SoC-FPGA MobileNets for Embedded Vision Applications

Tiago De Smet
tiago.smet@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal
September 2021

***Abstract*—Nowadays, the use of artificial intelligence in many software applications is increasingly common. However, decades of development were necessary, especially at hardware level for the use of artificial intelligence to become viable. MobileNets model developed by Google researchers is used for image classification and primarily suited for embedded systems because of the lighter computation compared to its competitors. This model uses the depthwise separable convolution concept to perform the convolutions and despite being a very optimized model, the processing time of this network may still be high when used on low-cost devices.**

This work aimed to develop a hardware/software multiprocessing architecture in an SoC FPGA platform for image classification using MobileNets model. The main contributions of this project are the development of 3 IPs to process the depthwise separable convolution layers, using an effective parallelization and allocation of resources to achieve an efficient multi-processing, and a quantized data model to reduce the memory requirements and improve the communications.

The system was implemented on a Zynq 7010 device using a quantized MobileNets model with 26% of the parameters represented in a 16-bit fixed-point format and 74% of the parameters using a 12-bit custom floating-point representation. This quantization process produced a model 60% smaller than the standard MobileNets with only 0.78% of accuracy loss. The final solution allows the classification of 1 image in 469 ms which corresponds to a speed up of 11 times compared to a software-only solution executing on the embedded ARM processor.

***Index Terms*—Artificial intelligence, Google, Depthwise Separable Convolution, MobileNets, FPGA.**

I. Introduction

Image recognition is an easy task for humans but it has proved to be a complex problem for machines to perform due to the computational effort involved. The evolution of high-capacity computers and new artificial intelligence (AI) techniques has generated an interest in object classification algorithms. Computer image classification uses a form of AI denominated machine learning, which uses a variety of algorithms that iteratively learn with the available data. This iterative learning is also called training, and is done to create a model to make predictions or decisions without being programmed for that. Specific models can be trained to be used on a wide range of applications such as object detection, medical diagnostic, voice processing, biometry, and many others.

The trained model receives an input (an image for instance) and gives an output (what the image represents), which sounds simple, but in reality, is a very complex process. To perform this classification, these models use the concept of network [1]. A complex system can be broken into simpler elements to be easier to solve, or vice versa, simple elements can be combined to build a complex system, and networks can be used to accomplish this. These networks are composed of a set of nodes and connections between them, used to transfer information. This way, nodes receive inputs and process them to obtain an output, which can be transferred to other nodes.

Nodes can be seen as artificial neurons by the networks, and in this case, the network is defined as an artificial neural network (ANN) [1]. An artificial neuron is inspired by the natural neuron functionality, which receives signals through synapses. When one of these received signals is strong enough, the neuron is activated and also emits a signal, that might be propagated to other neurons and even activate some of them.

The class of ANN covers several architectures, and one very popular kind of ANN used in image recognition is the convolutional neural network (CNN) [2]. CNNs are designed having in mind that the inputs will be images, which allows making the network more adapted to image tasks while reducing the parameters required. Despite this, CNN models have intensive computing and not all are efficient to be used in mobile or embedded devices. Some CNN models have been created to deal with this constraint, which is smaller and lighter than common CNNs, and have a decent accuracy in image classification. One of these models is MobileNets [3] which proposes to reduce the computational effort and resource requirements of standard CNNs, by reducing the number of parameters, allowing the model to be used in mobile and embedded vision applications.

Even with a smaller and lighter model, the implementation on mobile or embedded devices may not be easy. Moreover, some implementations need to process an image in hundreds or tens of milliseconds, which may require very expensive devices. Thus, an efficient and optimized architecture can help to accelerate the model with fewer resources, which represents less power consumption or even cheaper devices.

This work proposes two approaches to accelerate the MobileNets model in a low-cost device. Firstly, a quantization analysis of the model is conducted to evaluate the behaviour when the weights and activations are quantized using different representations. Quantizing the model allows reducing the memory requirements and improve communications. The second approach is to implement a hardware/software architecture capable to execute the inference process of the quantized MobleNets model, using an effective parallelization and allocation of resources. The final system should be able to classify the image in less than 1 second, and possible

optimizations of the MobileNets model, should not incur a loss in accuracy larger than 1% over the original model.

II. Background

This section presents the basic structure of CNNs, and describes the MobileNets architecture in detail.

A. Convolutional Neural Networks

CNNs are primarily suited for image-focused tasks like recognition or classification of images. These networks connect only a small region (receptive field) of the input to the neuron and are organized into three dimensions: the height and width that compose the spatial dimensionality of the input, and the depth, which represents the third dimension of an output map. CNN architectures have three types of layers: convolutional layers, pooling layers and fully connected (FC) layers. There are several types of CNNs models, and their use can be applied to different purposes and hardware platforms. One example is the MobileNets [3] model designed for image classification and used mostly on mobile and embedded devices.

B. MobileNets

MobileNets [3] is based on depthwise separable convolutions which factorize a standard convolution (figure 1) into a depthwise convolution and a pointwise convolution.

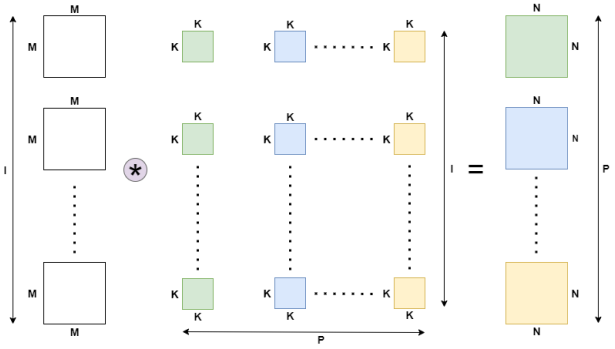

Figure 1: Standard convolution.

The depthwise convolution applies a single $k \times k$ kernel to each input channel (figure 2). Considering K as the height/length size of the kernel, N as the height/length size of the output feature maps (OFMs) and I as the number of input feature maps (IFMs), the total number of multiplication and accumulation (MAC) operations of a depthwise layer is given by:

$$Depthwise\ MAC\ op. = K \cdot K \cdot N \cdot N \cdot I \qquad (1)$$

The following pointwise convolution applies a $1 \times 1 \times I$ filter to linearly combine the output of the depthwise convolution (figure 3). Again, considering K as the height/length size of the kernel, N as the height/length size of the OFMs, I as the number of IFMs, and P as the number of filters, the total number of MAC operations of a pointwise convolution is given by:

$$Pointwise\ MAC\ op. = K \cdot K \cdot I \cdot N \cdot N \cdot P \qquad (2)$$

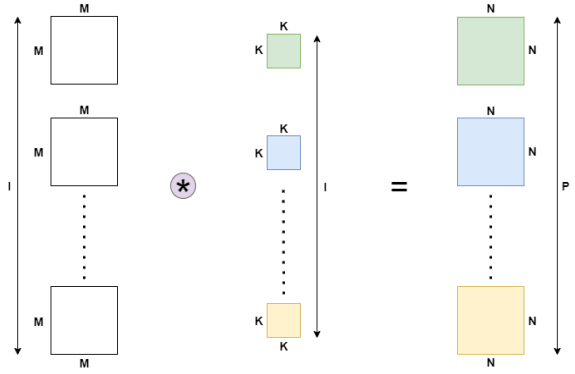

Figure 2: Depthwise convolution

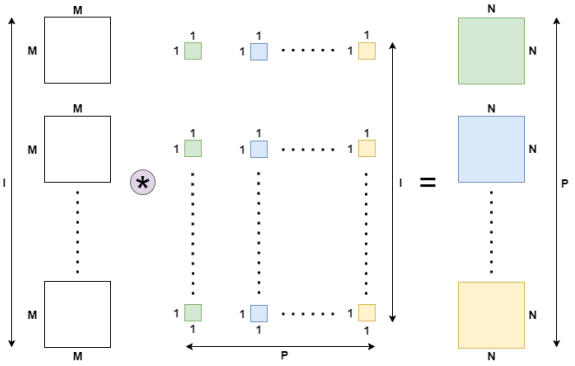

Figure 3: Pointwise convolution

The complete model of MobileNets is composed of 30 layers (table 1). To avoid losing too much information at the beginning, the first layer is a standard convolution, which receives the input image. Then, the next 26 layers are depthwise and pointwise convolutions, arranged in an interleaved way, which perform the feature maps. The last 3 layers are a pooling layer, to select the average value of the previous feature map; an FC layer that classifies the input image according to the available labels; and a SoftMax function to convert the values received from the FC layer to probabilities. All the layers are followed by Batch Normalization (Batch Norm) [4] and Rectified Linear Unit (ReLU) [5], except for the last 3 layers. Thus, from here, the set of one layer followed by Batch Norm and ReLU will be referred to a stage.

Batch Norm is given by algorithm 1, which normalizes the input layer by adjusting and scaling the activations. The values of variables μ_β, $\sigma^2{}_\beta$, γ and β are parameters from the MobileNets model, and for ϵ, the value used in this work is 0.001, the same as in Tensorflow [6]. Algorithm 1 can be rearranged and written as

$$y_i \leftarrow \gamma \frac{x_i - \mu_\beta}{\sqrt{\sigma^2{}_\beta + \epsilon}} + \beta = \gamma(x_i - \mu_\beta)V + \beta \qquad (3)$$

Considering N as the height/length size of the OFM and P as the number of OFMs, the total Batch Norm operations can be calculated by:

$$Batch\ Norm\ op. = N \times N \times P \times 4 \qquad (4)$$

Table 1: MobileNets model architecture and the respective number of operations for each convolutional, pooling, FC and SoftMax layer.

Type/Stride	Filter shape	Total filters	IFM size	Total IFMs	Total operations
3D Conv/2	3 × 3 × 3	32	224 × 224	3	21,676,032
DW / 1	3 × 3	32	112 × 112	32	7,225,344
PW / 1	1 × 1 × 32	64	112 × 112	32	51,380,224
DW / 2	3 × 3	64	112 × 112	64	3,612,672
PW / 1	1 × 1 × 64	128	56 × 56	64	51,380,224
DW / 1	3 × 3	128	56 × 56	128	7,225,344
PW / 1	1 × 1 × 128	128	56 × 56	128	102,760,448
DW / 2	3 × 3	128	56 × 56	128	1,806,336
PW / 1	1 × 1 × 128	256	28 × 28	128	51,380,224
DW / 1	3 × 3	256	28 × 28	256	3,612,672
PW / 1	1 × 1 × 256	256	28 × 28	256	102,760,448
DW / 2	3 × 3	256	28 × 28	256	903,168
PW / 1	1 × 1 × 256	512	14 × 14	256	51,380,224
5x DW / 1	3 × 3	512	14 × 14	512	1,806,336
PW / 1	1 × 1 × 512	512	14 × 14	512	102,760,448
DW / 2	3 × 3	512	14 × 14	512	451,584
PW / 1	1 × 1 × 512	1,024	7 × 7	512	51,380,224
DW / 1	3 × 3	1,024	7 × 7	1,024	903,168
PW / 1	1 × 1 × 1,024	1,024	7 × 7	1,024	102,760,448
Avg Pool / 1	7 × 7	1	7 × 7	1,024	51,200
FC / 1	1 x 1 x 1,024	1,000	1 × 1	1,024	2,049,000
SoftMax / 1	Classifier	0	1 × 1	1,000	4,000

Algorithm 1 - Batch Norm transform (adapted from [4]).

Input: Values of x over a mini-batch: $B = \{x1 \dots m\}$;
Parameters to be learned: γ, β
Output: $\{y_i = BN_{\gamma,\beta}(x_i)\}$

$$\mu_\beta \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2{}_\beta \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\beta)^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x_i} \leftarrow \frac{x_i - \mu_\beta}{\sqrt{\sigma^2{}_\beta + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x_i} + \beta \equiv BN_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

ReLU6 is the activation function that preserves the values between 0 and 6, assign 0 to all negative values and 6 to all values bigger than 6. Since the number of operations (comparisons) of this function depends on the input value, the total number of operations in the worst case (the process performs two comparisons) is given by:

$$ReLU6\ op. = N \times N \times P \times 2 \qquad (11)$$

III. CNN MODELS ON FPGAS

CNN models can be compressed in order to reduce the use of memory, communications and operations while minimizing accuracy loss. From the hardware perspective, specific architecture modules are designed to reuse data, accelerate convolution operations, and efficiently use all available resources. Also, when porting a CNN model to an FPGA device, the bit widths of operators and weights are often reduced.

A. CNN model optimization

A common strategy to optimize CNN models is quantization [7] [8] [9]. Data quantization defines how data values are represented and how many bits are used. The use of short fixed-point numbers is sufficient for CNN inference, but not for training. CNN models can be quantized using one of the two strategies described in [7]. The first strategy is post-training quantization, which simply converts the 32 bits floating-point parameters of the model usually to 16 or 8 bits fixed-point because these are the minimum values that can keep an acceptable accuracy of the model. The other is quantized aware training, where the model is normalized and converted from 32 bits floating-point to 8 bits fixed-point and then retrained. In this situation, since the network is retrained, the use of 8 bits is usually sufficient to obtain an accuracy near to the original, or in some cases, even better.

For the particular case of MobileNets, this model is commonly quantized using 8-bit fixed-point and then retrained [8] [9]. Three contributions to improve the latency-vs-accuracy trade-off of MobileNets on common mobile hardware are presented in [8], while [9] proposes rearranging the MobileNets architecture to become more quantization friendly, namely the depthwise separable convolution layers, which the authors state are the causes of large quantization loss.

Work [10] combines pruning and quantization during the training of MobileNets. First, a quantization training process is conducted, followed by an iterative pruning and retraining process. In each iteration, the number of filters is reduced resulting in fewer OFMs (figure 2 and figure 3), and in a smaller memory requirement. However, in this strategy, the filters are removed or kept as a whole, according to the summation of its values, instead of making them sparse. Although only less important filters are removed, some information is lost, so the model is retrained to compensate for the accuracy loss. During the execution of the algorithm, in forward propagation, weights W and activations a are quantized before actual computations during inference. Then the real values are converted to a pre-defined fixed-point representation. In backward propagation, the updating is applied to the real-valued weights W rather than their quantized alternatives W^Q, which keeps a higher precision during training.

B. *Hardware design: parallelism*

CNN computation can take advantage of different parallelization methods to accelerate the inference process. The design of CNN architectures mainly explores three types of parallelism [11]: Operator-level (fine-grained): in the convolution of a single input image of size $m \times m$ with a kernel of size $k \times k$, each output pixel requires $k \times k$ MACs operations, which can be executed in parallel. Intra-output (Coarse-grained): the computation of each pixel in a single OFM can be done in parallel since it is the sum of independent input-kernel convolutions. Inter-output (Coarse-grained): multiple OFMs can be computed in parallel by multiple processing elements (PEs).

C. *Hardware design: system architecture*

Commonly, FPGA implementations have on-chip memory and off-chip memory. All information can only be stored on the FPGA before computation begins if the CNN model is small enough to fit on the available on-chip memory [10]. Thus, the most common CNN accelerators read or write information on the external memory during computation and use on-chip buffers to save intermediate results [12].

In work [10], all the MobileNets parameters are transferred from the external memory to on-chip buffers (figure 4).

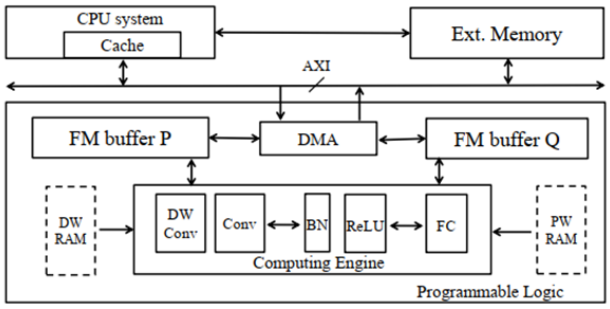

Figure 4: System Architecture Design for RR-MobileNets (from [10]).

The input images are stored in the FM buffer P and the network parameters in the DW RAM and the PW RAM. When all the initial data are stored in these buffers, the computation begins. Each layer is processed one at a time in the computation engine, which receives and stores the intermediate feature maps in the FM buffer P and the FM buffer Q, in an alternative manner for consecutive layers. The DW Conv module of the computation engine processes the depthwise convolutions, while the standard and pointwise convolutions are processed in the Conv module. BN and ReLU modules are common to these two convolution modules. The connection of the DW Conv, BN and ReLU modules forms a set of 32 PEs capable of processing 32 OFMs in parallel (inter-output parallelism). After processing the depthwise convolution, the BN module takes the outputs from the DW Conv and applies multiplication and addition for scaling operations. Then ReLU simply caps negative input values with zero. Similarly, the connection of the Conv module with BN and ReLU, also corresponds to a set of 32 PEs working in parallel. Lastly, the FC module is composed essentially of parallel multipliers and adder trees to execute de FC layer.

A variant of MobileNets [13] is used in work [12]. In this architecture, a matrix multiplication engine (MME) conducts all the CNN operations (figure 5). All the input images and parameters are stored on the external memory. To avoid excessive latency, a ping-pong weight buffer is placed between the MME array and the external memory to maximize bandwidth. The line buffers are connected to a multiplier array with 288 multipliers (32 slices with 9 multipliers) configurable to perform standard, depthwise or pointwise convolutions.

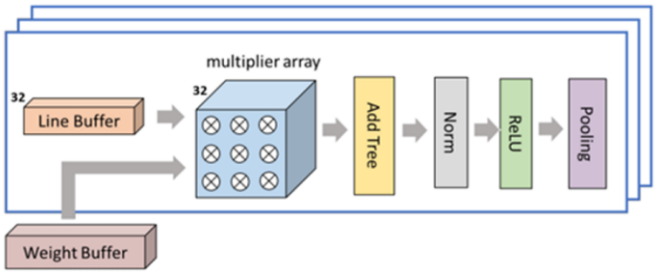

Figure 5: Block diagram of an MME (from [12]).

To perform a standard convolution, the line buffer is configured to input 3 IFMs, and the multiplier array performs the multiplication of these IFMs with the respective weights. Only the results from the 3 slices from the multiplier array connected to the 3 line buffers will be considered in this situation. After the multiplication in the multiplier array, the products are summed in an adder tree, configurable to perform depthwise or pointwise summing operation. When the depthwise convolution mode is selected, the adder tree sums up the products from each slice of the multiplier array in parallel. For one MME the maximum output channels number is 32. Pointwise convolution uses the divide and conquer algorithm in large matrix multiplication to divide the IFM into $M \times M \times 32$ submatrices, which are shifted into line buffers. This allows one PE to perform a maximum of $M \times M \times 32$ and 32×9 multiplications at once. The 32 products are then summed, and the output channels number is 9. The following steps perform the normalization, pooling and ReLU. ReLU has three selectable options: no ReLU, ReLU or ReLU6. Lastly, the pooling module can select between max and average pooling.

D. *Hardware design: tilling and reuse*

Due to the limited on-chip memory, some implementations use tilling and reuse of memory blocks [10]. Tilling divides the IFM into smaller groups of pixels and sends these groups to the PE. This allows performing the convolution by parts when there is not enough memory to store all the IFMs at once. Since some pixels are needed more than one time and to avoid repeating data communications, some IFM pixels are stored on the PE to be reused when needed again.

IV. MobileNets embedded software design

This section presents a study of the model to evaluate the number of parameters and operations necessary to each layer type, and for Batch Norm and ReLU6 functions. Also, the execution time of each layer type and activation function is obtained from the developed C program. Lastly, an analysis of fixed- and floating-point quantization schemes is performed to determine the most efficient for this project.

Table 2: Total parameters, operations and execution time by layer and function type of the MobileNets standard model.

Layer/function type	Total parameters	Total parameters (%)	Total operations	Total operations (%)	Total exec. time (ms)	Total exec. time (%)
Standard conv.	864	0.02	21,676,032	1.86	100.6	1.91
Depthwise conv.	44,640	1.05	34,771,968	2.98	228.6	4.35
Pointwise conv.	3,139,584	73.80	1,078,984,704	92.39	4636.5	88.17
Pooling	0	0.00	51,200	0.00	0.4	0.01
Fully Connected	1,025,000	24.10	2,049,000	0.18	9.0	0.17
SoftMax	0	0.00	-	-	1.7	0.03
Batch Norm	43,776	1.03	20,170,752	1.73	171.4	3.26
ReLU	0	0.00	10,085,376	0.86	110.7	2.10
Total	4,253,864	100.00	1,167,789,032	100.00	5,258.8	100.00

A. MobileNets memory requirements, number of operations, and execution time

The MobileNets model was developed using the Keras [14] deep learning framework with Python and TensorFlow backend. This standard MobileNets is a 32-bit floating-point model, with 4,253,864 parameters, that can classify the 1,000 different classes of the ImageNet test dataset with a 70.6% top 1 and 89.9% top 5 accuracies. In this project, the dataset used is the ImageNet validation dataset, the same used by Keras developers, who report 70.4% top 1 and 89.5% top 5 accuracies [15].

Table 2 presents the number of parameters, total operations and execution time for all the convolutional layers, as well as for the SoftMax function, of the MobileNets 32-bit floating-point model. The pointwise convolution represents 92% of the total operations, which consume about 88% of the execution time of the inference process. The Batch Norm and ReLU6 functions combined have almost the same number of operations of the depthwise convolutions, and their execution time combined is more significant than the convolutional layer, which makes these functions also candidates for hardware implementation.

B. Quantization

The model is mostly composed of pointwise parameters, therefore, the focus of the following quantization strategies was in reducing the number of bits used by these parameters to the minimum acceptable, which for this work is a loss of accuracy smaller than 1%. During inference, the OFMs are also represented with the minimum of bits because some of these OFMs will be exchanged between the on-chip and off-chip memory, which means that the fewer bits are used to represent the values of these OFMs, the faster the communications will be, and less on-chip memory will be required to store these values. Since the 3D convolution, depthwise and Batch Norm parameters represent only about 2% of all the model parameters, these will be quantized only to 16 bits. Although FC parameters are about 24% of the model size, these parameters are also quantized to 16 bits because this convolution will be performed on the software application. Two strategies of post-training quantization were developed and tested to evaluate the accuracy and behaviour of the model after quantization: fixed-point quantization of weights and activations and custom floating-point pointwise weights quantization.

The first step of this quantization study was to convert the weights and Batch Norm parameters from 32-bit floating-point to 16 or 8-bit fixed-point, following the practices in state-of-the-art post-training quantization work [7]. Dynamic quantization is used, where the number of integers and fractional bits of each layer is chosen according to the range of its weights and activations. To perform the dynamic quantization, the quantization algorithm first searches the maximum absolute value of the model parameters, for each layer, to detect how many bits need to be used in the fractional part of the number ($Search\_\max$ function). All the layer parameters are quantized with the maximum bits obtained for the fractional part, which will define the representation used for that layer. For batch norm layers, and before the quantization of these parameters, an extra procedure is applied in order to combine the two multiplications in a single one. Thus, equation (3) is reduced to:

$$\gamma(x_i - \mu_\beta)V + \beta = (x_i - \mu_\beta)\gamma V + \beta = (x_i - \mu_\beta)P + \beta \qquad (12)$$

where P is the multiplication of the gamma and variance parameters of the activation layer before the quantization process. Applying this procedure also represents a reduction of 25% in the Batch Norm parameters and therefore in the number of operations of this function. The OFMs of the convolution layers are quantized with either 8 or 16-bits and the appropriate fixed-point scales are selected independently for each layer, following the same approached used for the weights. As the ReLU6 function bounds the values between 0 and 6, the pixels of the OFMs can be represented using Q3.5 (or Q3.13) unsigned. Therefore, after applying the Batch Norm function to the convolutional layer results, the number is aligned with the fractional bits used in ReLU6.

The experiments of the previous strategy showed that when the pointwise weights were quantized with 8 bits, the accuracy dropped roughly between 8 and 10%. To improve the accuracy without using 16 bits overall, a custom floating-point representation is proposed. This representation uses 12 bits distributed as follows: 8 for the quantized number and 4 for the exponent as presented in figure 6.

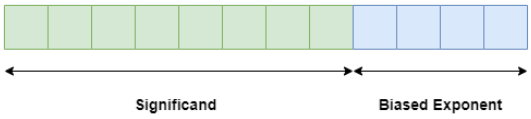

Figure 6: Distribution of the 12 bits of the proposed floating-point.

The real number can be represented by the proposed floating-point using:

$$R = Significand \times 2^{-(Biased\ Exponent + Bias)} \qquad (13)$$

where R is the real value to be quantized and $Bias$ the number of bits of the fractional part of the base representation obtained using the $Search\_\max$ function of the quantization algorithm. Since the Biased Exponent uses 4 bits and takes values between 0 and 15, the Exponent varies between $0 + Bias$ and $15 + Bias$. After obtaining the base representation, the algorithm selects the Exponent that allows each weight to be represented using the maximum significant bits. Using this floating-point representation, the memory requirements for the parameters is reduced in 20% compared to the 16-bit fixed-point (model 2), and 60% compared to the standard model. In terms of hardware requirements, there is only the need to insert a right shift operation after the multiplication, as shown in:

$$Conv_{pointwise} = \sum_{1}^{I} [(Qp_i \times Qw_i) \gg Biased\ Exponent_i] \qquad (14)$$

where Qp_i is the ith quantized pixel, Qw_i is the ith quantized weight from the ith pixel, and $Biased\ Exponent_i$ the Based Exponent from the ith quantized weight.

Table 3 compares the results obtained with the custom floating-point quantization with those obtained with the fixed-point quantization and the standard MobileNets (model 1). Comparing model 9 with model 2 and model 10 with model 4, shows that using the custom floating-point with 12 bits to represent the pointwise parameters practically maintains the same accuracy than using a 16-bit fixed-point representation, while requiring 25% fewer bits. As an example, the last pointwise layer has $1 \times 1 \times 1024 \times 1024 = 1{,}048{,}576$ parameters, which means that to perform the last layer on-chip, roughly 2.1 MB must be transferred to the FPGA local memory, when using a 16-bit fixed point. Using the 12-bit custom floating-point implementation, this value is reduced to 1.6 MB. The last models of table 2 are two variants of the custom floating-point strategy. Model 11 uses 3 bits to represent the exponent (8Exp3), and model 12 uses 2 bits (8Exp2). This reduces the data required to perform the last pointwise layer, which is approximately 1.4MB for model 11 and 1.3MB for model 12, but slightly reduces the accuracy. Models 10, 11 and 12 are the best for this project because they are the smallest models that use dynamic OFM quantization and meet the maximum accuracy loss of 1%. From these three, model 10 is selected because it has a slightly better accuracy (0.2% better than model 12). Also, model 11 uses 3 bits for the exponent, which is somewhat less efficient to group than using multiples powers of two. The result shows that using a custom 12-bit floating-point representation for the pointwise weights parameters achieves a better trade-off between accuracy and model size than the corresponding 16-bit fixed model. This allows having a MobileNets model 60% smaller than the standard with only a very small accuracy reduction of 0.78%.

Figure 7 compares the number of zero values after quantization for each pointwise layer when 16-bit fixed-point and the proposed 12-bit floating-point are used. The conclusion is that the custom floating-point is able to represent more effectively the very small (near zero) weights, as it reduces by almost 14% the number of pointwise parameters equal to zero after quantization.

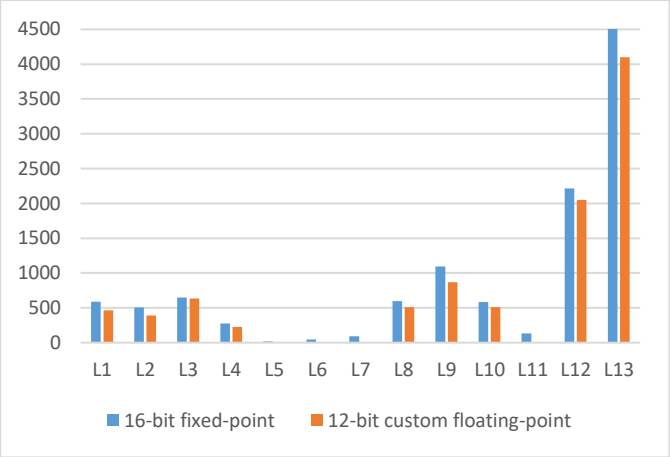

Figure 7: Total number of parameters equal to zero after quantization when 16-bit fixed-point and 12-bit custom floating-point are used.

V. Hardware Design

In this section, all the custom IPs used in this project are described individually, detailing their structure, the functionalities and features used, and explaining the specific design options taken.

A. *MobileNets padding and tiling*

Stages 7 to 13 are completely performed on hardware, however, stages 1 to 6 require a tiling procedure before starting the stage processing and a map reorganization after the stages are completed. These functions and also the half and complete zero-padding are performed on the software application, due to the cost it would bring to the hardware applying this procedure on tiled blocks, incurring only a small penalty in the final execution time of the inference process. After performing the half or complete zero-padding (depending on the stride applied) another function performs a tiling process before sending the IFMs to the PL. After receiving the data processed on the PL, the software application reorganizes the tiled blocks, which is the inverse process of the tiling function. The described tiling process selects a group of $16 \times 16 \times 32$ pixels from the IFMs when the stride is 1 (or $15 \times 15 \times 32$ pixels when the stride is 2). The last column between two tiles is sent twice to the hardware and the same happens with the last rows between two tiles, avoiding the extra complexity of implementing the padding in hardware which does not compensate for the small gain in memory and time communication. Also, since the IFMs are stored on a ping-pong memory, the loss on the communication is reduced as while the depthwise convolution is executing, the other memory is loading the new IFMs.

Table 3: MobileNets standard model vs fixed and floating-point quantization accuracies and sizes.

Model	Input images	3D conv weights	DW weights	DW OFM	PW weights	PW OFM	BN param.	FC IFM	FC weights	Top 1 (%)	Top 5 (%)	Size (MB)
1	32F	32F	32F	32F	32F	32F	32F	32F	32F	70.54	89.58	17.0
2	8	16	16	16	16	16	16	16	16	70.56	89.50	8.5
4	8	16	16	16	16	8	16	8	16	69.77	89.05	8.5
9	8	16	16	16	8Exp4	16	16	16	16	70.50	89.54	6.9
10	8	16	16	16	8Exp4	8	16	8	16	69.76	89.02	6.9
11	8	16	16	16	8Exp3	8	16	8	16	69.72	89.02	6.5
12	8	16	16	16	8Exp2	8	16	8	16	69.56	88.98	6.2

B. *MobileNets hardware implementation*

A simple scheme of the proposed MobileNets hardware design can be observed in figure 8.

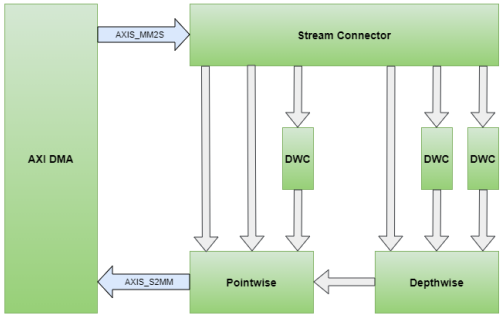

Figure 8:Simple scheme of the proposed MobileNets hardware design.

The first custom IP is the "MobileNets Stream Connector", designed to demultiplex the weights, Batch Norm parameters and IFMs which are stored on BRAMs. These BRAMs are within the "MobileNets Depthwise" and "MobileNets Pointwise" IPs (the other custom IPs) and are configured in simple dual-port mode with different port widths, depending on the data type. The Stream Connector demultiplex all the data received from the DDR to the "AXI- Stream Data Width Converter" (DWC) or directly to the correct IP port. The pointwise module also receives the depthwise intermediate results and interacts with the DMA to send the final OFMs pixels to the DDR.

C. *MobileNets Depthwise Stage*

Figure 9 presents the block diagram of the depthwise module. The black lines represent the AXI-Stream bus with data, valid, last and ready signals. Blue lines represent the buses used to load the BRAMs with the parameters. The red lines represent the input signals of the control unit, namely the valid and last signal, used to load the data and begin the process of a particular module. Lastly, the yellow lines are the output signals from the control unit to control the module and the readings from the BRAM memories. IFM-MEM1 and IFM-MEM2 are formed each one by 2 BRAMs allowing to store a maximum of $16 \times 16 \times 32$ pixels at once. When the Conv sub-module is performing the depthwise convolution using a group of pixels from the IFMs stored in IFM-MEM1, IFM-MEM2 is receiving another group of pixels of the IFMs, and vice-versa, when the depthwise convolution is using the pixels stored in IFM-MEM2, IFM-MEM1 is receiving the next group of pixels. DW-MEM is formed by 4 BRAMs which allows storing all the depthwise weights in each stage. A single read of the DW-MEM outputs 9 weights of 16 bits ($9 \times 16 = 144$). DW-BN-MEM uses 1.5 BRAMs and stores the depthwise Batch Norm parameters in each stage. Three Batch Norm parameters of 16 bits each ($3 \times 16 = 48$) bits, can be output with a single read.

The depthwise module execution begins by receiving the depthwise weights and Batch Norm parameters and stores them on the DW-MEM and DW-BN-MEM respectively. After that, the DMA starts sending the IFMs pixels, as needed, in several streams. When the IFM-MEM1 (or IFM-MEM2) loading process is complete, the last signal from the stream is asserted and detected by the control unit from the depthwise convolution module, which informs the DW image loader that this sub-module can start the input of the pixels. At the same moment, IFM-MEM2 (or IFM-MEM1) is receiving another group of IFMs pixels, that will be ready to use when the depthwise stage process ends, and so on. The depthwise convolution can be performed using a shift register (SR). An SR can be efficiently implemented on an FPGA using LUTs, which is commonly referred to as an SRL.

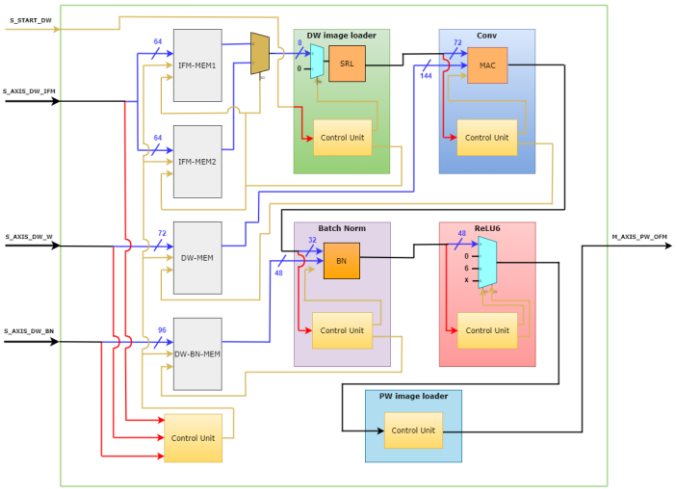

Figure 9: Depthwise convolution module.

To save resources, the proposed SRL is reconfigurable, allowing to perform convolutions of different IFMs sizes or strides. Since the maximum tiled block from the IFMs is 32 sets

of pixels of size 16×16, the SRL is composed of a total of 35 registers ($16 + 16 + 3$). Four multiplexers (mux) control how the data flows throw the SRL.

The MAC architecture of the Conv sub-module is pipelined, and executes the MAC of 9 pixels with 9 weights with a latency of 6 clock cycles. These 9 weights, corresponding to one kernel, are loaded by the control unit from DW-MEM in a single reading and stored in 9 registers. The 9 weights are then multiplied by the 9 pixels (read from IFM-MEM1 or IFM-MEM2 and stored in 9 registers) at once and accumulated by the adders in the following 4 clock cycles.

The BN sub-module performs all the arithmetic involved in the Batch Norm function (eq 12). Before performing the add operation or sending the pixel out to the ReLU6 sub-module, the number is aligned, using the shift right operation. Three registers store the values used for these shifts and they change every stage. Another register stores the pixel that comes from the Conv sub-module and other 3 registers store the Batch Norm parameters read from the DW-BN-MEM. The ReLU6 is the last sub-module used to perform the depthwise convolution and tests if the number is smaller than zero; greater than zero and smaller than six; or greater than six, and selects the output of the mux according to the evaluation result.

Lastly, the PW image loader is responsible for organizing the OFM pixels before sending them to the pointwise module. While the depthwise convolution is performed in height and width, the pointwise convolution is performed in depth. This requires grouping the pixels with the same index of the different depthwise OFMs while storing them on the IP memory. The control unit from the PW image loader executes this process by identifying the index from the valid input pixel and what OFM does it belong to. After that, an address is assigned to this pixel and the information is immediately sent to the pointwise convolution module, to store the pixel in the correct position of the IP memory. The process is repeated until the last pixel of the OFMs is sent to the pointwise module.

D. MobileNets Pointwise Stage

Figure 10 presents the block diagram of the pointwise module. As in the depthwise module, the black lines represent the AXI-Stream bus with data, valid, last and ready signals. Blue lines represent the buses used to load the BRAMs with the parameters. The red lines represent the input signals of the control unit, namely the valid and last signal, used to load the data and begin the process of a particular module. Lastly, the yellow lines are the output signals from the control unit to control the module and the readings from the BRAMs.

The module is designed to implement the accumulation of 32 filters at a time (i.e. before sending a group OFMs pixels to the DDR) when the layer stride is 1, or the accumulation of 128 filters at a time when the layer stride is 2. IFM-MEM3 and IFM-MEM4 are formed each one by 8 BRAMs allowing to store a maximum of $14 \times 14 \times 32$ pixels at once. These two BRAMs are connected to work as a ping-pong memory and each read of one of these BRAMs outputs 32 pixels of 16 bits ($32 \times 16 = 512$). When the Conv sub-module is performing the pointwise convolution using a group of pixels from the IFMs stored in IFM-MEM3, IFM-MEM4 can receive another group of pixels of the IFMs, and vice-versa, when the depthwise convolution is using the pixels stored in IFM-MEM4, IFM-MEM3 is receiving the next group of pixels. These pixels come from the depthwise module and do not require any writing control of the control unit, since the information that comes from the depthwise module contains the pixel, destination address and the other control signals. PW-MEM is formed by 7.5 BRAMs allowing to store a maximum of 32,768 pointwise values in each stage. This way, a single read of the PW-MEM, outputs 64 values of 8 bits ($64 \times 8 = 512$). This unit can work as a ping-pong memory depending on the layer. When the number of weights is fewer than 32,768, all the parameters can be stored in a single transfer on the 4,096 input addresses. On the other hand, when the number of weights is higher than 32,768, the parameters are sent in blocks of 16,384. This allows PW-MEM to receive data on half the addresses while the pointwise convolution is reading other data from the other half. The last BRAMs that receive data from an AXI-Stream interface is PW-BN-MEM, which uses 1.5 BRAMs and stores all the pointwise Batch Norm parameters in each stage. A single read allows to output 6 Batch Norm parameters of 16 bits each ($6 \times 16 = 96$) bits. The M_START_DW signal is used to start the depthwise convolution when the pointwise convolution finishes. This happens in each stage after the first pointwise convolution.

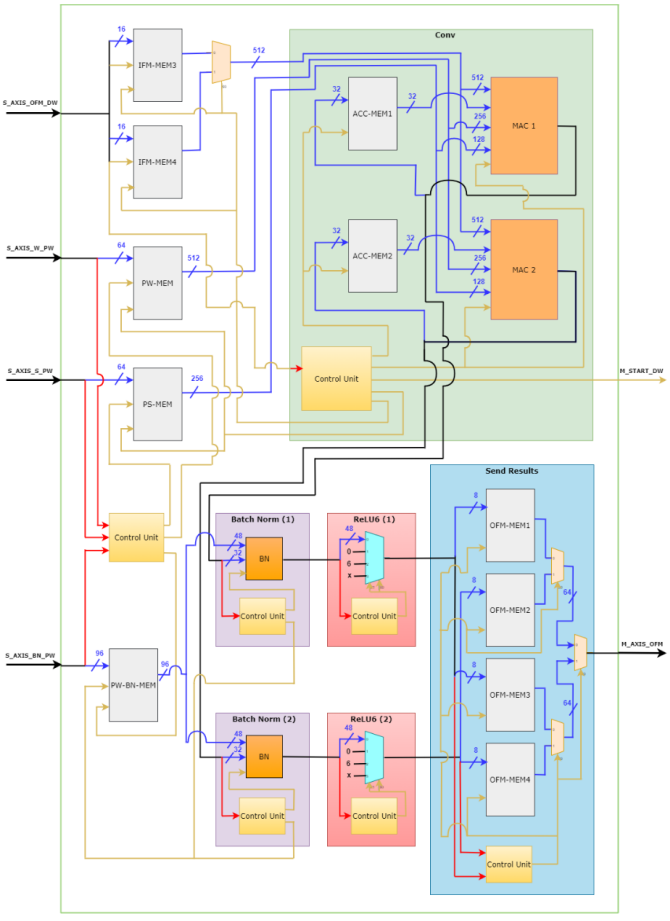

Figure 10: Pointwise convolution module.

The MAC architecture from the Conv sub-module is pipelined and allows the MAC of 32 pixels with 32 values with a latency of 8 clock cycles. These 32 values, corresponding to a portion of the filter, is loaded by the control unit from PW-MEM in a single reading and stored in 32 registers. The 32

values are then multiplied by the 32 pixels (read from IFM-MEM3 or IFM-MEM4 and stored in 32 registers) at once, and the result stored in the following 32 registers. These results are then shifted according to the scale factors used to quantize the respective weight, which were read from the PS-MEM and stored on 32 registers. After that, the shifted results are accumulated by the adders in the following 5 clock cycles. The option of having 2 MACs sub-modules allows performing two convolutions at the same time. However, the convolutions have to be processed in the following way: MAC 1 uses the first half of the filters and MAC 2 the second half. Therefore, ACC-MEM1 will accumulate the intermediate results from the first half of the OFMs, and ACC-MEM2 the intermediate results from the second half. To store these intermediate results, both ACC-MEM1 and ACC-MEM2 are formed by 3.5 BRAMs and have an input and output port of 32 bits wide.

The Batch Norm and ReLU6 sub-modules are implemented as in the depthwise convolution stage. Batch Norm (1) and ReLU6 (1) process the intermediate values stored in ACC-MEM1, while the other two sub-modules process the intermediate values stored in ACC-MEM2. Having two groups of these modules allows keeping the generation of two results at a time. The values are then propagated to the Send Results sub-module.

This Send Results sub-module precedes the transfer of the OFMs pixels to the DDR. OFM-MEM1, OFM-MEM2, OFM-MEM3 and OFM-MEM4 are formed, each one, by 1 BRAM. The function of this unit is to store the OFMs pixels before sending them to the DDR. The transfer can begin when the group of 32 or 128 filters (depending on the stride) finishes the convolution with the IFMs pixels and the pointwise sub-module starts the convolution with a new group of filters. This way, the OFMs pixels are sent while the pointwise convolution is producing new results, hiding the latency behind the communication. To perform this operation, the memories work as two ping-pong memories. OFM-MEM1 or OFM-MEM3 stores the data from ReLU6 (1) and OFM-MEM2 or OFM-MEM4 the data from ReLU6 (2). The reason to have the memory system implemented this way is that the OFMs pixels are not sent immediately when the other pointwise starts. When the system is performing a pointwise convolution, this execution time is used to exchange data between the PS and the PL, namely IFMs and pointwise weights. Therefore, the Send Results sub-module can only start to transfer data when the other transfers end. However, during this wait, other OFMs pixels may need to be stored in this sub-module, and to avoid the corruption of the data, these new pixels are stored in the other IP memory. Besides organizing the OFMs pixels and synchronizing the transfers, the Send Results sub-module also allows sending 8 pixels at a time to the DDR, since the output data port of the IP memories is 64 bits wide communication.

VI. SoC-FPGA MobileNets Results Analysis

This section presents the hardware resources used by the accelerator and a comparison of the execution times for the software and hardware/software implementations for each stage, and the respective speed up.

A. *Resources utilization*

Table 4 presents the utilization of all the main components used on the MobileNets accelerator. The implementation was designed to take full advantage of the available BRAMs and DSPs because these are crucial to perform the MACs. Comparing with the depthwise module, the pointwise implementation uses roughly 5 times more BRAMs and 6 times more DSPs.

B. *Experimental results*

The architecture was synthesized using the Vivado 2019.1 and executed using a clock frequency of 115MHz. To maximize the number of results per clock cycle, the architecture uses a pipeline implementation and two types of parallelism. Operator level parallelism is used when performing the MAC in both depthwise and pointwise PEs, using the 79 DSPs at the same time to produce 2 OFM results per clock cycle. On the other hand, intra-output parallelism is used in the Conv sub-module of the pointwise PE since two different filters are applied simultaneously to the IFM pixels. With the best optimization (O3) the hardware/software implementation for the 13 depthwise/pointwise stages is 16 times faster than the software application, and considering only the last 7 stages (all implemented in hardware), the speed up is 18 (table 5).

Table 4: Distribution of the resources used in the MobileNets accelerator.

Component	LUT (%)	FF (%)	BRAM (%)	DSP (%)
DMA	9.1	6.6	5.0	0
AXI Smart Connect	12.7	8.7	0.0	0
Width converter	5.6	7.0	0.0	0
Stream Connector	2.4	1.6	0.0	0
Depthwise stage	5.8	3.8	15.8	13.8
Pointwise stage	44.8	16.4	72.5	85.0
Others	2.3	1.4	0.0	0
Total	82.7	45.5	93.3	98.8

Table 5: Time execution results for software and hardware/software using optimization O3.

Stage (DW +PW)	SW (03) (ms)	HW + SW (03) (ms)	Speed up
1	387	21	19
2	303	32	9
3	549	38	14
4	250	23	11
5	492	33	15
6	232	19	13
7	451	26	17
8	451	26	17
9	451	26	17
10	450	26	17
11	450	26	17
12	223	13	17
13	438	18	24
Total	5,126	328	16

VII. Conclusion

This work focused on the research and development of a custom hardware/software architecture to execute the MobileNets inference process on an SoC FPGA. The main requirement defined for this project was implementing efficiently the software/hardware architecture on a low-cost device making a minimal trade-off between accuracy and execution time. Although MobileNets is a CNN model for image classification designed to run on embedded systems, the standard model may still be too computing intensive to run on low-cost devices such as the Xilinx Zynq-7010 or 7020. Due to this, a quantization analysis was conducted to evaluate the model when different representations are used for the model parameters, instead of the 32-bit floating-point. This allowed to reduce the MobileNets model using 16-bit fixed-point to quantize 26% of the parameters, and 12-bit custom floating-point to quantize the remaining 74%. The final result obtained was a MobileNets model about 60% smaller than the standard with only a very small accuracy reduction of 0.78%, which is below the 1% of maximum stipulated for the accuracy loss.

The hardware was designed to execute the quantized MobileNets model resulting from the previous analysis. Two types of parallelism were used to process the MACs in parallel. Operator-level parallelism is used when performing the MAC in both depthwise and pointwise PEs, and intra-output parallelism is used in the Conv sub-module of the pointwise PE since two different filters are applied simultaneously to the IFM pixels. The architecture developed follows a pipeline structure to produce two valid results per clock cycle. Also, the use of ping-pong memories hides the latency and allows the exchange of data between the PS and the PL while the computation of the depthwise and pointwise is running.

Lastly, the proposed hardware/software was demonstrated and analysed on the Xilinx Zynq 7010 device. The design of the architecture was the main challenge of this project because of the limited number of resources of the device. Almost all of the BRAMs and DSPs were used, which was another requirement defined for this project. The proposed hardware/software achieved a speed up of 16 times for the 13 stages, compared to the software implementation (with the O3 optimization). Another requirement for this project was to reduce the execution time of the inference process below 1 second, which was also successfully achieved since the final execution time of the hardware/software implementation (considering all layers and functions) is 469 ms.

VIII. Future work

Despite all the project requirements being achieved, there are some considerations that can improve this work. The architecture can be adapted to perform also the Conv3D and FC layer in hardware since now they represent about 30% of the execution time. The filter of the Conv3D layer has a size of $3 \times 3 \times 3 = 27$. Therefore, the depthwise IP may be adapted to perform the Conv3D convolution since the module can perform $3 \times 3 = 9$ MACs every clock cycle. This means that the depthwise IP would need 3 clock cycles to produce one MAC of the Conv3D convolution.On the other hand, the FC layer is similar to the pointwise convolution. The difference is the size of the IFMs and the bias sum after the convolution. Therefore, the pointwise IP may be adapted to perform the FC layer.

Lastly, the custom floating-point can be explored in future works, namely, quantizing all the MobileNets parameters with the custom floating-point strategy or extend the concept to other CNNs. Therefore, as more bits are used in the exponent, higher accuracies may be achieved.

IX. References

[1] C. Gershenson, "Artificial Neural Networks for Beginners," *arXiv:cs/0308031,* September 2003.

[2] K. O'Shea e R. Nash, "An Introduction to Convolutional Neural Networks".*arXiv:1511.08458.*

[3] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto e H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv:1704.04861,* 17 April 2017.

[4] S. Ioffe e C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv:1502.03167,* February 2015.

[5] A. Fred e M. Agarap, "Deep Learning using Rectified Linear Units (ReLU)," *arXiv:1803.08375,* March 2018.

[6] "TensforFlow," [Online]. Available: https://www.tensorflow.org/. [Accessed 16 September 2020].

[7] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *arXiv:1806.08342,* 21 June 2018.

[8] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam e D. Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," *arXiv:1712.05877v1,* December 2017.

[9] T. Sheng, C. Feng, S. Zhuo, X. Zhang, L. Shen e M. Aleksic, "A Quantization-Friendly Separable Convolution for MobileNets," *arXiv:1803.08607,* March 2018.

[10] J. Su, J. Faraone, J. Liu, Y. Zhao, D. B. Thomas, P. H. W. Leong e P. Y. K. Cheung, "Redundancy-reduced mobilenet acceleration on reconfigurable logic for imagenet classification," *in Applied Reconfigurable Computing. Architectures, Tools, and Applications,* April 2018.

[11] S. Chakradhar, M. Sankaradas, V. Jakkula e S. Cadambi, "A Dynamically Configurable Coprocessor for Convolutional Neural Networks," *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture,* p. 247–257, June 2010.

[12] L. Bai, Y. Zhao e X. Huang, "A CNN Accelerator on FPGA Using Depthwise Separable Convolution," *arXiv:1809.01536v2,* 6 September 2018.

[13] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov e L.-C. Chen, "MobileNetV2 Inverted Residuals and Linear Bottlenecks," *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) arXiv:1801.04381,* pp. 4510-4520, 13 January 2018.

[14] "Keras," "MobileNets" [Online]. Available: https://keras.io/api/applications/mobilenet/#mobilenet-function. [Accessed 18 November 2020].

[15] "Keras applications," [Online]. Available: https://keras.io/api/applications/. [Accessed 18 November 2020].