 kernel. For an example of 4-bit activations, the Depthwise Convolution IP has 16 PEs with 9 MACS each and therefore is able to compute a total of 144 MACs/Cycle. As in the Convolution IP, at the beginning of the execution, the bias and weights are read from the AXI-Stream input followed by 3 lines of the input map. Since the IP does a full 3×3 kernel in each PE per cycle, 9 input map values, as well as 9 weight values, are fed into each PE. For the Input Map, there are 12 groups of BRAMS or 4 groups of lines, 3 for the convolution and 1 for storing the next line in parallel with the computation. Each PE computes its output value and then all the values are concatenated and sent out to the output AXI-Stream Port.

The accumulator bitwidth is also determined by the formula 4.1. As $NumAdds$ is now 3^2 and for a A_{Width} and W_{Width} of 4, the accumulator bitwidth would be set to 12.

Figure 4.5 shows the block diagram of the Depthwise Convolution IP, where the behaviour described above is illustrated.

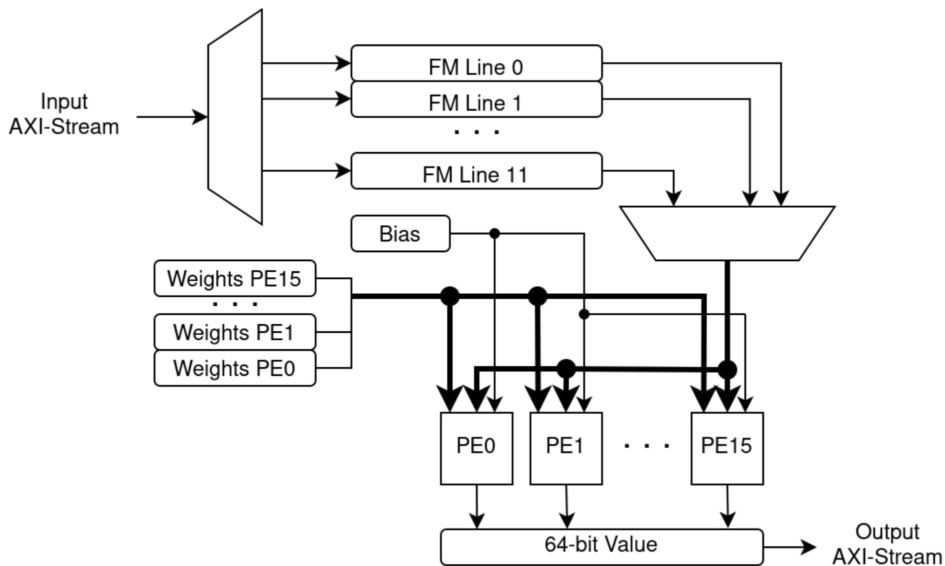

Figure 4.5: Depthwise IP

4.1.3 Pre-Processing IP

The Pre-Processing IP is able to average a variable-sized map into a 1x1 map by reading the input stream values and adding them to the previously read values. Once all the values are read, the division by the total number of pixels of the resulting value of the sums is done. The addition, as well as the division of the values, is done in parallel 16 times, for an example of 4-bit activations, in order for there to be an iteration interval of 1. The division is replaced by a multiplication of a value calculated at the beginning of the IP's execution ($\frac{2^{25}}{N}$) and a bit shift represented by a division by 2^{25} in order to reduce the hardware and execution time, as shown in equation 4.2.

$$Avg = \frac{\sum_{i=0}^{N-1} val[i]}{N} = \frac{\sum_{i=0}^{N-1} val[i] * \frac{2^{25}}{N}}{2^{25}} \tag{4.2}$$

The accumulator bitwidth is set to 13, as determined by the formula 4.1, where $NumAdds$ is now 19^2, A_{Width} is 4 and W_{Width} is 0. The BRAM bitwidth is $208 = 16 \times 13$, because there must be 16 operations in parallel.

If there is no need for averaging of maps then the streamed values from the input port are then streamed out.

4.1.4 Post-Processing IP

The Post-Processing IP implements the addition and interpolations of maps. In order to achieve an iteration interval of 1 while adding maps, it is necessary to have 2 input AXI4-Stream ports, one connected directly to the output of Convolution and the other to the external memory. When a value from each port is received the 64-bit values are partitioned into 4-bit values, if using 4-bit activations, and added in parallel. The additions are set to be signed or unsigned by setting the signed parameter previously via AXI4-Lite. Afterwards, each value is scaled according to the scale parameter, also configured via AXI4-Lite, and if the resulting value underflows or overflows it is saturated to the minimum or maximum representable values. Afterwards, the values are attached again into a 64-bit value and then streamed out to the external memory.

Bilinear interpolation works by applying linear interpolation in two directions. The linear interpolation formula can be seen in formula 4.3, where x1 and x2 are the known coordinates values in a 1 dimensional plane, Q1 and Q2 are the values associated with those coordinates and P is the value to be calculated for the new x.

$$P = Q_1 \frac{x_2 - x}{x_2 - x_1} + Q_2 \frac{x - x_1}{x_2 - x_1} \tag{4.3}$$

The Bilinear Interpolation formula, seen in formula 4.4, can be deduced from the linear interpolation

formula when moving on to a 2 dimensional plane.

$$P = Q_{11}\frac{(x_2-x)(y_2-y)}{(x_2-x_1)(y_2-y_1)} + Q_{21}\frac{(x-x_1)(y_2-y)}{(x_2-x_1)(y_2-y_1)} + Q_{12}\frac{(x_2-x)(y-y_1)}{(x_2-x_1)(y_2-y_1)} + Q_{22}\frac{(x-x_1)(y-y_1)}{(x_2-x_1)(y_2-y_1)} \tag{4.4}$$

Implementing this formula with fixed point numbers requires scaling the position of the known points (smaller map) in order for there not to be any fractional bits. The scaling factor is related to the size of the bigger map e.g if the bigger map has a size of 10 pixels then the scaling factor would be 9. Formula 4.4 can be rewritten in the following way:

$$\begin{aligned} first &= \frac{(x_2-x)(y_2-y)*f^2}{(x_2-x_1)(y_2-y_1)*f^2} \\ second &= \frac{(x-x_1)(y_2-y)*f^2}{(x_2-x_1)(y_2-y_1)*f^2} \\ third &= \frac{(x_2-x)(y-y_1)*f^2}{(x_2-x_1)(y_2-y_1)*f^2} \\ fourth &= \frac{(x-x_1)(y-y_1)*f^2}{(x_2-x_1)(y_2-y_1)*f^2} \\ P &= Q_{11}first + Q_{21}second + Q_{12}third + Q_{22}fourth \end{aligned} \tag{4.5}$$

In order to reduce hardware and execution time, the algorithm is further simplified, as

$$\begin{aligned} first &= (x_2-x)(y_2-y)*f^2 \\ second &= (x-x_1)(y_2-y)*f^2 \\ third &= (x_2-x)(y-y_1)*f^2 \\ fourth &= (x-x_1)(y-y_1)*f^2 \\ P &= \frac{Q_{11}first + Q_{21}second + Q_{12}third + Q_{22}fourth}{(x_2-x_1)(y_2-y_1)*f^2} \end{aligned} \tag{4.6}$$

The final division of the interpolation can be replaced by a bit shift by first multiplying a factor that is the result of the division between a power of two and the actual denominator. This factor is calculated at the beginning of the interpolation process since the value is always the same. The formula can then be rewritten:

$$\begin{aligned} F &= \frac{2^{25}}{(x_2-x_1)(y_2-y_1)*f^2} \\ P &= \frac{(Q_{11}first + Q_{21}second + Q_{12}third + Q_{22}fourth)*F}{2^{25}} \end{aligned} \tag{4.7}$$

Before starting the Bilinear Interpolation the IP stores into BRAMs 2 lines of the map. Afterwards, at every clock cycle, 4 64-bit values, containing 16 values each, are loaded and the interpolation is performed while loading the next line of values. The interpolation is performed for each value to be streamed out of the IP achieving a parallelism of 16.

If there is no need for interpolation or the addition of maps, the streamed values from the input ports are directly streamed out.

The flowchart detailing the algorithm can be seen in figure 4.6.

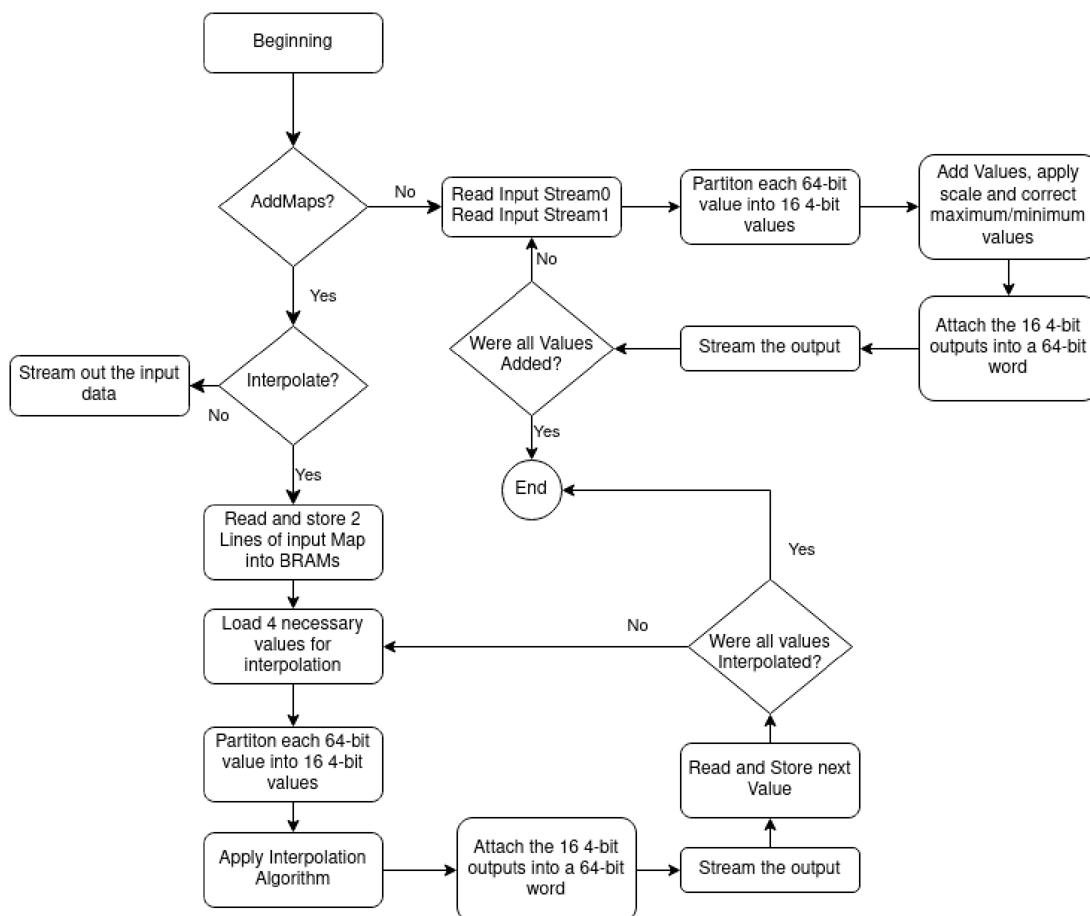

Figure 4.6: Post Processing IP Flow Chart

4.2 FPGA Image Build

The IPs were exported from Vitis HLS to Vivado where they were interconnected according to the topology designed in Figure 4.1. Two FIFO BRAM blocks were added at the end of the Depthwise Convolution and at the end of the Post-Processing IP in order for the values to be instantly written out of the IPs and not halt their processing. The sizes of these FIFO blocks were set to 2048 64-bit values due to the avail-

able BRAMs on the device. Three DMAs were also added in order for the inputs to be written to the IPs and for the outputs to be read into the external memory. 2 DMAs have a write and read channel for writing the weights and inputs FMs and reading the output FMs, one (dma2) has the input and output ports connected to the Depthwise IP and the other (dma0) has the input port connected to the Pre-Processing Ip and the output port connected to the Post-Processing IP. The remaining DMA (dma1) only has a write port for writing input FMs to be added in post-processing and is connected to the second input port of the Post-Processing IP.

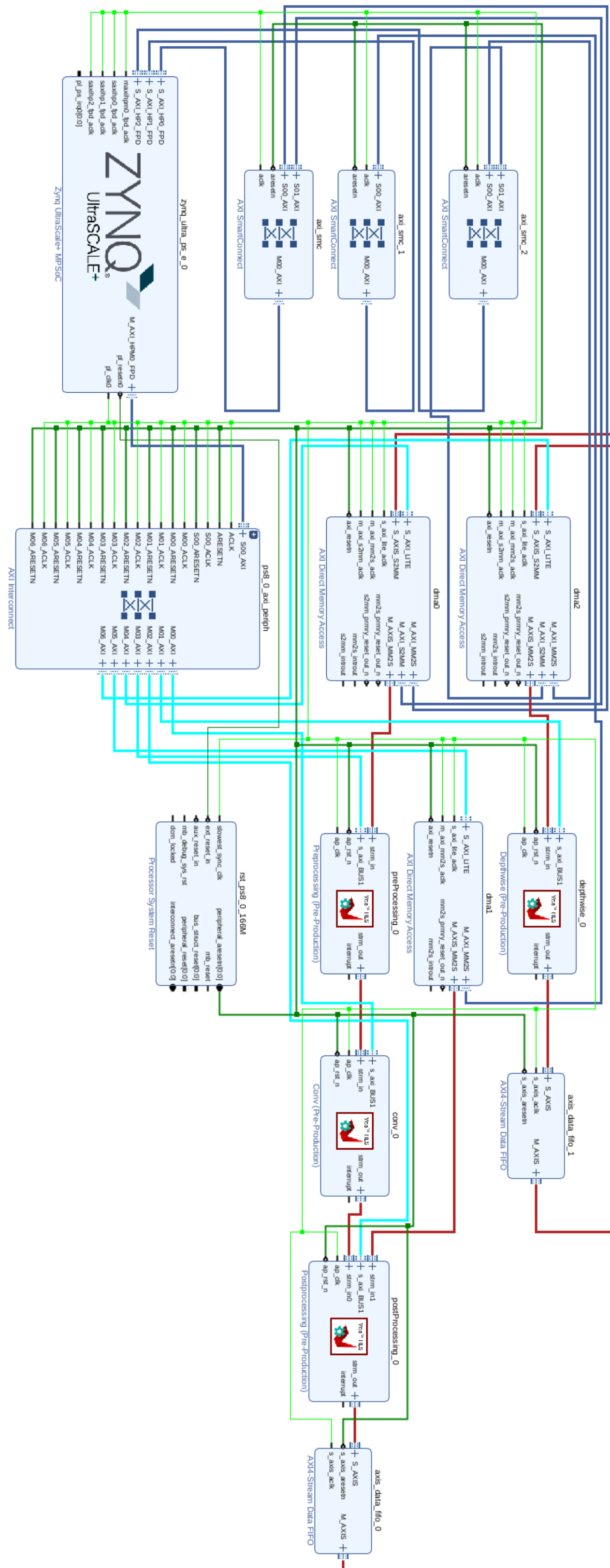

Figure 4.7: Vivado Block Design

Figure 4.7 shows the complete Vivado Block Design where it is visible the connections between the developed IPs and the DMAs. The PS block and the rest of the blocks are generated by Vivado in order to have a functioning system.

4.3 Baremetal Deployment

The network will be executed layer by layer by sending the corresponding data to the correct IPs and receiving the output FMs through the DMAs with the use of the developed drivers. A platform was created in Vitis IDE in order for the drivers to be written in C. It was necessary to write drivers for different kinds of convolution flows: normal flow, partitioned flow, and dilated flow. The normal flow consists of setting the convolutions parameters through AXI-lite which can be done by writing to the necessary addresses, starting the IP execution by setting the IP's start signal on the control register by AXI-Lite, sending in order the bias and weights through AXI-Stream/DMA from the external memory to the IPs in a contiguous fashion, setting up the transfer from the IPs output to the external memory through AXI-Stream/DMA and then sending the FM through AXI-Stream/DMA. Setting up the output transfer before the input FM transfer makes sure that minimal time is lost between transfers because once the output is ready it can be transferred immediately. After every transfer of inputs and before starting the next one it is verified if the transfer is already finished in order for no errors to occur. The convolution is considered done once the output transfer is finished and the signal done of the control register from the IP AXI-Lite is read. The partitioned flow has the same steps in the beginning as the normal flow, the difference being when receiving the output FM, since it is partitioned the memory positions won't be contiguous for different pixels, the output DMA Transfer is set up as many times as there are pixels, each one set to the correct memory address. FM concatenations can be done with the use of the partitioned flow.

Since dilated convolution can't be produced in a straightforward way with the developed IPs, they have to be partitioned as in [23].

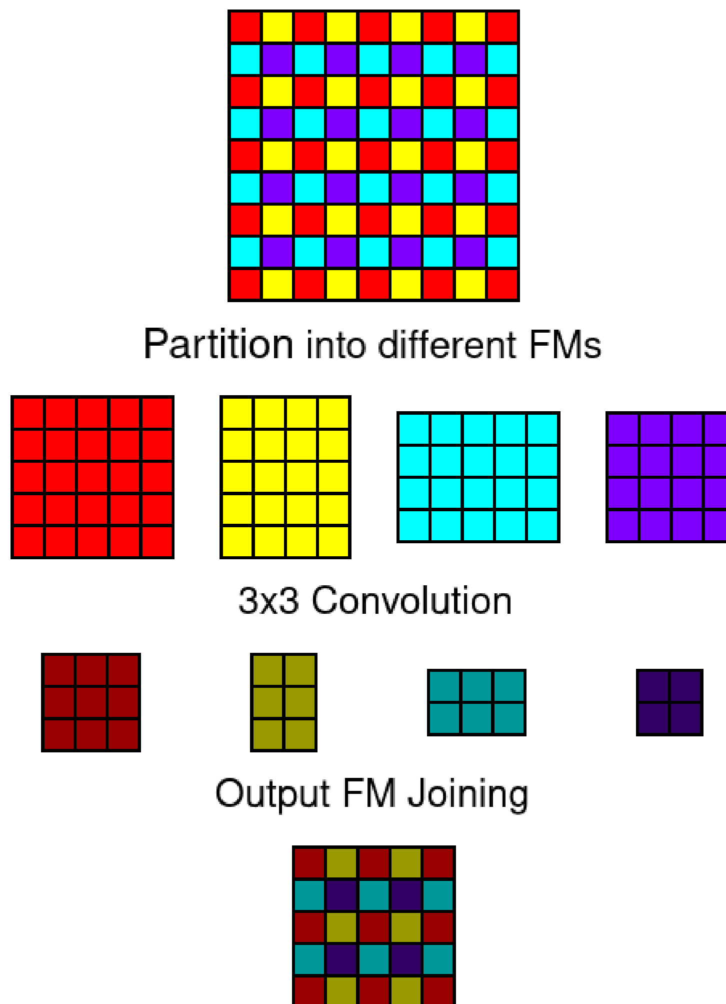

Figure 4.8: Atrous Convolution with dilation rate of 2 with partitioned FMs

Figure 4.8 demonstrates how atrous convolutions can be processed with partitioned FMs, using the example of a 3×3 Convolution with a dilation rate of 2. First, the original FM is partitioned into $rate^2$ sections followed by separate convolutions for each section. Then, the outputs of these separate convolutions are joined with their pixels being interleaved in the same manner as the first partitioning.

The dilated flow consists of the same steps of the beginning of the partitioned flow, this time the difference being that, when sending the input FM, several DMA transfers are set up to correctly send the required pixel positions, since they aren't in contiguous memory addresses. The output FMs are also saved with several DMA Transfers since the output values are also not in contiguous positions for different pixels.

These flows work for either the depthwise convolutions which use only the Depthwise Convolution IP as well with the normal convolutions which use the Pre-Processing, Convolution and the Post-Processing IPs.

The whole network layers were written in the baremetal C code main file with the created drivers with the help of the output description file from the Weights Extractor tool and the original network description file, in order for the execution of the network to be the same as the original.

The mIoU test is done by receiving the 2 channels of the last layers output FM and setting each output pixel as background if the first channel's respective pixel value is higher than the second channel, if not then the output pixel is set as fire, just like the original Pytorch training script does. With this test, the mIoU can be calculated for the total test set and for the individual images.

5

Results

This chapter describes and analyzes the results obtained throughout this work, from the training and testing of the Pytorch network, through the Quantization attempts and the final quantized network results. The hardware resources and timing values of the Built FPGA System will also be displayed along with the developed system timing values.

5.1 Pytorch Training

Using PyTorch, a DeepLabV3+ with an Xception Backbone was trained with the Corsican Fire dataset using only RGB images sized 300x300 due to the highest accuracy results achieved here [22]. The dataset was divided randomly into three parts: the first being the training partition where 682 images are used for training the network, the second being the validation partition where 226 images are tested after each epoch of training to keep track of the network train process and the third being the test partition where 226 images tested to evaluate the network final mIoU. The Training was done with a learning rate of 0.02 and with a Cross-Entropy loss function. The training machine used a 12GB Nvidia RTX2080Ti for training and it took around 1 hour to train on 100 epochs. The mIoU reached for validation and testing are 92.45% and 91.5% respectively. The reason the values of [22] were not reached might be due to a different dataset partition organization and the small adjustments made to the Xception backbone.

Figure 5.1 shows one example of the output of the trained neural network, where the output image has white pixels where the fire is identified and black pixels otherwise.

Figure 5.1: Example of input (left) and corresponding output (right) of the neural network

5.2 Quantization

After training the network, several quantization attempts with varying weights and activation bitwidths were done to determine the best mIoU to Hardware Resources compromise achievable. These quantization attempts consisted of using the developed tool Brevitas Converter and fine-tuning for 30 epochs. Since Brevitas training requires more computations than Pytorch-only training the combined time of processing these attempts was of around 72 hours. Table 5.1 shows the results obtained.

Table 5.1: Test mIoU(%) of Quantized Network with varying bitwidths

		Activations					
		8	7	6	5	4	3
Weights	8	92.02	92.05	91.97	91.67	90.16	77.24
	7	92.02	92.02	91.95	91.23	88.33	77.32
	6	92.02	92.03	91.92	91.66	87.87	76.40
	5	92.09	92.06	92.02	91.54	87.71	77.72
	4	92.00	92.01	91.82	91.47	**87.31**	76.22
	3	91.84	92.05	91.81	91.42	85.62	70.82

The bitwidths were tested from 8 to 3 for both activation and weights as lowering, even more, the size of the bitwidths would wield mIoU results below 80%. These results are probably not the best achievable with these bitwidths, since we stopped training after 30 epochs as running for more epochs would take a considerable amount of time. However, at 30 epochs the loss variation is small, meaning that only a very small improvement would be achieved with more training epochs. Decreasing both the weights and activations bitwidths lower the accuracy of the network, and with an activation bitwidth of 3, the accuracy decreases significantly independently of the weights bitwidth. Since, as shown in Figure 3.7, using weights with a bitwidth size that is not a power of 2 wastes memory resources, and as the achieved mIoU of 87.31% for a bitwidth of 4 for both activations and weights is only less than 5% lower than the mIoU achieved with the original Pytorch network, the rest of this work will use this bitwidth configuration.

Afterwards, the Batch normalization layers were merged with their respective convolutional layers,

using the developed tool Batch Normalization Merger, which decreased the mIoU to 81.21%. After further 50 Epochs of fine-tuned training, the mIoU reached 92.76% (an even higher value than the original Pytorch mIoU of 91.53%). The reason the quantized network performs better than the non-quantized Pytorch network might be due to the noise associated with the quantization.

5.3 Vitis HLS Results

5.3.1 Pipeline Characteristics

Vitis HLS displays after synthesis of the C functions the pipeline characteristics of the algorithm, them being the Iteration Latency and Iteration Interval, which describes the number of cycles needed for an entire pipeline iteration to be completed and the number of cycles needed to start a new iteration respectively. Keeping the Iteration Interval to 1 is paramount to achieving the best timing results as this value multiplies the number of clock cycles needed for the function to complete i.e. a function with an Iteration Interval of 2 takes 2 times longer to complete than one with an Iteration Interval of 1.

Tables 5.2, 5.3, 5.4 and 5.5 show the pipeline characteristics of the loops of the developed IPs.

Table 5.2: Pipeline characteristics of the Convolution IP

	Iteration Latency	Iteration Interval
Bias Store Loop	1	1
Filter Store Loop	1	1
FM Store Loop	1	1
Convolution	4	1

Table 5.3: Pipeline characteristics of the Depthwise Convolution IP

	Iteration Latency	Iteration Interval
Bias Store Loop	1	1
Filter Store Loop	1	1
FM Store Loop	1	1
Convolution	6	1

Table 5.4: Pipeline characteristics of the Pre-Processing IP

	Iteration Latency	Iteration Interval
Weights/Bias/FM Passthrough	1	1
Avg Add Loop	1	1
Avg Div Loop	5	1

Table 5.5: Pipeline characteristics of the Post-Processing IP

	Iteration Latency	Iteration Interval
FM Passthrough	1	1
FM Add Loop	2	1
Interpolation Store FM	1	1
Interpolation A	2	1
Interpolation B	13	1

The Loops consist of Storing Loops where values are read via AXI4-Stream and stored on the BRAMs, Passthrough Loops where values are read and sent directly to the output on the Pre-Processing and Post-Processing IPs when no FM modifications are needed. For the Pre-Processing IP, there is the Average Add Loop, where the input values are added and stored and the Average Division Loop where the added values are divided thus computing the average of those sums. For the Post Processing IP, there is the FM Addition Loop where the 2 FMs are added, the Interpolation A where the input FM is only 1x1 thus the interpolation only consists of repeating the input value several times, and the Interpolation B where the input FM has to be bilinearly interpolated. As shown, all Loops have an Iteration Interval of 1 and the Iteration Latency for all the Loops has a range between 1 and 13, the highest being on the Post Processing Interpolation B due to its high number of operations.

5.3.2 HLS Resource Estimate

Table 5.6: Vitis HLS Resource Estimates

	DSP	FF	LUT
Available	360	141120	70560
Convolution	14	5349	24259
Depthwise	9	9742	23080
PreProcessing	20	1477	4157
PostProcessing	105	5104	12738
Total	148	21672	64234

Table 5.6 shows the LUT, FF and DSP available resources and resource estimates for the IPs developed. All of the IPs hardware resources combined do not exceed the device hardware limitations.

5.3.3 BRAM Sizing

The local memory of all the IPs must be sized according to the network model to be executed.

Table 5.7: Biggest FM, Weights and Bias values of the network

	Max FM Values	Max Weights Value	Max Bias Values
Convolution	2048x19x19	256x2048x3x3	2048
Depthwise	728x40x40	1536x3x3	1536
Pre-Processing	2048x19x19	X	X
Post-Processing	256x19x19	X	X

Table 5.7 shows the sizes of the biggest FM needed for each IP (the sizes include padding, if any).

Table 5.8 shows the number of 64-bit Values needed for the FMs, except for the Pre-Processing IP where the values are 224-bit as stated before.

Table 5.8: Number of BRAMs for FM for each IP

	Number of Lines	Number of Values	Number of Values per Line	Number of BRAMs
Convolution	4	9728	2432	19.0
Depthwise	4	7084	1771	24.0
Pre-Processing	X	128	X	6.5
Post-Processing	3	912	304	8.0

The Depthwise IP needs more BRAMs for fewer Values than the Convolution IP because the Depthwise IP must access 9 FM values at the same time. This also means that utilized BRAMS can hold more values than the ones stated in the table above i.e the BRAMs are not being fully utilized as they only need to hold the maximum that the network requires.

Table 5.9: Number of BRAMs for Bias for each IP

	Number of Bias	Total Number of 64-Bit Values	Number of BRAMs
Convolution	2048	256	2
Depthwise	1536	192	2

Table 5.10: Number of BRAMs for Weights for each IP

	Number of Filters	Number of Kernels	Kernel Size	Total Number of Values	Total Number of 64-Bit Values	Number of BRAMs
Convolution	256	2048	3x3	4718592	294912	528
Depthwise	1	1536	3x3	13824	432	9

Tables 5.9 and 5.10 show the number of 64-bit Values needed for the BRAMs for the Bias and for the weights, respectively. The Convolution IP has the highest requirement in local memory (528 BRAMs for the weights alone). As the device contains only 216 BRAMs, the weights memory of the Convolution IP must be reduced to a total of 128 BRAMS, in order for all of the IPs BRAMs to fit in-chip, as seen in table 5.11. This means that 6 Convolutions will have to be partitioned, using the partitioned flow, due to their filters not fitting on the IP at once.

Table 5.11: Total Number of Brams Needed for each IP

	FM BRAMs	Bias BRAMs	Weights BRAMs	Total BRAMs
Convolution	19.0	2.0	128.0	149.0
Depthwise	24.0	2.0	9.0	35.0
Pre-Processing	6.5	X	X	6.5
Post-Processing	8.0	X	X	8.0
			Total	198.5

As seen in the table, the total number of BRAMs needed is thus reduced to 198.5 BRAMs, less than the total 216 available.

5.4 FPGA Build

Several Synthesis and Implementation runs were done in order to determine the highest frequency possible of these IPs and the maximum frequency reached is 175MHz. Table 5.12 shows the Hardware Utilization of the whole system.

Table 5.12: Vivado Implementation Resources

	LUTs (70560)		FFs (141120)		BRAMs (216)		DSPs (360)	
Block Name	**Qty**	**%**	**Qty**	**%**	**Qty**	**%**	**Qty**	**%**
Convolution IP	15075	21.4	4221	3.0	140.5	65.1	18	5.0
Depthwise IP	14503	20.6	6641	4.7	26.0	12.0	9	2.5
Pre-Processing IP	1487	2.1	1321	0.9	6.5	3.0	21	5.8
Post-Processing IP	4007	5.7	3024	2.1	8.0	3.7	106	29.4
FIFOs	146	0.2	122	0.1	9.0	4.2	0	0.0
DMAs	4062	5.8	5833	4.1	22.5	10.4	0	0.0
Others	6007	8.5	9098	6.5	0.0	0.0	0	0.0
Total	45287	64.2	30260	21.4	212.5	98.4	154	42.8

As shown, no hardware limits are hit, the highest resource usage being the BRAMs. The power report also states a power consumption of 2.82 W, where 1.64 W are for the Processing System alone. The FPGA bitstream was then generated and exported along with the necessary files to Vitis IDE in order to be able to run the network.

5.5 System Results

The network was then run in the Ultra96-v2 Board, the 226 test images were processed and their respective mIoU was evaluated. The input images, weights and truth images were stored, before execution, into the board DDR memory. The input images were saved already normalized, so no pre-processing on them is needed. Every output image was compared with the truth images, calculating their mIoU and

confirming that the implementation is correct. Then each of the layer's execution times were recorded and averaged over the 226 runs in order to figure out the system's throughput. The timing is recorded by calculating the difference between the number of PS clock cycles before and after executing each layer, and so these values include both the PL execution time and the data transfer times.

Table 5.13 shows the timing results measured along with the size in 64-bit words for the weights (including bias), the input FM and the Output FM. Also stated is the Number of MAC operations associated with the convolution and the number of cycles needed for the Depthwise or Convolution IP to process these layers (Pre-processing and Post-Processing are not included). The theoretical times of the convolutions along with the real averaged timing of each layer and their difference are also shown.

Table 5.13: Timing Results per Layer

Conv #	Layer	Input Weights	Input FM	Output FM	# Macs(M)	# Cycles	Time	Real Time	Time Factor	Comments
1	Conv1	2336	91204	45000	19.44	1631998	9325.7	9804.4	1.05	
2	Conv2	9280	46208	90000	414.72	1643672	9392.4	9868.7	1.05	
3	DepthWise	352	92416	90000	12.96	93068	531.8	696.2	1.31	
4	PointWise	4224	90000	180000	184.32	722328	4127.6	4342.7	1.05	
5	DepthWise	704	184832	180000	25.92	186136	1063.6	1396.0	1.31	
6	PointWise	8320	180000	180000	368.64	1444640	8255.1	8677.0	1.05	
7	DepthWise	704	184832	45000	6.48	186136	1063.6	1121.0	1.05	
8	PointWise	8320	45000	45000	92.16	362840	2073.4	2186.5	1.05	
9	Conv	4224	90000	45000	46.08	722328	4127.6	4341.6	1.05	
10	DepthWise	704	47432	45000	6.48	48136	275.1	352.2	1.28	
11	PointWise	16640	45000	90000	184.32	723880	4136.5	4352.2	1.05	
12	DepthWise	1408	94864	90000	12.96	96272	550.1	707.6	1.29	
13	PointWise	33024	90000	90000	368.64	1447728	8272.7	8695.3	1.05	
14	DepthWise	1408	94864	23104	3.33	96272	550.1	583.1	1.06	
15	PointWise	33024	23104	23104	94.63	375616	2146.4	2263.3	1.05	
16	Conv	16640	45000	23104	47.32	723880	4136.5	4352.9	1.05	
17	DepthWise	1408	25600	23104	3.33	26416	150.9	187.3	1.24	
18	PointWise	93912	23104	66424	269.12	1076347	6150.6	6468.0	1.05	
19	DepthWise	4040	73600	66424	9.46	75945	434.0	532.5	1.23	
20	PointWise	268632	66424	66424	765.30	3094327	17681.9	18575.5	1.05	
21	DepthWise	4040	73600	16606	2.37	75945	434.0	459.8	1.06	
22	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.5	1.05	
23	Conv	93912	23104	16606	67.28	1076347	6150.6	6466.2	1.05	
24	DepthWise	4040	20286	16606	2.37	21757	124.3	142.3	1.14	
25	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.0	1.05	
26	DepthWise	4040	20286	16606	2.37	21757	124.3	142.6	1.15	
27	PointWise	268632	16606	16606	191.32	800077	4571.9	4810.1	1.05	
28	DepthWise	4040	20286	16606	2.37	21757	124.3	142.5	1.15	
29	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.7	1.05	
30	DepthWise	4040	20286	16606	2.37	21757	124.3	142.4	1.15	
31	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.3	1.05	
32	DepthWise	4040	20286	16606	2.37	21757	124.3	142.5	1.15	
33	PointWise	268632	16606	16606	191.32	800077	4571.9	4810.5	1.05	
34	DepthWise	4040	20286	16606	2.37	21757	124.3	142.6	1.15	
35	PointWise	268632	16606	16606	191.32	800077	4571.9	4810.0	1.05	
36	DepthWise	4040	20286	16606	2.37	21757	124.3	142.6	1.15	
37	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.6	1.05	
38	DepthWise	4040	20286	16606	2.37	21757	124.3	142.4	1.15	
39	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.2	1.05	
40	DepthWise	4040	20286	16606	2.37	21757	124.3	142.4	1.15	
41	PointWise	268632	16606	16606	191.32	800077	4571.9	4810.3	1.05	
42	DepthWise	4040	20286	16606	2.37	21757	124.3	142.6	1.15	
43	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.9	1.05	
44	DepthWise	4040	20286	16606	2.37	21757	124.3	142.5	1.15	
45	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.6	1.05	
46	DepthWise	4040	20286	16606	2.37	21757	124.3	142.4	1.15	
47	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.2	1.05	

Conv #	Layer	Input Weights	Input FM	Output FM	# Macs(M)	# Cycles	Time	Real Time	Time Factor	Comments
48	DepthWise	4040	20286	16606	2.37	21757	124.3	142.4	1.15	
49	PointWise	268632	16606	16606	191.32	800077	4571.9	4810.2	1.05	
50	DepthWise	4040	20286	16606	2.37	21757	124.3	142.6	1.15	
51	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.8	1.05	
52	DepthWise	4040	20286	16606	2.37	21757	124.3	142.5	1.15	
53	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.4	1.05	
54	DepthWise	4040	20286	16606	2.37	21757	124.3	142.4	1.15	
55	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.0	1.05	
56	DepthWise	4040	20286	16606	2.37	21757	124.3	142.5	1.15	
57	PointWise	268632	16606	16606	191.32	800077	4571.9	4810.1	1.05	
58	DepthWise	4040	20286	16606	2.37	21757	124.3	142.6	1.15	
59	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.7	1.05	
60	DepthWise	4040	20286	16606	2.37	21757	124.3	142.4	1.15	
61	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.4	1.05	
62	DepthWise	4040	20286	16606	2.37	21757	124.3	142.5	1.15	
63	PointWise	268632	16606	16606	191.32	800077	4571.9	4810.5	1.05	
64	DepthWise	4040	20286	16606	2.37	21757	124.3	142.6	1.15	
65	PointWise	268632	16606	16606	191.32	800077	4571.9	4810.1	1.05	
66	DepthWise	4040	20286	16606	2.37	21757	124.3	142.6	1.15	
67	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.6	1.05	
68	DepthWise	4040	20286	16606	2.37	21757	124.3	142.4	1.15	
69	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.2	1.05	
70	DepthWise	4040	20286	16606	2.37	21757	124.3	142.5	1.15	
71	PointWise	268632	16606	16606	191.32	800077	4571.9	4810.3	1.05	
72	DepthWise	4040	20286	16606	2.37	21757	124.3	142.7	1.15	
73	PointWise	268632	16606	16606	191.32	800077	4571.9	4810.0	1.05	
74	DepthWise	4040	20286	16606	2.37	21757	124.3	142.4	1.15	
75	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.6	1.05	
76	DepthWise	4040	20286	16606	2.37	21757	124.3	142.5	1.15	
77	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.1	1.05	
78	DepthWise	4040	20286	16606	2.37	21757	124.3	142.5	1.15	
79	PointWise	268632	16606	16606	191.32	800077	4571.9	4810.3	1.05	
80	DepthWise	4040	20286	16606	2.37	21757	124.3	142.6	1.15	
81	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.9	1.05	
82	DepthWise	4040	20286	16606	2.37	21757	124.3	142.5	1.15	
83	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.4	1.05	
84	DepthWise	4040	20286	16606	2.37	21757	124.3	142.4	1.15	
85	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.0	1.05	
86	DepthWise	4040	20286	16606	2.37	21757	124.3	142.6	1.15	
87	PointWise	268632	16606	16606	191.32	800077	4571.9	4810.2	1.05	
88	DepthWise	4040	20286	16606	2.37	21757	124.3	142.6	1.15	
89	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.8	1.05	

Conv #	Layer	Input Weights	Input FM	Output FM	# Macs(M)	# Cycles	Time	Real Time	Time Factor	Comments
90	DepthWise	4040	20286	16606	2.37	21757	124.3	142.5	1.15	
91	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.3	1.05	
92	DepthWise	4040	20286	16606	2.37	21757	124.3	142.4	1.15	
93	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.7	1.05	
94	DepthWise	4040	20286	16606	2.37	21757	124.3	142.7	1.15	
95	PointWise	268632	16606	16606	191.32	800077	4571.9	4810.2	1.05	
96	DepthWise	4040	20286	16606	2.37	21757	124.3	142.7	1.15	
97	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.7	1.05	
98	DepthWise	4040	20286	16606	2.37	21757	124.3	142.4	1.15	
99	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.2	1.05	
100	DepthWise	4040	20286	16606	2.37	21757	124.3	142.5	1.15	
101	PointWise	268632	16606	16606	191.32	800077	4571.9	4810.3	1.05	
102	DepthWise	4040	20286	16606	2.37	21757	124.3	142.7	1.15	
103	PointWise	268632	16606	16606	191.32	800077	4571.9	4810.0	1.05	
104	DepthWise	4040	20286	16606	2.37	21757	124.3	142.5	1.15	
105	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.6	1.05	
106	DepthWise	4040	20286	16606	2.37	21757	124.3	142.4	1.15	
107	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.2	1.05	
108	DepthWise	4040	20286	16606	2.37	21757	124.3	142.5	1.15	
109	PointWise	268632	16606	16606	191.32	800077	4571.9	4810.3	1.05	
110	DepthWise	4040	20286	16606	2.37	21757	124.3	142.6	1.15	
111	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.8	1.05	
112	DepthWise	4040	20286	16606	2.37	21757	124.3	142.5	1.15	
113	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.5	1.05	
114	DepthWise	4040	20286	16606	2.37	21757	124.3	142.5	1.15	
115	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.0	1.05	
116	DepthWise	4040	20286	16606	2.37	21757	124.3	142.6	1.15	
117	PointWise	268632	16606	16606	191.32	800077	4571.9	4810.2	1.05	
118	DepthWise	4040	20286	16606	2.37	21757	124.3	142.6	1.15	
119	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.8	1.05	
120	DepthWise	4040	20286	16606	2.37	21757	124.3	142.6	1.15	
121	PointWise	268632	16606	16606	191.32	800077	4571.9	4809.4	1.05	
122	DepthWise	4040	20286	16606	2.37	21757	124.3	142.4	1.15	
123	PointWise	377856	16606	23104	269.12	1112638	6357.9	6685.2	1.05	
124	DepthWise	5632	28224	23104	3.33	30272	173.0	197.0	1.14	
125	PointWise	525312	23104	23104	378.54	1547968	8845.5	9297.9	1.05	
126	Conv	377856	16606	23104	269.12	1112638	6357.9	6684.9	1.05	
127	DepthWise	5632	33856	23104	3.33	33088	189.1	2076.6	10.98	Dilation=2
128	PointWise	787968	23104	34656	567.80	2320128	13257.9	13958.4	1.05	Part
129	DepthWise	8448	50784	34656	4.99	49632	283.6	2236.1	7.88	Dilation=2
130	PointWise	1181184	34656	34656	851.71	3480096	19886.3	20968.6	1.05	Part
131	DepthWise	8448	50784	34656	4.99	49632	283.6	2236.2	7.88	Dilation=2
132	PointWise	1574912	34656	46208	1135.61	4638304	26504.6	28163.3	1.06	Part

Conv #	Layer	Input Weights	Input FM	Output FM	# Macs(M)	# Cycles	Time	Real Time	Time Factor	Comments
133	AtrousConv1	262400	46208	5776	189.27	779424	4453.9	4684.0	1.05	Part
134	AtrousConv2	2359552	123008	5776	1703.41	11163296	63790.3	76181.6	1.19	Dilation=6
135	AtrousConv3	2359552	236672	5776	1703.41	15370400	87830.9	117879.0	1.34	Dilation=12
136	AtrousConv4	2359552	387200	5776	1703.41	19577504	111871.5	165418.0	1.48	Dilation=18
137	Conv	262400	128	16	0.52	35232	201.3	1189.9	5.91	Avg&Int&Part
138	Conv	164096	28880	5776	118.29	487152	2783.7	11413.0	4.10	Int&Part
139	Conv	3120	45000	16875	34.56	137190	783.9	9696.3	12.37	Part
140	Conv	350464	112651	90000	3939.84	15848597	90563.4	95096.6	1.05	
141	Conv	295168	94864	90000	3317.76	13346192	76264.0	80082.2	1.05	
142	Conv	264	90000	5625	2.88	93633	535.0	1046.6	1.96	Int

As shown, regular convolutions with no Pre-Processing or Post-Processing take around 5% more

than the theory time while the Depthwise convolutions with the same conditions take 15% more or higher. This is due to the number of MACs and size of memory transfers (Input Weights, Input FM and Output FM) associated with each convolution as shown in the table. Partitioned and dilated convolutions also have a significant timing increase compared to the theoretical times, this is due to the elevated number of DMA transactions generated in between processing. Interpolated and Averaged convolutions also have a Time Factor value higher than other convolutions, this is due to the increased number of cycles associated with the pre and post-processing and the theoretical times do not include these extra cycles.

Table 5.14: Timing Summary for various types of convolutions flows

		Theoretical		Real		Extra		
	#Convs	Time	%	Time	%	Time	%	%/Conv
Dilated Flow	6	264	31.2	366	36.6	101	67.0	11.2
Partitioned Flow	7	67	8.0	90	9.0	22	14.6	2.1
Normal Flow	129	514	60.8	543	54.4	28	18.5	0.1
Total	142	847	100.0	999	100.0	152	100.0	0.7

Table 5.14 details the theoretical time, the real-time and the extra time for each of the convolution flows used. The whole network takes 999ms to run 1 image, which means it has a throughput of 1 frame per second. The theoretical time, without pre-processing and post-processing, is 847ms which would mean a theoretical throughput of 1.18 frames per second. It is relevant to point out that the Atrous convolutions and partitioned convolutions take 81.6% of the extra time involved in the network, while only making up for only 13 of the 142 total convolutions. Future work could improve this by making use of DMAs with Scatter and Gather which are able to store DMA transaction requests in sequence, thus eliminating the delay between waiting for a transaction to end and starting a new one. If the dilated and partitioned flow layers took as much extra time as the normal flow layers (average time factor is 1.11%) then the estimated throughput would be 1.07 FPS or 931 ms per image.

Other solutions were also briefly explored in order to compare with this system. The first was a Software only version executing on one ARM A53 processor present in the device, where the number of floating point MAC operations per second measured was 66 million MACs per second. Since the network execution requires 28.9 GMACs, an estimate for the throughput for the software-only solution is 2.3 mFPS or 438 seconds per Image. The throughput for a CPU-only execution and a GPU execution was also measured using the original Pytorch network executing on an AMD Ryzen 7 5800H processor and on an Nvidia RTX3060 GPU. Table 5.15 presents this comparison.

Table 5.15: Throughput, Power and Cost Comparison of several Solutions

	Throughput	Power	Cost
Ultra96-v2	1FPS	5.82W	260€
Ultra96-v2 (SW only)	2mFPS	4.64W	260€
RTX3060	71FPS	170W	340€
AMD Ryzen 7 5800H	3.6FPS	45W	460€

The values for the Power for the Ultra96 boards were taken from Vivado power report with 3 Watts added to them as an estimate for the board's external components and DDR memory accesses while the CPU and GPU power values were taken from the manufacturer's maximum power consumption. The values for the cost were taken from the manufacturer's suggested retail price and converted to euros. These cost values only take into account the main components of their solutions and not the price of the necessary peripherals in order to run them, the exception being the Ultra96-v2 board which is that solution's only component. As can be seen in the Table above the throughput for the HW-SW solution is much better than the SW-only solution, only the Power being slightly higher in the hardware-software solution. The throughputs for the CPU and GPU solutions are also higher than the hardware-software solution with the trade-off of higher Cost and Power. For on-site fire-detecting applications, the developed HW-SW solution can be a great solution even with the smaller throughput due to its low cost and power consumption.

6 Conclusion

This work succeeded in training a fire-detecting DeepLabV3+ with a Modified Aligned Xception neural network with a final mIoU accuracy of 91.53%. The network was quantized and fine-tuned to a Fixed Point representation of 4 bits for both weights and activations. The quantized network achieved a final mIoU accuracy of 92.76%. A network hardware accelerator was developed and implemented on a Xilinx Zynq UltraScale+ SoC ZU3EG. A Configurable Layer Processor processing flow, with configurable weights and activation bitwidth, was implemented where the same accuracy was obtained. A final throughput of 1 FPS was achieved with an estimated power consumption of 5.82W thus beating the initial requirement of 0.1 FPS and low power consumption. The hardware resources used for the network hardware accelerator were 45K LUTs, 30K Flip Flops, 212.5 BRAMs and 154 DSPs, which equates to roughly 55% of the available resources of the ZU3EG device. This work can fit in low to mid-range devices.

6.1 Future Work

This work can be improved by better utilizing the device hardware resources since the available resources were only partially used. Increasing the number of PEs on the convolution engines and increasing the bitwidth of the AXI-Stream inputs and outputs will directly increase the throughput of the system. If using a bitwidth of 128 and doubling the number of PEs the estimated throughput would be 2FPS. Using DMAs with the Scatter and Gather functionality can also increase the throughput on the convolutions that use the partitioned and dilated flows due to the high number of DMA transactions associated with them. Using Scatter and Gather DMAs could increase the throughput to 1.07 FPS if the dilated and partitioned flow layers took as much extra time as the normal flow layers.

Bibliography

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[4] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 1800–1807.

[5] "Visual object classes challenge 2012 (voc2012)." [Online]. Available: http://host.robots.ox.ac.uk/pascal/VOC/voc2012/#devkit

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

[7] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping cnn onto embedded fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 35–47, 2018.

[8] L. Gong, C. Wang, X. Li, H. Chen, and X. Zhou, "Maloc: A fully pipelined fpga accelerator for convolutional neural networks with all layers mapped on chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2601–2612, 2018.

[9] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015.

[10] K. Smeda. (2019, Nov) Understand the architecture of CNN. Accessed 31-Dec-2021. [Online]. Available: https://towardsdatascience.com/understand-the-architecture-of-cnn-90a25e244c7

[11] C.-F. Wang. (2018, Aug) A basic introduction to separable convolutions. Accessed 31-Dec-2021. [Online]. Available: https://towardsdatascience.com/a-basic-introduction-to-separable-convolutions-b99ec3102728

[12] S. Minaee, Y. Y. Boykov, F. Porikli, A. J. Plaza, N. Kehtarnavaz, and D. Terzopoulos, "Image segmentation using deep learning: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2021.

[13] Y. Xing, L. Zhong, and X. Zhong, "An encoder-decoder network based fcn architecture for semantic segmentation," *Wireless Communications and Mobile Computing*, vol. 2020, pp. 1–9, 07 2020.

[14] H. Noh, S. Hong, and B. Han, "Learning deconvolution network for semantic segmentation," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1520–1528.

[15] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 12, pp. 2481–2495, 2017.

[16] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation," 2018.

[17] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," 2015.

[18] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2016.

[19] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[20] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 535–547.

[21] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017, pp. 1–6.

[22] H. Harkat, J. M. Nascimento, and A. Bernardino, "Fire detection using residual deeplabv3+ model," in *2021 Telecoms Conference (ConfTELE)*, 2021, pp. 1–6.

[23] K.-W. Chang and T.-S. Chang, “Efficient accelerator for dilated and transposed convolution with decomposition,” in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, pp. 1–5.