SoC-FPGA Deep Neural Network for Object Detection

Pedro Direita
Instituto Superior Técnico,
Universidade de Lisboa
Email: pedro.direita@tecnico.ulisboa.pt

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

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

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

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

1. Introduction

The evolution in deep neural networks (DNN) and it's application in image processing, unleash new developments in computer vision, including in the object detection field, where greater accuracy is achieved [1]. There is a broad range of applications where object detection is applied and can be applied in the future.

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

2. Deep Neural Networks

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

2.1. Convolutional Neural Network

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

The convolution layer is the main layer of a CNN. Considering a convolution layer with a kernel size of KxK, and with an input (g) with N channels where w is the tensor of weights and b the tensor of bias, the output of the neuron (f) in the position i row, j column of the filter l is modeled by equation 1.

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

2.2. Image classification DNN Models

Image classification is a process where is identified the object depicted in the image, in accordance to a set of possible classes [2]. The implementation of image classifiers with DNN is done using as the input layer the matrix of pixels that compose the image, and as output a set of values that show the confidence of that image belonging to each one of the classes.

The first convolutional neural networks for image classification was the LeNet model, that was introduced in 1989 [2]. This was a very shallow network, the latter version of this neural network, LeNet-5 was composed of a total of 6 layers (2 convolutional, 2 average pooling and 2 fully-connected) [3]. This network was designed to classify grayscale images of handwritten digits with a resolution of 32 by 32. This was the first CNN that led to a commercial success, since it was deployed in the ATM machines for recognizing digits in check deposits. LeNet uses as activation the sigmoid function.

AlexNet was the winner of 2012 image classification contest of the ImageNet LSVRC. It has a total of 11 layers (5 convolutional, 3 Max pooling and 3 fully-connected) [4]. It receives as input a 227x227 image with 3 channels one for each RGB color. AlexNet uses a ReLU as activation function decreasing the complexity of the activation function, compared to the sigmoid or tanh function.

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

ResNet introduces even more layers than the previous networks, reaching a total of 152 layers in its bigger version [5]. This deep neural network was the winner of the 2015 ImageNet challenge in the classification task, exceeding for the first time human-level accuracy.

The deeper the network, the harder it is to train it. Deeper networks suffer from vanishing gradient during training, degrading the accuracy of the network [2]. The vanishing gradient problems is caused by the impact of the successive accumulation of several activation functions through back propagation, making it difficult to adjust the weights of the first layers.

ResNet solves the vanishing gradient problem by making use of shortcut connections. The shortcut connections are used to skip some layers, adding up the input of a set of layers directly with those layers output. The ResNet is organized in blocks of layers that are bypassed by shortcuts as showed in figure 1.

2.2.1. Performance comparison.

Table 1 shows a comparison between the different CNN introduced earlier. Showing the Top-5 error rate for each one of the models apart from the LeNet, obtained using the ImageNet dataset.

Table 1 shows that in general deeper networks achieve higher accuracies. From LeNet 5 to AlexNet the number of weights and the number of multiply and accumulate (MAC) operations have increased about a thousand times and two thousand times respectively, in spite of only having doubled the number of layers. The disproportionate increase of computational resources is due to the increase resolution of input image and the increased number of filters per convolutional layer. This table also shows that from VGG 16 to ResNet 50 the number of weights and MACs has fallen despite ResNet 50 having a better accuracy and being a deeper network then VGG 16, this is because the first layers of VGG have a high number of filters.

2.3. Object detection DNN Models

Object detection is used to locate the object position and size, and predict its class [7]. The output of the system is composed of a set of variables that specify the dimensions and location of the box that frames the object, and a set of values that show the confidence of each object belonging to each one of the classes.

One way to architect a object detection system, is to divide it in several stages. Region-based Convolutional Network (R-CNN) consists of three modules [8]. The first module generate category independent region proposals of different sizes. The second module is a deep convolutional neural network that extracts from a region of an image a feature map. The third module is composed by a set of support vector machines (SVM) [9] that classify the image using the feature map. Each one of the proposed regions by the first module are independently fed to the second module to produce a different feature map for each one of the regions. R-CNN is a hybrid solution between CNN and traditional algorithms, being the second module the only module with a CNN.

Fast R-CNN improves both the accuracy and the computational resources used by the model. This model uses the entire input image to extract the features and then uses the region proposals to extract from the entire image feature map a subset of features corresponding to that region. By sharing the feature map the computational operations can be greatly reduced.

Faster R-CNN is similar to the Fast R-CNN being the main difference, the way that region proposals are computed. Faster R-CNN uses a separate CNN to compute the region proposals, instead of the time consuming selective search algorithm used in predecessors models.

You Only Look Once (YOLO) approaches object detection in a different way, using a single CNN to both locate and classify objects in an image [10].

RetinaNet, just like YOLO uses a single DNN to both classify and locate objects. Most of the layers of this model are convolution layers and no fully connected layers are used [11]. This DNN is grouped in smaller sub-networks:

- ResNet backbone — This is where the input image is received. This DNN is composed by 5 stages that output smaller and smaller feature maps as the stage number increases.
- Feature Pyramid Network (FPN) [12] — This DNN enhance the capability to detected small objects [12].
- Classification — There is a network of this type attached to each one of the stage outputs of the FPN as shown in figure 2. In this way the classification

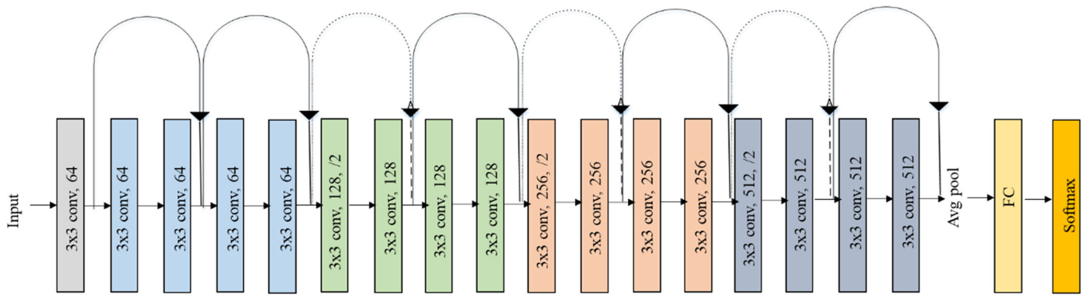

Figure 1. Representation of ResNet-18 architecture [6].

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

TABLE 1. Comparison of popular DNN's with the Top-5 error rate obtained with ImageNet dataset.

sub-networks receives input feature maps of different dimensions, each one being prone to detect objects of different sizes. These subnetworks are composed by 5 convolution layers that classify the objects. The output of each one of the subnetworks are concatenated to achieve a final result with objects of different sizes. The weighs and bias are shared among all the classification sub-networks.

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

2.3.1. Performance comparison. Along with the mAP metric in table 2 it is showed the measured inference time for each one of the models, apart from the R-CNN family models, when executing these models in a Nvidia M40 GPU.

By the performance results in table 2 it can be concluded that as the accuracy increases the inference time that it takes for the model to make its predictions also increases. YOLO is a fast model and ideal for real time applications with strict temporal restrictions, but its biggest implementation has an mAP of 4.8% lower than the biggest implementation of RetinaNet showed in this table.

The chosen object detection network to be implemented is RetinaNet-18 (RetinaNet with a ResNet-18 backbone), due to the high accuracy and high inference times that show potential for hardware acceleration as seen in table 2. To facilitate an embedded implementation the RetinaNet version chosen uses the smaller available backbone for this DNN, ResNet-18.

3. DNN implementations on FPGAs

Deep Neural Networks present several opportunities to explore parallelism and pipelining, to maximize the throughput of the available hardware. A convolution layer can be seen as a set of nested loops that can be computed with the algorithm in figure 3 which is based in equation 1. A technique called spacial unrolling can be used, which is the hardware equivalent of loop unrolling in software, where a certain loop is eliminated or shortened by making those operations in parallel in different hardware rather than sequentially [14].

The basic component of a DNN hardware accelerator is the processing element (PE), which performs the computation for the most important layers, such as the convolutional and fully-connected layers [2]. The accelerator must have several PEs to make it possible to compute operations in parallel. The composition of a PE varies with the implemented architecture and one or more multiply and accumulate (MAC) units may be available per PE.

3.1. Data Quantization

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

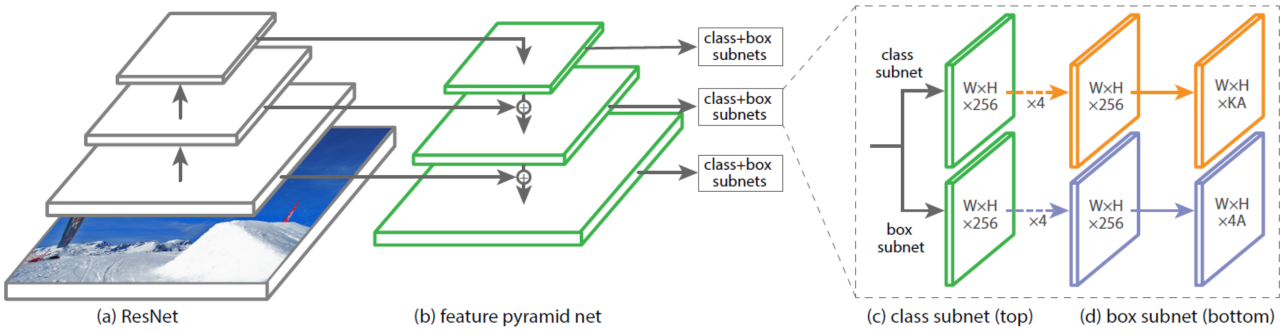

Figure 2. RetinaNet algorithm architecture, consisting of a feature pyramid network and fully connected sub-networks for classifying and produce bounding box localization [11].

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

TABLE 2. COMPARISON OF POPULAR OBJECT DETECTION MODELS WITH MAP OBTAINED USING THE COCO DATASET [11] [13].

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

Figure 3. Algorithm for computing convolutional layers, composed by nested loops, adapted from [14].

the accuracy of the model, thus being important to study the best trade-off.

Existing works study the trade-off between different quantizations and accuracy of the network, compared to the model implemented using 32 bit floating point. Figure 4 shows an existing work that evaluated the impact of quantization in a DarkNet DNN (the backbone of YOLO), trained with the ImageNet dataset [15]. The number of filters in each convolutional layer of DarkNet was multiplied by a scaling factor in order to expand or shrink the model, obtaining implementations with different accuracies and throughputs to evaluate the accuracy/throughput trade-off at different DNN sizes.

The results in figure 4, indicate that both binary and INT2 data types cannot achieve comparable accuracy to the

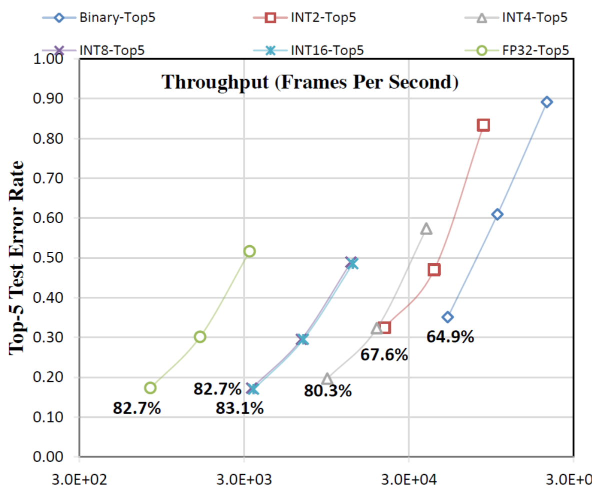

Figure 4. Trade-offs for different quantization techniques using DarkNet as DNN model [15].

other data types. 32 bit floating point (FP32) does not show better accuracy than INT8 and INT16. In addition FP32 has a lower throughput, not having any advantage over the INT8 and INT16 data types. This results shows the big opportunity that quantization brings by reducing resources spent and improving the throughput without damaging the accuracy of the model.

In the work of [15] the quantization strategy was symmetrically used for activations and weights, using in each implementation the same quantization for all data. Other studies have exploited different quantizations for activations and weights and also different quantizations along the depth of the DNN model.

Quantization aware training is a technique used to minimize these accuracies drops. This technique takes into account in the training process the quantization errors produced by quantizing weights and activations, so that the produced weights are already tweaked to minimize the quantization error [16].

4. RetinaNet embedded software implementation

The first step in this work is to train an object detection DNN. RetinaNet-18, was the chosen deep neural network for this work.

4.1. Quantization aware training

A quantization aware training is used to quantize data in this work in order to reduce as much as possible the accuracy loss in the quantization process. This work uses Brevitas python library to make the quantization aware training [17]. Different quantizations were evaluated by changing the bit width of weights in different parts of the DNN and comparing its impact on the overall accuracy of the network. The terminology used to refer to each one of the quantized models is: RetinaNet_*b1_b2_b3* where:

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

All activations in all layers are quantized using 8 bits.

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

The results in table 3 show that the RetinaNet that uses 8 bits in all weights (RetinaNet_8_8_8) is able to achieve the same mAP as the non-quantized RetinaNet FP, which demonstrates the success of quantization aware training and indicates that there is no need to try a higher bit width quantization than 8 bits.

Table 3 shows a more significant reduction in memory usage than in accuracy when reducing the bit width of weights. The quantization with the biggest reductions in accuracy correspond to the ones that use less memory space. For example the quantization of ResNet backbone with 4 bits weights has the biggest drop in accuracy but at the same time introduces the biggest reduction in the size of weights, since it is accounted with 52 % of the total convolution weights in RetinaNet-18. The weights size is reduced by half when using 4 bits weights across the whole network instead of 8 bits weights.

The chosen quantization for the following work was RetinaNet_4_4_4. The 8.9 % drop in accuracy of this quantization in comparison with the original RetinaNet without quantization does not compromise the practical results, producing as output similar bounded boxes and classifications. The accuracy of 23.4 % of this quantization model exceeds the accuracy of YOLO v2 floating-point implementation that has a similar size backbone as shown in table 2. The reduction in weight size enables less resource utilization and the possibility to choose a low cost target device.

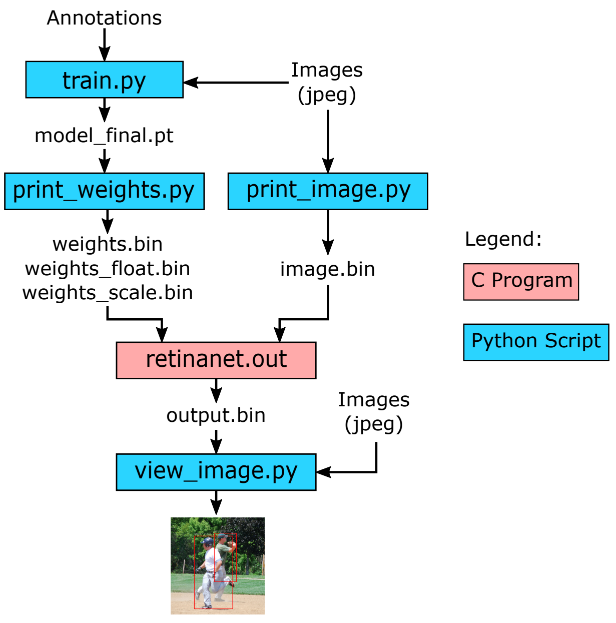

Figure 5. RetinaNet usage workflow.

4.2. Baseline implementation of RetinaNet in C

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

Figure 5 shows the workflow used to make an inference using the C program. The training script produces a file called *model_final.pt* that stores all the trained model parameters including the weights and bias values. The *model_final.pt* is parsed by the python script *print_weights.py*, whose propose is to produce three binary files storing the weights and bias. The image to be subjected to inference has to be decoded from the jpeg format and normalized first, this is made using the python script *print_image.py*.

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

TABLE 3. Comparison of trade-offs between different quantization models.

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

Functions were created for convolution, batch normalization, ReLU, maxpool, sigmoid, upsample and matrix sum layers. A set of nested loops are used to implement the convolution like the algorithm shown in figure 3.

The C implementation of RetinaNet was run on a PC using an Intel I7-5700HQ CPU, in order to make a time profiling of the DNN, surveying which part has the most impact in the execution time of the DNN. The time profiling results, in table 4, show that regression and classification sub-networks account for 27.3% and 48.9% of the DNN execution time respectively.

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

The only part to be deployed to the embedded system is the one to be accelerated in the hardware architecture, this is the regression and convolution sub-networks, this is done to ease up development by reducing the execution times it takes to run on an embedded target. In this way the input of the embedded implementation is the output from the FPN.

In total the embedded systems take about 40 minutes to execute the regression and convolution sub-networks of RetinaNet. A single convolution layer with 80×80×256 dimensions take about 186 s to execute. The results of the embedded software implementation of RetinaNet are used as baseline to compare with the hardware/software implementation acquired in the end of this work.

5. Convolution accelerator hardware architecture

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

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

TABLE 4. Time profiling of RetinaNet DNN.

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

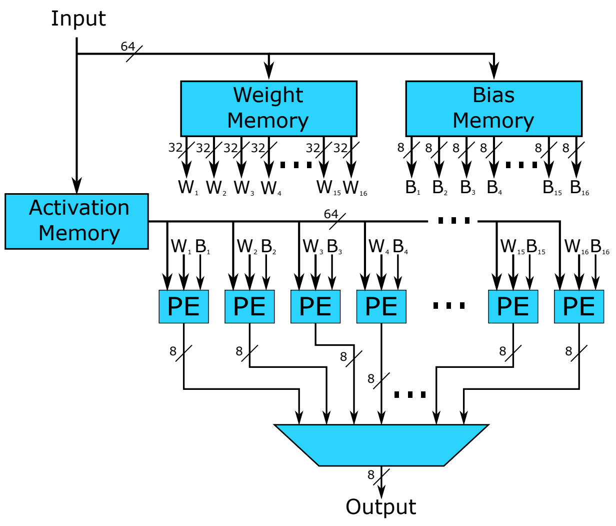

Figure 6. General overview of the accelerator hardware architecture.

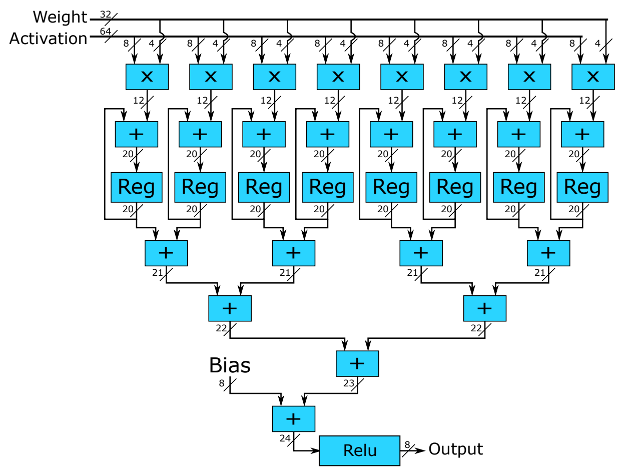

Figure 7. PE hardware architecture.

activation is transferred only once from the external memory.

Figure 6 presents an overall overview of the developed hardware architecture. The main data input of the hardware architecture is an AXI-Stream port, that is connected to activation, weight and bias memories. These three memories are used to feed the processing elements (PE) with the data to be computed. The output of each PE is multiplex, which makes it possible to use only one AXI-Stream output port for all the PE's.

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

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

Each processing element is scheduled to compute a different output feature map in a given time, this means that every PE is doing a different output activation in the z dimension at the same time. However, all PEs are processing pixels with the same x and y dimensions in a given time. With this parallelization strategy the same activation values may be broadcast to every PE as seen in figure 6, since the same set of input activations are used for output pixels with the same x and y coordinates. The broadcast of activations reduces the bandwidth requirements of the activation memory and the complexity of the architecture.

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

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

5.1. Processing Elements

The processing elements (PE) are composed of 8 multiply and accumulate operators (MAC) that make the multiplications between activations and weights, as described in more detail in subsection 5.1.1. Those 8 MACs are then connected to a sum tree to obtain the final result for the output pixel calculated by the PE, as described in more detail in subsection 5.1.2. The PE output is quantized to a bit width of 8, as described in more detail in subsection 5.1.3. All the components and connections that compose the PE are shown in figure 7.

5.1.1. Multiply and accumulate operators. Each output pixel of a convolution is a sum of multiplications of weights with activations as shown in equation 1. In this way, multiply and accumulate operator (MAC) is ideal for this application.

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

The activation and weight memories are arranged in a ZXY format and since every MAC is processing contiguous values in the z dimension, it is possible to receive a chunk of data from the activation memory containing 8 activations and a chunk of data from the weight memory containing 8 weights. Figure 7 shows the input activation bus with a bit width of 64 bits, because it contains 8 activations, these 8 activations are then separated for the different MACs. Similarly, the input weight bus has a bit width of 32 bits, because it contains 8 weights that are also separated for each one of the MACs.

5.1.2. Sum tree. After all the multiplications between weights and activations are concluded for a given output pixel, all the MAC results are added in a sum tree, as shown in 7. The sum tree result is then added with the bias used in that output feature map, this adder result is then used as input in a quantizer.

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

5.2. Activation memory

The adopted strategy was to store in local memory only enough activations so that each activation is only transferred once from the external memory. It was decided to store all the activations across the z axis together because to determine a single output pixel all the activations across that axis are needed.

The activation values were stored in the different types of memory shown in figure 8. The input activations are stored in shift-registers accordingly to their coordinates if they are part of the calculations of the output pixel being determined at that moment. In case the activation values are in local memory waiting to be used later they are stored in FIFOs. The way the FIFOs and shift-registers are connected with each other, as shown in figure 8, is related to the way that activations flow from being actively used in a calculation of an output pixel and are out on hold to be used later. A multiplexer is used to select which one of the shift registers would output its value to be used in the PEs, as shown in figure 8.

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

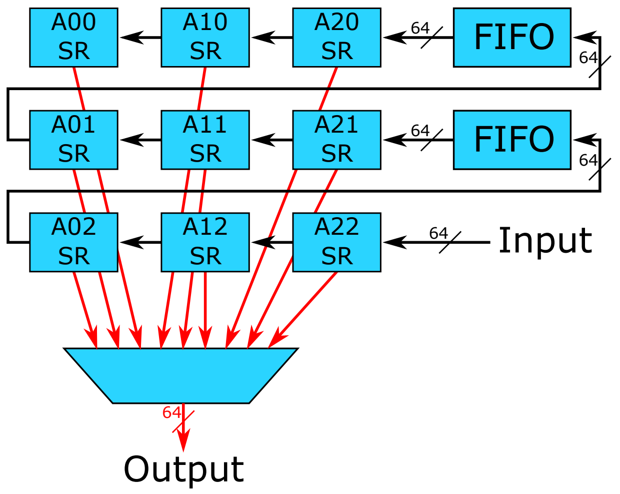

Figure 8. Activation memory hardware architecture. Arrows in black show the loading path of activations. Arrows in red show the reading path of activations

5.3. Weight memory

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

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

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

6. RetinaNet hardware/software implementation

The hardware/software architecture run some layers in the embedded ARM processor and others in the convolution hardware accelerator described in section 5. All the convolution layers in the implemented part of RetinaNet, apart from the ones that were not quantized, are processed in the convolution hardware accelerator, which accounts for 40 layers of the total of 50 layers. The sigmoid layers, the input quantization layers and the data conversions and arrangements functions are processed in the ARM processor.

The hardware architecture is composed by two parts: the programmable logic (PL), this is all the components that are

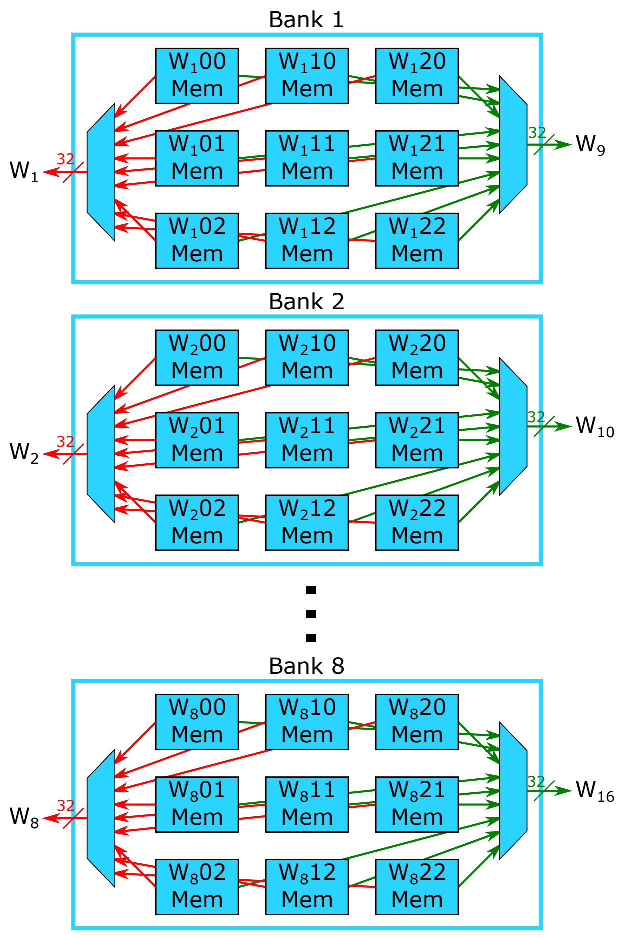

Figure 9. Weight memory hardware architecture. Arrows in different colors correspond to different ports of the dual port memories.

implemented using the FPGA resources; and the processing system (PS), this is the ARM processor and some related peripherals.

The PS uses an AXI-Lite interface to connect to an AXI interconnect IP, this IP splits the AXI-Lite interface to the convolution hardware accelerator and the DMA IP, enabling one single interface to communicate with to different hardware components. The PS sends through the AXI-Lite interface to the convolution accelerator, control data, specifying several characteristics of the convolution layer being executed in the accelerator. The PS also uses the same AXI-Lite interface to control the DMA, controlling the content that is being transferred to and from the convolution accelerator at any given time.

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

6.1. Experimental results

Table 5 show the resource usage of the RetinaNet hardware/software implementation. Most of the resources are used by the convolution accelerator. All the DSPs used are in this component, and 89 of the 92 BRAMs used are in the accelerator.

The high LUT usage of the system makes it harder to optimize the signal paths ending up with long paths between registers. This issue make it difficult to achieve high clock frequencies. A clock period of 10 ns was attempted, however the big route delays made it unsuccessful.

The achieved clock period was 14 ns, having a worst setup slack of 0.038 ns. The critical path is in the data path of the convolution accelerator between a flip-flop and a DSP register. The route delay accounts for 89 % of the total delay of this path.

Table 6 show the time profiling of the hardware/software implementation together with the software only implementation.

The layers with an input activation matrix of $80 \times 80 \times 256$ dimension take about 0.519 s to execute. Comparing the execution of a single layer between both software only and hardware/software implementation a speedup of 359 is achieved. The total speedup achieved by the HW/SW implementation is 4.6, since one of the layers in each stage in not executed in the convolution accelerator. In the HW/SW implementation the time taken by the convolution that is processed in software accounts for 87 % of time taken by the regression stage where it belongs. This number is even higher for convolution stages, the software processed convolution accounts for 98 % of the total time of the stage. There is a great potential to achieve higher speedups by accelerating more layers.

7. Conclusion

The chosen object detection DNN to be accelerated in this work was RetinaNet-18. This network is composed by a total of 174 layers of which 78 are convolution layers. This DNN is known for high-end accuracy results, however due to this network extensive size it has above average inference latencies. The objective of this work was reducing the inference latencies without damaging a lot the accuracy of the network in an embedded environment.

A study with different quantizations was made to determine the right trade-off between accuracy and memory usage. The chosen quantization of 4 bits weights and 8 bits activations and bias not only achieved better results than competitive DNNs with similar sizes, but also occupies 8 times less memory for the storage of the weights.

A convolution hardware accelerator was created, with 16 PEs, each one capable of doing 8 MAC operations in parallel, which make it possible to achieve 128 MAC operations per clock cycle.

The convolution hardware accelerator was integrated in a co-designed hardware software implementation. This system achieved a speedup of 4.6 compared to the baseline implementation. This results also showed that there is a speedup of 359 comparing a hardware processed layer with a software processed layer.

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

TABLE 5. Resource usage of RetinaNet hardware/software implementation.

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

TABLE 6. Time profiling of RetinaNet hardware/software implementation and comparison with software only implementation.

7.1. Future Work

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

References

[1] J. Walsh, N. O' Mahony, S. Campbell, A. Carvalho, L. Krpalkova, G. Velasco-Hernandez, S. Harapanahalli, and D. Riordan, "Deep learning vs. traditional computer vision," 04 2019.

[2] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec 2017.

[3] 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, Nov 1998.

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

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[6] F. Ramzan, M. U. G. Khan, A. Rehmat, S. Iqbal, T. Saba, A. Rehman, and Z. Mehmood, "A deep learning approach for automated diagnosis and multi-class classification of alzheimer's disease stages using resting-state fmri and residual neural networks," *Journal of Medical Systems*, vol. 44, p. 37, 12 2019.

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

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

[9] T. Evgeniou and M. Pontil, "Support vector machines: Theory and applications," vol. 2049, 01 2001, pp. 249–257.

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

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

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

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

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

[15] J. Su, "Artificial neural networks acceleration on field-programmable gate arrays considering model redundancy," Ph.D. dissertation, Imperial College London, 2018.

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

[17] A. Pappalardo, "Xilinx/brevitas." [Online]. Available: https://doi.org/10.5281/zenodo.3333552