IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received May 18, 2020, accepted May 25, 2020, date of publication June 8, 2020, date of current version June 18, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3000444

A Configurable Architecture for Running Hybrid Convolutional Neural Networks in Low-Density FPGAs

MÁRIO P. VÉSTIAS[1], **RUI P. DUARTE**[2], **(Member, IEEE), JOSÉ T. DE SOUSA**[2], **(Member, IEEE), AND HORÁCIO C. NETO**[2]

[1]INESC-ID, Instituto Superior de Engenharia de Lisboa, Instituto Politécnico de Lisboa, 1959-007 Lisboa, Portugal
[2]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, 1049-001 Lisboa, Portugal

Corresponding author: Mário P. Véstias (mvestias@deetc.isel.ipl.pt)

This work was supported in part by the National Funds through Fundação para a Ciência e a Tecnologia (FCT) under GrantUID/CEC/50021/2019 and Grant UIDB/50021/2020, and in part by project SARRROCA PTDC/EEI-HAC/31819/2017, through INESC-ID and inCNeuralN IPL/IDI&CA/2019/inCNeuralN/ISEL through Instituto Politécnico de Lisboa.

ABSTRACT Convolutional neural networks have become the state of the art of machine learning for a vast set of applications, especially for image classification and object detection. There are several advantages to running inference on these models at the edge, including real-time performance and data privacy. The high computing and memory requirements of convolutional neural networks have been major obstacles to the broader deployment of CNNs on edge devices. Data quantization is an optimization method that reduces the number of bits used to represent weights and activations of a network model, minimizing storage requirements and computing complexity. Quantization can be applied at the layer level, by using different bit widths in different layers: this is called hybrid quantization. This article proposes a new efficient and configurable architecture for running CNNs with hybrid quantization in low-density Field-Programmable Gate Arrays (FPGAs) targeting edge devices. The architecture has been implemented on the Xilinx ZYNQ7020/45 devices and is running the AlexNet and VGG16 networks. Running AlexNet, the architecture has a throughput up to 508 images per second on the ZYNQ7020 device, and 1639 images per second on the ZYNQ7045 device. Considering VGG16, the architecture delivers up to 43 images per second on the ZYNQ7020 device, and 81 images per second on the ZYNQ7045 device. The proposed hybrid architecture achieves up to 13.7× improvement in performance compared to state-of-the-art solutions, with small accuracy degradation.

INDEX TERMS Convolutional neural network, deep learning, embedded computing, field-programmable gate array, hybrid quantization.

I. INTRODUCTION

Convolutional neural network (CNN) is a type of deep learning network used for image classification [1] and many other applications in computer vision. The CNN model mimics the structure of the human brain, with the neurons organized in a series of layers, and whose interconnections have associated weights that enhance specific characteristics of the image. A training process is used to determine the weights, after which the network is able to classify images other than those used during the training process. A trained network will classify a new image as belonging to one of the classes considered in a process known as *inference*.

CNNs demand large storage and computational resources and therefore are typically run on high-performance computing platforms. However, a vast application set can be enabled by running CNN models on edge devices [2]. These smart embedded devices can take real or almost real-time decisions based on the analysis of locally collected data.

The fact that edge devices are normally resource-constrained, makes it difficult to run CNN models on them with acceptable accuracy and response time. To solve this

The associate editor coordinating the review of this manuscript and approving it for publication was Ting Li.

problem, many optimization methods have been devised and applied to network models and architectures in order to reduce the computation complexity without significant accuracy degradation.

Data quantization is one of these methods. Data quantization reduces the computational weight by using fixed-point instead of floating-point hardware and smaller data words. It has been shown that using different data sizes in different layers can considerably reduce the hardware complexity, improve the performance, and still keep an acceptable accuracy of the results. This method is called hybrid data quantization [3].

Designing a dedicated architecture to run CNNs with hybrid data quantization is challenging since different networks will require different data quantizations in different layers. Hence, the devised architecture must be flexible enough so that different data sizes in different layers may be configured. A few works have used Field-Programmable Gate Arrays (FPGAs) to implement hybrid architectures [3], [4], since the FPGA can be reprogrammed for each specific network. However, these approaches considered a complete pipeline structures, with a dedicated module for each layer, and their size only allowed mapping on medium to large FPGAs.

In this paper a different approach is proposed. The proposal consists in using modular hybrid computational cores, which allow further reducing the data sizes and storage requirements of the configurable architecture, while improving the inference run time. Such architecture can efficiently run CNNs in low-density FPGAs, enabling smart embedded computing [5]. The proposed hardware modules can efficiently run dot-products of different data sizes in the FPGA, and effectively support hybrid data quantization. The experimental results show throughput improvements of up to 13.7× when compared to state-of-the-art approaches that tackle the same problem.

The paper is organized as follows. Section II describes related work on CNN FPGA implementations and optimization methods. Section III describes the baseline architecture without hybrid quantization. Section IV describes the proposed architecture for hybrid CNNs. Section V describes the area and performance results of the architecture running well-known CNNs. Section VI concludes the paper.

II. RELATED WORK

The ability of CNNs to classify images comes from the convolutional layers that run 3D convolutions of weight kernels and input feature maps (IFM). Each 3D convolution produces an output feature map (OFM) to be used by the next layer. Different kernels are used to identify different characteristics of the image stored in different output feature maps. Successive layers correlate more complex features, allowing the identification of particular classes of objects. Complex features are finally fully correlated with fully-connected layers, where all neurons are connected to all neurons of the previous layer. A final layer called *softmax* outputs the result of the inference process. The *softmax* layer uses a node for each class of object and produces the relative probability of the input image belonging to the object class. In all layers the neuron outputs are driven by an activation function. Several activation functions have been proposed but recently the Rectified Linear Unit (ReLU) function and its variations are the most commonly used due to their simplicity, advantages during training and good inference results.

The size of the layers, number of kernels and the interconnections between layers determine the accuracy of the network for each class of classification problems. As a consequence, many different CNNs have been proposed in the last years for improving the accuracy in different sets of images. LeNet [6] was one of the first CNNs used for classification of hand-written digits in 32×32 images, and achieved a very high (over 99%) accuracy. LeNet used 2 convolutional and 3 fully-connected layers, in a total of 60K weights.

The evolution of high-performance computing platforms made possible the training of networks with 1000× more weights than those used in LeNet. This way, larger images could be classified. The AlexNet network [7] has 5 convolutional and 3 fully-connected layers, in a total of 61M weights. This increase in the number of weights required much more memory and many more operations (1.5 GOp) to process a single image. AlexNet won the 2012 ImageNet Challenge (for the classification of images of size $224 \times 224 \times 3$ in 1000 different classes), achieving top-5 and top-1 error rates of 17% and 37.5%, respectively.

The achievements of AlexNet have opened the road to new, larger and more accurate networks. VGG16 [8], a version of VGG with 16 layers, has 2.2× more parameters than AlexNet, and executes 31 GOp to run inference over an image. This large model achieved a top-5 error rate around 10% in the 2014 ImageNet Challenge. GoogleNet [9] is an irregular CNN with 22 convolutional layers and a new type of group layer (the inception layer consisting of parallel convolutions). The model needs 7M parameters and 1.58 GOp and achieved a top-5 error of 7% in the 2015 ImageNet Challenge. ResNet [10] introduced a new composite module containing an identity connection to reduce the complexity of the training. Several versions of ResNet have been designed with a number of convolutional layers ranging from 53 to 155. With up to 11.3 GOp of workload, ResNet reduced the top-5 error from 6.7% to 5.8%. ResNet was the first CNN exceeding human-level accuracy in the ImageNet Challenge.

CNNs, like any other deep neural network, are very compute-intense and therefore are typically run on high-performance platforms. However, high-performance devices are not appropriate for smart embedded computing, since they are energetically inefficient and too expensive. Current embedded Central Processing Units (CPUs) achieve only a few dozen Giga FLoating-point Operations per Second (GFLOPS) with insufficient power efficiency for real-time or almost real-time processing of CNNs. Graphics Processing Units (GPUs) offer thousands of GFLOPS at the cost of high energy consumption, disqualifying them for embedded computing. Finally, dedicated high-performance units (Application-Specific Integrated Circuits) are quite high-performance and energy-efficient, but they are not programmable and unable to map different networks or adapt to the constant evolution of deep neural networks.

Reconfigurable computing devices, in particular FPGAs, are used to implement accelerators for CNN inference [11], [12]. They are more energy-efficient than GPU and CPU based-platforms and achieve better performance than CPUs. The reconfigurability of FPGAs allows designing a dedicated datapath for a specific CNN model [13], and efficiently implement optimization strategies. Designing a dedicated accelerator for CNN inference requires some hardware expertise when following the traditional FPGA design flow. To speed-up the design process, a few works have proposed automatic frameworks and high-level synthesis (HLS) tools that automatically convert a specification of the network model into a dedicated FPGA architecture [14]. HLS tools allow the fast design of architectures but are not optimized for best performance and resource usage. A template-based approach considers configurable cores or architectural templates that can be configured for a particular network [15]. Solutions obtained from an architectural template are more optimized but less flexible. A mid-term solution considers an overlay processor [16], which is a programmable hardware structure that runs a compiled CNN model, but adds some overhead associated with the overlay. Since our proposal targets constrained devices, it follows a template-based approach, where a generic and configurable architecture can be tailored for each specific network and target device.

Initial FPGA implementations of small CNNs [17], [18] have been followed by FPGA implementations of all layers of a CNN model [19], [20]. CNN models are made of different types of layers, and each layer has filters and maps of different sizes. Therefore, the architectures proposed are flexible enough to run both fully connected and convolutional layers of different shapes and sizes. One of the main problems when designing a flexible enough accelerator is how to deal with different convolution window sizes from layer to layer. To solve this problem, convolutions are converted to matrix multiplications by rearranging the input feature maps [20]. To reduce the overhead associated with map rearrangement, dedicated hardware units are used to convert the input maps into a matrix [21].

The utilization of a single monolithic hardware module to run layers with different characteristics may lead to low throughput, since different layers require different computing patterns. Alternatively, the module may be designed allowing for various options and exceptions, which becomes overly complicated. In this paper, 3D convolutions are calculated as long inner products, without any additional computational effort for rearranging the IFM. As such, the hardware structure to run 3D convolutions does not change with the window size.

The first generation of CNN implementations take performance as the main optimization metric. Recently, a few works based on the single module approach [22]–[24] have started to consider other metrics such as area and power, to enable design trade-offs.

In [22], only convolutional layers are considered, while in [23], [24], convolutional and fully-connected layers are considered but the same module is used in both. Instead of using the same hardware module in all layers, pipelines of layer-specific modules have been proposed [25], [26]. These architectures are quite efficient, since unique modules are optimized for each layer. However, they require significant memory resources to store all intermediate maps and weights, which may easily consume all available on-chip memory in low-density FPGAs. A trade-off between a single hardware module for all layers and a pipeline of layer-specific modules was proposed in [27], where layer subsets are mapped to a fixed set of modules. This solution sacrifices performance for resource efficiency and consists of several modules, each maps to a subset of the CNN layers. This system achieved a $2\times$ speedup for SqueezeNet in a Virtex7 FPGA compared to state-of-the-art architectures.

To further reduce the number of computing modules, the present proposal considers only two different modules: one for convolutions and the other for fully-connected layers. Since the structure is the same regardless of the window size, it achieves the same efficiency as a solution with multiple modules while reducing the required area.

A pipelined accelerator for CNNs on low-density FPGAs was proposed in [28]. To reduce the required on-chip memory, it applies a layer-fused technique. With dedicated modules for each layer, the solution runs AlexNet at 80 Giga Operations per Second (GOPS) on a ZYNQ7020 FPGA. The problem of this approach is that it requires on-chip memory in all layers. Given the finite on-chip memory resources, it needs a complex circuit to orderly access external memory, which reduces the available resources to implement the computing cores. This unbalance between computing and memory resources reduces the performance efficiency of the solution for small FPGAs. Our approach also manages to map large CNNs into low-density FPGAs and achieves better performance efficiency.

Data quantization is an optimization technique that changes the type and reduces the size of the numeric representation of parameters in the model, and consequently reduces the required storage size and computing resources. In [29], 8-bit fixed-point representations are shown to guarantee an accuracy close to that obtained using 32-bit floating-point numerical representations. While fixed-point quantization schemes have proliferated, a new and notable floating-point representation, called the block floating-point scheme, using 8 and 16-bit data widths has been proposed [30]. The new representation reduces the accuracy loss by using simplified floating-point operations. The proposal has been tested using the VGG16 network on a Virtex 7 VX690T and has achieved a performance of 760 GOPS.

The data widths can be fixed for all layers or optimized for each layer. In [3], an implementation of a mixed-precision

VGG16 network on a ZYNQ7045 FPGA, achieved a performance of 316 GOPS, almost three times better than previous approaches with fixed data sizes for all layers. In [4], a hybrid quantization scheme uses 8-bit fixed-point and shift quantization (powers of 2) to represent weights, with fixed 8-bit activations in all layers. The solution improves the hardware area due to shift quantization. However, like in [3], the system is implemented as a pipelined architecture and mapped to high-density FPGAs.

In [31], hybrid quantization is explored in implementations both for low-density FPGAs (edge computing) and high-density FPGAs (cloud computing). However, the adopted bit-serial matrix multiplication architecture for low-density FPGAs has a negative impact on the system performance. The hybrid approach in [3] has the best performance efficiency overall, but its pipelined implementation of layers can not be mapped into low-density FPGAs. In [32], a core that uses 8 and 2-bit weights is proposed.

The solution proposed in this paper is able to run networks with hybrid representations in low-density FPGAs, since the same hardware module is able to run the inner products of vectors with different bit sizes.

In extreme quantization implementations, CNNs are converted to Binary Neural Networks (BNNs) [33], [34]. In BNNs, weights or both activations and weights are represented with a single bit, further reducing memory requirements. Networks having 1-bit weights can lose more than 10% accuracy for large networks. To obtain an accuracy comparable to using floating-point, a BNN needs from 2 to 11× more weights and operations [33]. It is also known that the first and last layers require full precision, forcing the architecture to support both representations. The accuracy gets even worse when both weights and activations are represented with a single bit. BNNs can be efficiently implemented with FPGA Look-Up Tables (LUTs), sparing the DSPs for performing additions only, which greatly reduces the resource utilization.

Our solution supports 1-bit weights but not 1-bit activations. Since we are targeting large CNNs, for which BNNs show considerable accuracy losses, we do not use fully binary networks.

A few works have considered the implementation of CNNs in a ZYNQ7020 FPGA with data quantization. [15] reported the implementation of a small CNN in this device, where the representation of weights and activations with 16-bit fixed-point data limits the average performance to 13 GOPS. In [35], AlexNet and VGG16 are implemented in the same ZYNQ7020 FPGA using 8-bit fixed-point data. The performance improves to 84 GOPS, partly because the data is represented with half the size compared to [15]. In [36], a low power 8-bit network implemented in the same ZYNQ7020 device, manages an average performance of 41 GOPS. A pipelined architecture using 16-bit fixed-point data, and pruning applied to fully connected layers, achieved a performance of 76 GOPS [28]. These works targeted low-density FPGAs but achieved a relatively low performance because a homogeneous data representation has been considered. Our proposal also targets low-density FPGAs but can achieve over 1 Tera Operations per Second (TOPS) of average performance using hybrid quantization.

In this paper, a configurable architecture to execute CNNs with hybrid data quantization is proposed. The proposal consists of a two-stage pipeline architecture with two separate modules to process the two types of regular layers: convolutional and fully-connected layers. Compared to previous CNNs, the proposed architecture improves on the inference performance and resource efficiency, making it suitable for low-density FPGAs as well as high-density ones.

III. BASELINE ARCHITECTURE OVERVIEW

The architecture proposed in this paper applies hybrid quantization to a state-of-the-art architecture that implements large CNNs in low density FPGAs using an 8-bit fixed-point representation format. This baseline architecture is a follow-up of the work presented in [5], [37] and is described in this section.

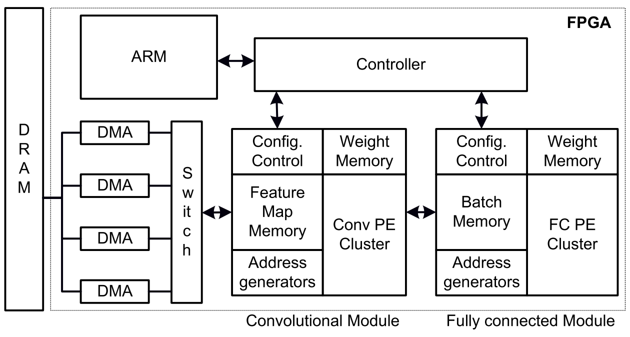

FIGURE 1. Block diagram of the baseline architecture.

The baseline architecture has one dedicated hardware module to run convolutional layers and another to run fully-connected layers in parallel (see Figure 1). This allows for different optimization techniques to be applied to convolutional or fully-connected layers. Each block executes one layer at a time and must be configured before running each layer. The configuration specifies the number of kernels, their size, memory addresses, the existence of the pooling layer and its window size, and the scale of the fixed-point format. The input image and the feature maps are loaded to on-chip memory in order to be processed. Depending on the size of the image and intermediate feature maps, they may fit or not in the on-chip memory. If it does not fit, the image or the feature maps are divided and processed in pieces. The output feature maps generated by a layer are stored back in external memory, and reloaded for the next layer. This increases the volume of data communication with the external memory but allows for the execution of CNNs whose maps do not fit in the available on-chip memory.

Both the convolutional and the fully connected modules have a cluster of Processing Elements (PEs) comprising local memories and computing units. Pooling layers and activation functions are implemented outside the PE clusters, since they are applied only to the result of convolutions and inner products. The execution flow of a CNN model in the baseline architecture consists of the sequential application of the following steps to each layer: loading feature maps and weights to on-chip memory, executing convolutions or inner-products, and storing output maps in external memory. The process takes into consideration the size and number of on-chip memories. If the input maps do not fit in the feature map memory then they are loaded in smaller chunks. If the number of kernels is higher than the number of weight memories, they are loaded in small groups.

The architecture is controlled at three levels. The convolutional and fully connected modules each have a local configuration controller, and the complete accelerator has a general controller. The configuration of each layer is done by a host processor, including the configuration of the direct memory access (DMA) modules that read kernels from external memory and read/write feature maps to external memory.

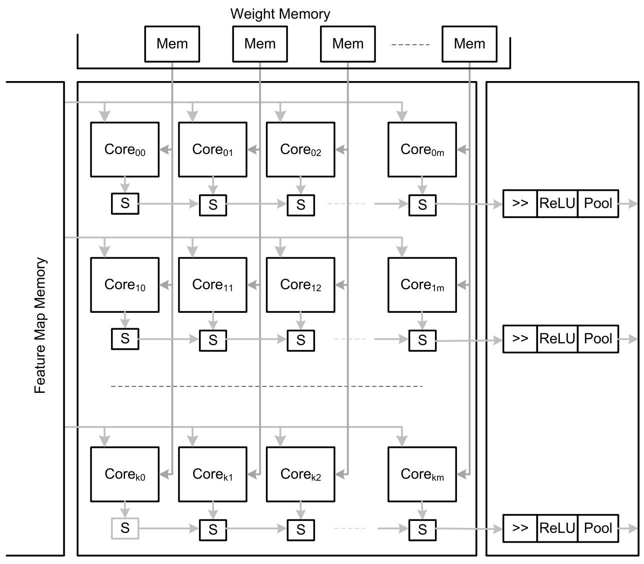

FIGURE 2. **PE cluster architecture for convolutional layers [37].**

The PE cluster of the convolutional module has an array of cores organized by lines and columns. Each line of cores is connected to a single port of the feature map memory and therefore executes over the same block of activations. Each core of the line receives a different weight kernel. Therefore, each core in a line generates an activation of a different output map. Cores in the same column receive the same weight kernel. Each core in a column receives a different block of activations from the feature map memory. Thus, cores in the same column generate activations of the same feature map (figure 2). Each core has a parallel multiply-add (MAD) unit to calculate inner-products. Multiple weights and activations are read and processed in parallel by the MAD units. The baseline architecture has been implemented with 8-bit fixed-point data. Memory ports are 64-bit wide. Hence, 8 activations and weights are read and processed in parallel by each parallel MAD unit.

The baseline architecture can execute convolutions with different window sizes with the same hardware module. This is done by reading activations and weights as long vectors. The inner product of these two vectors is computed by the parallel MAD units.

Let us consider a set of input feature maps with size $x_p \times y_p \times z_p$, where $x_p \times y_p$ is the size of the maps and z_p is the number of maps. One output activation is the result of the inner product between a kernel with size $x_k \times y_k \times z_k$ and the corresponding input activations. Both 3D kernels and activations are read sequentially from memory following the order z, x, y. The addresses to read the activations in this order are generated by address generators configured before running the layer. Formally, the inner product between a kernel and the corresponding activations is given by:

$$DP_{conv} = \sum_{i=0}^{y_k-1} \sum_{j=0}^{x_k z_k-1} W_{i\,x_k\,z_k+j} \times A_{firstAddr+i\,x_p\,z_p+j} \qquad (1)$$

where *firstAddr* is the address of the first activation of the block being convoluted. The complete convolution is obtained by sliding the kernel over the input maps with a particular stride, while applying equation 1 to each position. The output activation is then stored back in the feature map memory, or first pooled from the pooling window an then stored in the map memory. The advantage of this method is the fact it works regardless of the size of kernels or convolution windows.

The convolution of the 3D kernel with the 3D input map is given by Algorithm 1, considering the set of all 2D input feature maps (which together form a 3D input map of size $x_p \times y_p \times z_p$), a 3D kernel of size $x_k \times y_k \times z_k$, a pooling window of size $x_{pool} \times y_{pool}$ and a stride of size t. In the algorithm, *poolFunction* is the maximum or average pooling function. The *startAddr* function adds the correct offset to the address pointer of the feature map memory, depending on the next activation to be calculated. The *startAddr* function takes as arguments the size of the input feature map, the pooling size, and the stride value. Given the first memory address where the feature map is stored (*initialAddr*), the *startAddr* function is given by:

$$\begin{aligned} startAddr = initialAddr + k \times z_p + l \times z_p \times x_p \\ + m \times z_p \times s + r \times z_p \times x_p \times s \end{aligned} \qquad (2)$$

As described in Algorithm 1, each output neuron is calculated as the inner product of the kernel and an IFM block of activation values using equation 1. For example, considering a kernel of size 3 × 3 × 128, each output neuron is calculated as the inner product of the kernel and a 3 × 3 × 128 activation block. The 3D kernel slides over the whole input map, spanning the $x_p \times y_p$ space. When the stride is greater than one, the 3D kernel slides over the input map, skipping over input activation values according to the stride size. For example, a stride of two means that the every other activation is skipped, reducing the output map to half the size of the input map. The algorithm also includes the optional

Algorithm 1 Convolution With a 3D Kernel

```
Input: Set of all input 2D feature maps (3D input map) and
    one 3D kernel of weights
Output: One output feature map result of the convolution of
    the 3D input map with the 3D kernel
  for $r = 0$ to $y_p/t - 1$ do
    for $m = 0$ to $x_p/t - 1$ do
      $poolVar \Leftarrow 0$
      for $l = 0$ to $x_{pool} - 1$ do
        for $k = 0$ to $y_{pool} - 1$ do
          $dp = \sum_{i=0}^{y_k-1} \sum_{j=0}^{x_k z_k -1} W_{i x_k z_k + j} \times A_{startAddr(r,m,l,k)+i x_p z_p + j}$
          $poolVar \Leftarrow poolFunction(poolVar, dp)$
        end for
      end for
      $neuron_{(m,r)} \Leftarrow poolVar$
    end for
  end for
```

pooling step. In this case, the output neuron is generated only after computing all neurons in the pooling window. The algorithm sequentially calculates all neurons in the pooling window in order to merge the pooling layer with the convolutional layer.

The PE cluster of the fully-connected module consists of a matrix of cores with a structure identical to the PE cluster for convolutions. Each line of cores receives the activations of an image in the batch memory. Compared to the convolutional module, there is a major simplification in the address generators. Instead of an image convolution, a single dot product of the complete batched image and the kernel is performed. Parallel computation of dot products of a batched image and different kernels is possible by using multiple cores in a cluster line. The number of cores is limited by the available memory bandwidth, and is in general much lower than in the convolution PE cluster. This architectural difference between the two PE clusters is the main reason for having independent modules for convolutional and fully-connected layers, as it improves the hardware efficiency.

IV. HYBRID ARCHITECTURE

This work modifies and extends the baseline architecture to support the execution of layers with different weight sizes (8, 2 and 1 bit). Two new hybrid cores are proposed, which use these weight sizes and are able to efficiently calculate multiple dot-products. The size of the activations is kept fixed, since the precision of activations has a higher impact on the network accuracy than the precision of weights. The architecture described considers 8-bit activations but the design can straightforwardly support other activation sizes.

In the baseline architecture, each core receives eight 8-bit activations and eight 8-bit weights of a kernel in parallel. The new hybrid core always receives 64 bits of activations and 64 bits of weights. The dot-product level of parallelism is always 8, since there are only 8 different activations available in each cycle. However, in layers whose weights are represented with fewer bits, the 64 bits of weights can represent more than 8 weights. Since there are only 8 activations available, the 64-bit weight word is used to represent weights of different kernels.

The hybrid cores proposed in this work are able to calculate a number of dot-products in parallel, depending on the weight sizes. Consider, for example, a CNN where some layers have 8-bit activations and 8-bit weights and other layers have 8-bit activations and 2-bit weights. The hybrid core will execute one dot-product between eight 8-bit activations and eight 8-bit weights, in the first case, and four distinct dot-products between eight 8-bit activations and eight 2-bit weights, in the second case.

The two hybrid cores proposed herein are the following: one for 8- and 2-bit weights, named C8:82, and another for 8- and 1-bit weights, named C8:81. Their architectures are described in the following two sub-sections.

A. HYBRID CORE FOR 8-BIT ACTIVATIONS AND 8/2-BIT WEIGHTS - C8:82

This subsection details the hybrid core that supports the execution of two different *activation* $\times$ *weight* representations: 8×8 and 8×2 (C8:82). As stated above, using a different fixed size for activations is straightforward, since the architecture is generic for this parameter.

The approach proposed for the hybrid calculation consists on designing the core to conditionally execute 4 partial products of eight 8×2 dot-products when the layer is requesting an 8×8 dot-product.

Let us consider eight 8-bit activations, $A_0, A_1, \ldots, A_7$, that are read in parallel from the on-chip memory in each clock cycle. Weights are read from the weight memory. With 2-bit weights, 4 groups of eight 2-bit weights are read in parallel, $W_{00}, W_{01}, \ldots, W_{07}, W_{10}, W_{11}, \ldots, W_{17}, W_{20}, W_{21}, \ldots, W_{27}$ and $W_{30}, W_{31}, \ldots, W_{37}$. Then, the 4 dot products, DP_0, DP_1, DP_2 and DP_3 are calculated as:

$$DP_0 = \sum_{i=0}^{7} A_i \times W_{0i} \quad (3)$$

$$DP_1 = \sum_{i=0}^{7} A_i \times W_{1i} \quad (4)$$

$$DP_2 = \sum_{i=0}^{7} A_i \times W_{2i} \quad (5)$$

$$DP_3 = \sum_{i=0}^{7} A_i \times W_{3i} \quad (6)$$

These calculations are done using 4 dot-product units, each implemented with 8 cascaded MAD units (Figure 3). The eight 8-bit weights $W_0, W_1, \ldots, W_7$ are set as $W_k = W_{3k}W_{2k}W_{1k}W_{0k}$, where $k = 0, 1, 2 \ldots, 7$. Then, the 4 *DP* core outputs correspond to partial products that must be added

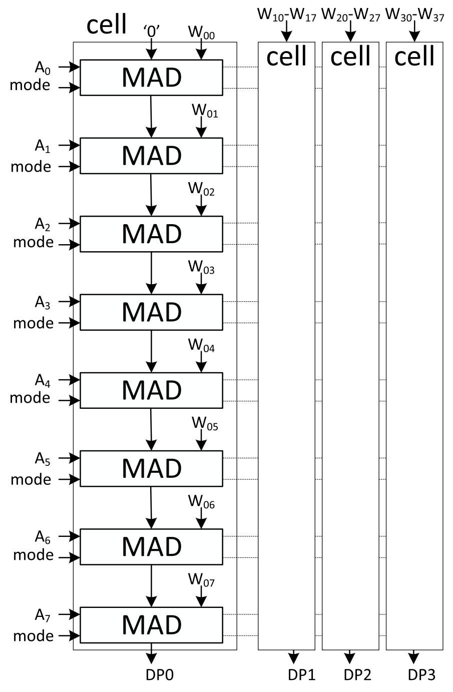

FIGURE 3. Architecture for 4 dot-products with 2-bit weights.

to calculate the final 8 × 8 dot-product:

$$A \cdot W = DP_3 \times 2^6 + DP_2 \times 2^4 + DP_1 \times 2^2 + DP_0 \qquad (7)$$

Equation 7 is executed only after calculating the complete convolutions. Therefore, it is implemented outside the core, and the addition is shared by all cores of a PE cluster line, followed by the ReLU activation function (not shown in figure 3).

Each cascaded MAD unit determines $R = X + A \times W$, where $A = \sum_{i=0}^{7} a_i \times 2^i$ is an 8-bit activation, $W = w_1 \times 2 + w_0$ is a 2-bit weight and $X = \sum_{i=0}^{9} x_i \times 2^i$ is the 9-bit output of the previous MAD.

The 2-bit sections represent different numbers depending on the layer weight size. If the weight size is 8 bits (mode 0 of operation), the 2-bit sections are partials of the 8-bit weight and represent the numbers $\{0, 1, 2, 3\}$, except for the most significant two bits which represent the numbers $\{-2, -1, 0, 1\}$. If the weight size is two bits (mode 1 of operation) then the 2-bit sections represent the numbers $\{-1, 0, 1\}$.

TABLE 1. MAD result for mode 1 of operation.

$w_1 w_0$	Operation
0 0	X + 0
0 1	X + A
1 0	not used
1 1	X - A

Let us first consider the case of weights represented with the 2-bit signed numbers $\{-1, 0, 1\}$ (mode 1). In this mode, X is added with one of the multiples of A in {-A, 0, A}, depending on the 2-bit weight value, as shown in table 1.

The 6-input LUTs of FPGAs can implement two 5-input functions as long as the inputs are common. The addition of X with one of these multiples can be efficiently implemented with $(m + 3)$ 6-input LUTs and the carry-chain, where $m = 8$ is the number of bits of the activation. Note that the generate and propagate signals of the carry chain are functions of only four variables (w_0, w_1, x_i, a_i). The LUTs produce the appropriate generate and propagate signals (and carry-in) to conditionally subtract/add A (or 0) from/to X.

For the multiplication by 8-bit weights (mode 0), one first determines the partial products of the activation by two bits of the weight. In this mode, the two bits of the weight represent the unsigned numbers $\{0, 1, 2, 3\}$, except the most significant 2 bits, which represent the signed numbers $\{-2, -1, 0, 1\}$.

When W is a signed 2-bit number, we need multiples A and 2A to implement $X + A \times W$. In this case, the generate and propagate signals are functions of the five variables $(w_0, w_1, x_i, a_i, 2a_i)$, and thus can be implemented with the same number of LUTs as with the signed 2-bit independent weights described before. However, in the case of the multiplication by the unsigned 2-bit weights $\{0, 1, 2, 3\}$, the generate and propagate signals are functions of the six variables $w_0, w_1, x_i, a_i, 2a_i, 3a_i$, since the $3\times$ multiple is also needed, and cannot be implemented with a single LUT.

However, using a modified Booth recoding algorithm [38] and the implementation method proposed in [39], it is possible to avoid the $3\times$ multiple, and implement the addition of a variable using a 5-variable function with a single level of LUTs.

The modified Booth algorithm recodes two bits at a time with overlapping 3-bit groups (a '0' is appended to the right of the number). The 3-bit recoding is done according to table 2.

TABLE 2. Recoding with modified Booth.

w_1	w_0	w_{-1}	Multiple
0	0	0	0
0	0	1	+A
0	1	0	+A
0	1	1	+2A
1	0	0	-2A
1	0	1	-A
1	1	0	-A
1	1	1	0

With weight recoding, only multiples A and 2A are needed. For example, applying Booth recoding to -89 = "10100111", the recoded number is given by: -1:-2:$+2$:-1. Thence, the multiplication of this number by A is given by "10100111" $\times A = (-1 \times 2^6 + (-2) \times 2^4 + 2 \times 2^2 + (-1)) \times A$.

In this case, the propagate and generate functions of the multiply accumulate operation $R = X + A \times W$ are still functions of the six variables $w_0, w_1, w_{-1}, a_i, 2a_i, x_i$, since recoding requires 3-bit inputs. However, using the method proposed in [39], variable x can be added with

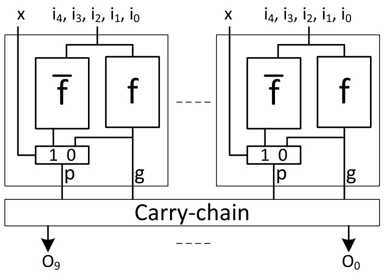

FIGURE 4. **MAD unit architecture for adding variable X with a 5-variable function.**

a function of the five variables $w_0, w_1, w_{-1}, a_i, 2a_i$, using the implementation illustrated in figure 4. The unit shown in figure 4 implements $x \oplus f(i_4, i_3, i_2, i_1, i_0)$ using both f and its complement $\bar{f}$.

Finally, both functions must be implemented with a single MAD, which implements both operation modes, 8-bit weights (mode 0) and 2-bit weights (mode 1). Therefore, an extra input is needed to specify the mode and, again, an addition of variable x with a 6-variable function is needed. To avoid this 6-input function, a different recoding is considered that takes into account the operation mode (see recoding in table 3).

TABLE 3. **Weight recoding considering the mode of operation.**

mode	w_1	w_0	w_{-1}	Multiple
0	0	0	0	0
0	0	0	1	+A
0	0	1	0	+A
0	0	1	1	+2A
0	1	0	0	-2A
0	1	0	1	-A
0	1	1	0	-A
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	+A
1	0	1	1	+A
1	1	0	0	–
1	1	0	1	–
1	1	1	0	-A
1	1	1	1	-A

Given the mode of operation, the weights are recoded into three bits (wr_2, wr_1, wr_0): one bit specifies the sign (wr_2) and the other two encode the multiples 0, A and 2A. In this method, the weights are first recoded and then sent to the MAD unit, where the applied function f_i (wr_2, wr_1, wr_0, $2a_i$, a_i) is given by:

$$f_i = \overline{wr_1} \cdot \overline{wr_0} \cdot \overline{(wr_2 \oplus a_i)} + wr_1 \cdot \overline{wr_0} \cdot (wr_2 \oplus 2a_i) \quad (8)$$

Input wr_2 is the carry-in bit needed to implement the 2's complement of the multiples. The recoded weights are shared by all cores in the same column of the PE cluster, and a single LUT can generate two recoded bits. This way, the resource overhead associated with recoding is very low.

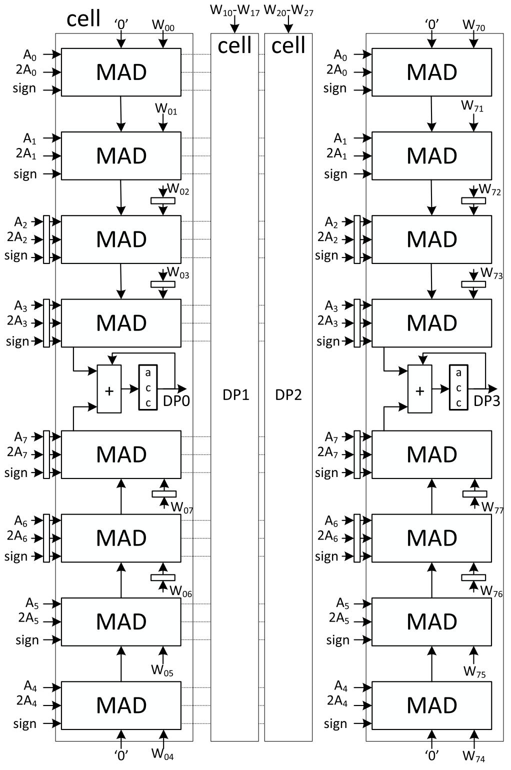

FIGURE 5. **Final architecture for 4 dot-products with 2-bit weights.**

The final C8:82 circuit diagram is illustrated in figure 5. The core inputs are the activation multiples A and 2A, the 2-bit weights and the MAD's second operand sign. In addition, a pipeline level has been added to improve the throughput of the circuit (other pipeline levels can be easily considered). The chain of eight MAD units has been broken into two sub-chains of four MAD units each, in order to reduce the delay of the complete cell.

The cores of the baseline architecture have been replaced with the new hybrid C8:82 core. Recode units have been added after the weight memories in both the convolutional and fully connected modules. The conditional addition of the partial products has been included before the shift and activation function. Figure 6 illustrates the new architecture of the convolutional module. The fully connected module follows a similar structure, as explained.

B. HYBRID CORE FOR 8-BIT ACTIVATIONS AND 8/1-BIT WEIGHTS - C8:81

The design of the hybrid C8:81 core, supporting the product of 8-bit activations by 8-bit weights, 8 × 8 (mode 0) and the product of 8-bit activations by 1-bit weights, 8 × 1 (mode 1), follows a similar approach as that for the design of core C8:82.

The C8:81 core generates 8 dot products, conditionally added as partials of an 8 × 8 product for 8-bit weight layers. For this core, a more efficient hardware solution is

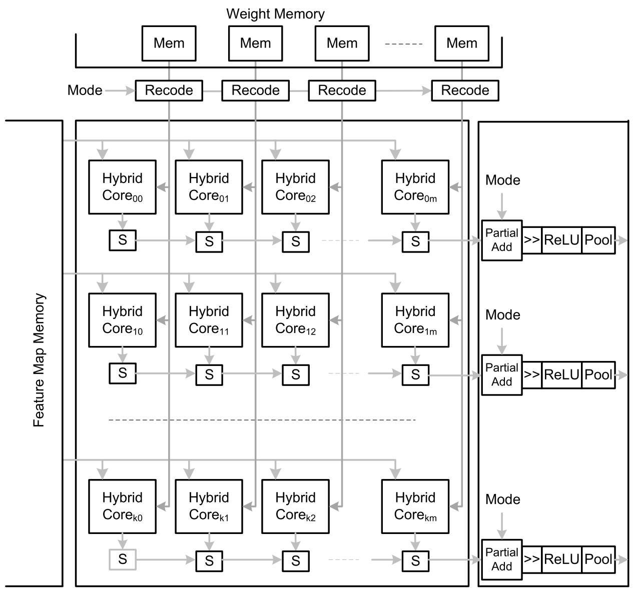

FIGURE 6. Hybrid PE cluster C8:2 core architecture for convolutional layers.

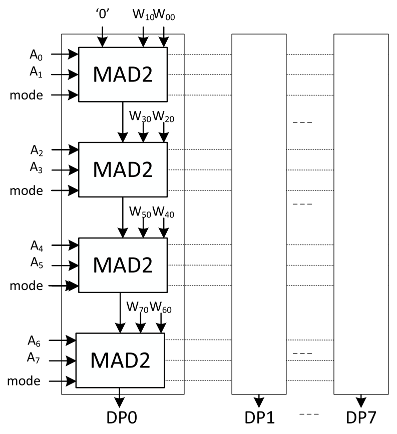

FIGURE 7. 8/1-bit weight core architecture.

implemented by multiplying and adding two activations in a single unit.

Each double multiply-add unit (MAD2) multiplies and accumulates two activations and two 1-bit weights, and adds the result of the previous MAD2. Four MAD2 units work together to calculate the dot-product between 8 activations and 8 weights (see Figure 7).

Considering 8-bit weights, the dot-product $A \cdot W$, between eight activations and eight 8-bit weights $W_0, W_1, \ldots, W_7$, is calculated by considering each weight as $W_k = W_{k7}W_{k6}W_{k5}W_{k4}W_{k3}W_{k2}W_{k1}W_{k0}$, where $k = \{0, 1, 2 \ldots, 7\}$ and then adding the eight partial products as:

$$A \cdot W = (\sum_{i=0}^{i=6} DP_i \times 2^i) - DP_7 \times 2^7 \qquad (9)$$

As before, equation 9 is implemented outside the core and before applying the activation function. Each MAD2 unit implements the dot-product between two 8-bit activations, A_a and A_b, and two 1-bit weights, W_a and W_b and adds it to the output of the previous MAD2, X. Formally, it calculates:

$$R = X + (A_a \times W_a + A_b \times W_b)$$

The 1-bit weight has different interpretations depending on the weight size. If the size is 1 bit then it represents the numbers $\{-1, 1\}$; if the size is 8 bits, the 1-bit weights are partials of the 8-bit weight multiplication and represent the numbers $\{0, 1\}$, except for the most significant bit that represents the numbers $\{0, -1\}$ (2's complement property).

Let us consider two 8-bit activations, A_a and A_b, two 1-bit weights, W_a and W_b, and the selector *mode* to choose the weight size or mode of operation. The operations to be executed depend on the weights according to table 4.

TABLE 4. Core implemented operation according to mode and weights.

W_b	W_a	Mode = 0	Mode = 1
0	0	X	$X - (A_b + A_a)$
0	1	$X + A_a$	$X - (A_b - A_a)$
1	0	$X + A_b$	$X + (A_b - A_a)$
1	1	$X + (A_a + A_b)$	$X + (A_b + A_a)$

According to these functions, X is being added/subtracted to/from 7-variable functions $A_a, A_b, (A_b + A_a)$, or $(A_b - A_a)$). As explained above, this function has to be reduced to five variables so it can be implemented with a single level of LUTs. Let us consider $Y = X + (A_a + A_b)$ and rewrite the expressions in table 4 terms of Y. The result is in table 5).

TABLE 5. Core implemented operation according to mode and weights using variable $Y = X + A_a + A_b$.

W_b	W_a	Mode = 0	Mode = 1
0	0	$Y - (A_b + A_a)$	$Y - 2 \times (A_b + A_a)$
0	1	$Y - A_b$	$Y - 2 \times A_b$
1	0	$Y - A_a$	$Y - 2 \times A_a$
1	1	$Y - 0$	$Y - 0$

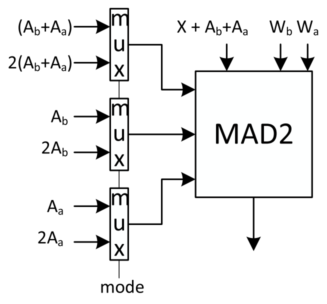

FIGURE 8. MAD2 inputs for 8/1-bit weights.

For the first three combinations of weights, some function f or its double $2 \times f$ is subtracted from Y. For $W_a = W_b = 1$, there is no subtraction (0 is subtracted from Y). Hence, 3 MAD2 inputs are selected using 3 multiplexers depending on the mode (see figure 8). Then, a 5-input (W_a, W_b and the 3 multiplexer outputs) function is subtracted from Y.

Being a 5-input function, it can be implemented with a single level of LUTs.

In the first MAD2 of the cascade $X = 0$, making $Y = A_a + A_b$, which is already the input of the first multiplexer. However, X, which is the result of the previous MAD2 block, needs to be added to A_a+A_b before being input to the MAD2. This would require an extra adder between MAD2s whose outputs are not shared with the MAD2s of other cells. A simpler solution is to do all the required additions beforehand, such that the input of the first MAD2 block is $Y = \sum_{i=0}^{k-1} A_i$, where k is the number of activations in the (sub-)chain of MAD2s. Doing this addition at the beginning avoids adders between MAD2s, and allows Y to be shared by the other DP_i cells in the core.

The extra logic required to add the activations before the MAD2 sub-chain and to implement the multiplexers is shared by all cells of the core and by all cores of the PE cluster. This sharing considerably reduces the overhead associated with the extra logic and makes the solution quite efficient.

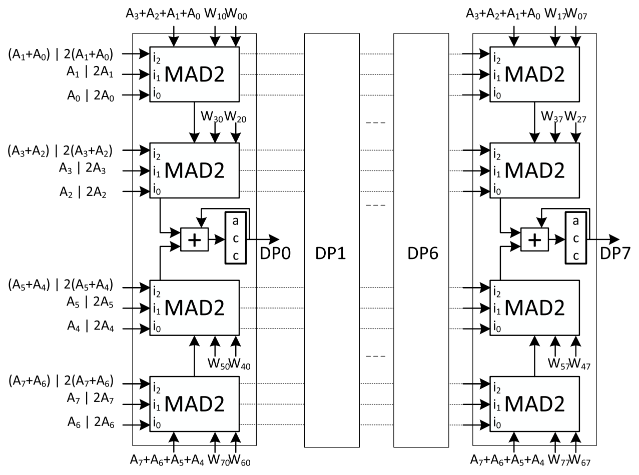

FIGURE 9. Architecture for 8 dot-products with 1-bit weights.

The final C8:81 circuit diagram is illustrated in figure 9. The MAD2 units have the same structure of the MAD units used in the C8:82 core (see figure 4) but function f_i in this case is given by:

$$f_i = \overline{w_1} \cdot \overline{w_0} \cdot i_0 + \overline{w_1} \cdot w_0 \cdot i_1 + w_1 \cdot \overline{w_0} \cdot \overline{i_2} \qquad (10)$$

where signals i_n correspond to the MAD2 inputs shown in figure 9. As in the C8:82 core, the MAD2 chain is divided in two sub-chains of two MAD2 units each, in order to reduce the delay of the complete cell.

V. RESULTS

The proposed hybrid cores have been implemented using the Vivado 2019.1 Design Suite and run on the ZedBoard FPGA card. The Zedboard card comprises a low-density ZYNQ7020 device belonging to the Artix-7 FPGA series, which contains a dual ARM Cortex-A9 CPU. The circuit operates at 200 MHz. To demonstrate its scalability, the proposed hybrid architecture has also been implemented in a Xilinx SoC ZC706 Evaluation Kit, containing a ZYNQ7045 device of the Kintex-7 FPGA series, also featuring a dual ARM Cortex-A9 CPU. With this device, an operating frequency of 240 MHz has been used.

ZYNQ FPGAs have four 64-bit High-Performance ports working at 150 MHz, which give the programmable logic direct access to the external memory. The measured external memory bandwidth is 3.3 GBytes/s on the ZedBoard and 4.2 GBytes/s on the ZC706 board.

All architectures have been described in VHDL, simulated and implemented with Vivado. In particular, the hybrid cores have been described by direct instantiation of the FPGA primitive instances LUT6_2 and CARRY4. The LUT6_2 primitives support equations 8 and 10. Four different architectures have been implemented and tested in FPGA:

1) Architecture 8:8888 - Baseline architecture with both activations and weights represented with 8 bits;
2) Architecture 8:8228 - Hybrid architecture with activations represented with 8 bits. The weights of the first and last layers are represented with 8 bits and the weights of the hidden layers are represented with 2 bits;
3) Architecture 8:8218 - Hybrid architecture with activations represented with 8 bits. The weights of the first and the last layers are represented with 8 bits, the weights of the hidden convolutional layers are represented with 2 bits and the weights of the hidden FC layers are represented with 1 bit;
4) Architecture 8:8118 - Hybrid architecture with activations represented with 8 bits. The weights of the first and last layers are represented with 8 bits and the weights of the hidden layers are represented with 1 bit;

The AlexNet and VGG16 networks have been trained for each hybrid size combination using a framework developed in the scope of this work, which is an extension of Ristretto [29] integrated with Caffe [40]. To reduce the training time of a large set of CNNs with different hybrid sizes, only the new architectures 8:8228, 8:8218 and 8:8118 have been considered. The framework also allowed validating the results of the FPGA implementations. On a first validation level, the outputs of all layers have been compared with their expected values, in order to determine the correctness of the internal node values. On a second validation level, the classification results for each image have been compared with the framework inference results. All networks have been trained for the ImageNet data set [41]; their achieved accuracy is indicated in table 6.

TABLE 6. AlexNet and VGG16 top-1 accuracy for different formats.

Format	AlexNet	VGG16
8:8888	54.7%	67.6%
8:8228	53.2%	65.8%
8:8218	52.7%	65.1%
8:8118	51.0%	63.2%

TABLE 7. Hybrid core resource usage for the different implementations.

	A		B	
	LUTs	DSPs	LUTs	DSPs
C8:88	305	2	411	1
C8:82	442	2	514	1
C8:81	456	4	528	3

There is an accuracy degradation of less than 4.5% between the baseline architecture and the 8:8118 architecture. This is expected since there is a considerable reduction in weight size.

A. AREA RESULTS

The convolutional and fully connected cores determine the FPGA area occupation of the architecture. The cores have been designed so that both LUT and DSP resources of the FPGA are used in a balanced way. To achieve this, different implementations of the cores have been considered with different numbers of DSPs. Table 7 shows the resources occupied by the cores in 2 alternative implementations, *A* and *B*.

TABLE 8. Resource utilization for ZYNQ7020.

AlexNet				
	8:8888	8:8228	8:8218	8:8118
ConvCores	16 × 7	16 × 5	16 × 5	16 × 4
FCcores	2 × 7	2 × 10	1 × 7	1 × 4
BATCH	7	10	7	4
LUTs	46914	48224	42802	42338
BRAMs	126	132	124	124
DSPs	212	200	188	200
VGG16				
	8:8888	8:8228	8:8218	8:8118
ConvCores	16 × 8	16 × 6	16 × 6	16 × 5
FCcores	1 × 1	1 × 2	1 × 1	1 × 1
BATCH	1	2	1	1
LUTs	48151	47450	48540	48736
BRAMs	114	114	112	112
DSPs	216	196	180	216

TABLE 9. Resource utilization for ZYNQ7045.

AlexNet				
	Arq. 8:8888	Arq. 8:8228	Arq. 8:8218	Arq. 8:8118
ConvCores	64 × 7	64 × 6	64 × 6	64 × 5
FCcores	2 × 23	1 × 14	1 × 18	1 × 10
BATCH	23	14	18	10
LUTs	186502	180050	183336	192123
BRAMs	406	386	394	394
DSPs	690	796	840	850
VGG16				
	Arq. 8:8888	Arq. 8:8228	Arq. 8:8218	Arq. 8:8118
ConvCores	64 × 7	64 × 6	64 × 6	64 × 5
FCcores	2 × 6	1 × 3	1 × 2	1 × 2
BATCH	6	3	2	2
LUTs	181432	175188	175240	186280
BRAMs	372	364	362	362
DSPs	572	774	776	796

The overall resource utilization of the architectures when implemented in the ZYNQ7020 and ZYNQ7045 devices is shown in tables 8 and 9, respectively.

The architectures have been configured for best performance and tailored for the target CNN network. All architectures have been designed with a comparable number of resources, for a fair comparison. The batch sizes are larger for AlexNet, since the number of weights of the fully connected layers is almost 30× greater than the number of weights of the convolutional layers. For the VGG16 network, the distribution of weights is evener, allowing for a smaller batch. Additionally, since the number of resources of the hybrid cores varies with the weight size, the total number of cores of the architectures also varies with the weight size. For lower weight sizes, the hybrid core is larger, and fewer cores can be mapped to a particular FPGA device. The PE clusters occupy most of the total resources of the architecture (around 90%), with an area given by the number of cores multiplied by the resources of a single core (tables 8 and 9).

B. PERFORMANCE RESULTS

The performance results obtained when running the inference step for AlexNet are shown in table 10 for all architectures. The hybrid architectures achieve throughputs ranging from 448 to 508 images per second on the ZYNQ7020 device, and from 1429 to 1639 images per second on the ZYNQ7045 device. These results clearly show that, by using hybrid data quantization, it is possible to run large CNNs in low-density FPGAs while meeting real-time requirements (>30 images per second). Compared to the baseline architecture, the best hybrid architecture improves the image throughput by 2.2× on the ZYNQ7020 device and 2.1× on the ZYNQ7045 device. The highest measured performance is 735 GOPS for the ZYNQ7020 and 2.38 TOPS for the ZYNQ7045 with AlexNet. Considering VGG16 the measured performance increases to 1.34 TOPS for the ZYNQ7020 and 2.5 TOPS for the ZYNQ7045. These results come at the cost of a small accuracy reduction: from 1.5 to 3.7%.

In terms of performance efficiency, measured performance/kLUT and measured performance/DSP, the results show that the ZYNQ7020 is more efficient. This has to do with the fact that, in spite of a 4-fold increase in the resources used, when using the ZYNQ7045 instead of the ZYNQ7020, there is not a proportional increase in the external memory bandwidth: from 3.3 to 4.2 GBytes/s only. This shows the importance of the memory bandwidth in the design of CNNs.

For the VGG16 network, the performance results follow a similar trend (table 11). VGG16 is a larger network compared to AlexNet. The computation/communication ratio is larger for VGG16 compared to AlexNet. This explains the better performance efficiency when running VGG16. In spite of this, the 4× more resources of the ZYNQ7045 device do not lead to a 4× faster design, since, as already mentioned, the memory bandwidth does not scale proportionally.

To better understand the influence of the hybrid cores over the execution times of each layer, the average processing times of each layer for the various architectures are shown in figure 10.

TABLE 10. Results for AlexNet inference on ZYNQ FPGAs.

	ZYNQ7020				ZYNQ7045			
	8:8888	8:8228	8:8218	8:8118	8:8888	8:8228	8:8218	8:8118
Images/s	229	448	457	508	781	1429	1639	1639
Performance (GOPS)	332	650	662	735	1132	2070	2375	2375
GOp/kLUT	7.1	13.5	15.5	17.4	6.1	11.5	13.0	12.4
GOp/DSP	1.6	3.2	3.5	3.7	1.6	2.6	2.8	2.8

TABLE 11. Results for VGG16 inference on ZYNQ FPGAs.

	ZYNQ7020				ZYNQ7045			
	8:8888	8:8228	8:8218	8:8118	8:8888	8:8228	8:8218	8:8118
Images/s	12	33	33	43	45	82	84	81
Performance (GOPS)	373	1012	1012	1341	1405	2540	2594	2513
GOp/kLUT	7.8	21.3	20.8	27.5	7.7	14.5	14.8	13.5
GOp/DSP	1.7	5.2	5.6	6.2	2.5	3.3	3.3	3.2

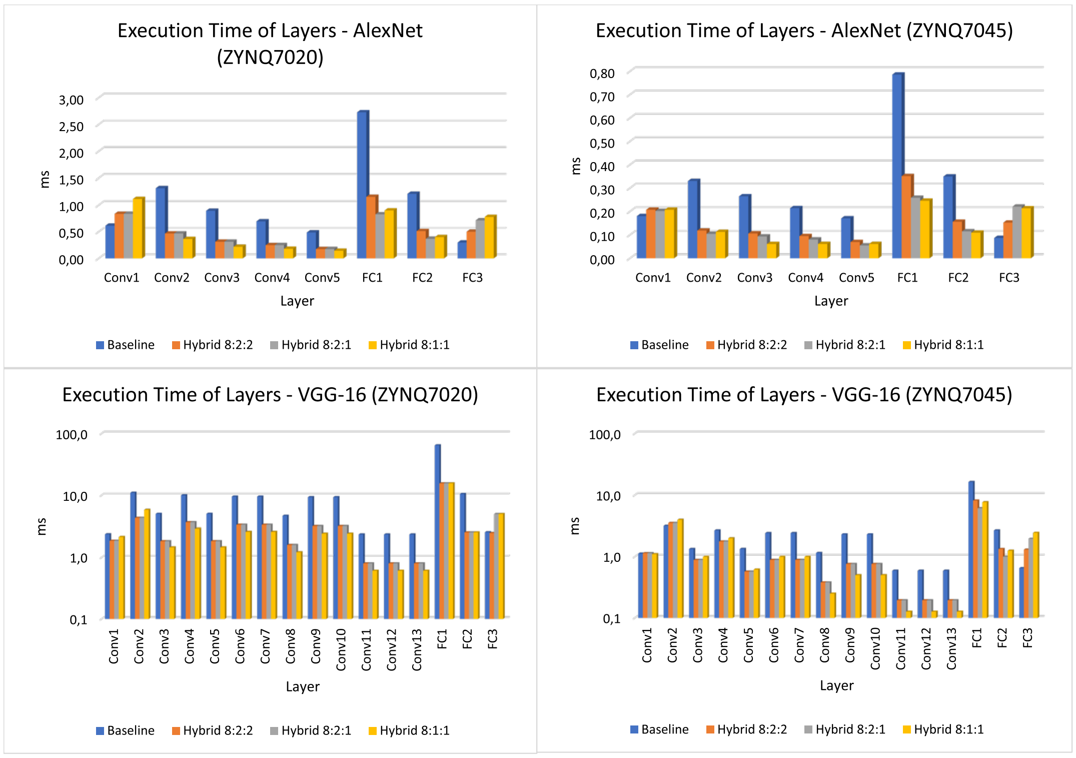

FIGURE 10. Layer execution time for each architecture, CNN and target device.

The first layer always uses 8-bit weights in all architectures. Since the baseline implementation has more cores, the execution time of AlexNet's first layer in the baseline architecture is lower than that of the hybrid architecture. In all other layers, the hybrid architectures achieve the best execution times.

When running VGG16, the hybrid architectures achieve the best execution times for almost all layers. The execution time of the convolutional layers decreases with the weight data size as expected. Exceptions exist for the first and second layers running on the ZYNQ7045 device. For the first layer, the communication time for weights exceeds the execution time. For the second layer, the fact it has only 64 different kernels limits the exposed parallelism, making it useless to increase the number of operations.

In the fully connected layers a significant difference between the baseline and the hybrid architectures is observed, caused by the time to transfer weights from external to internal memories. Another observation is that the 8:8118 hybrid architecture is slower than the other hybrid architectures in the last layers, when running VGG16 on the ZYNQ7045 device. Once again, this is due to the inability of using all the available MAD units in parallel.

TABLE 12. Performance comparison with previous works on ZYNQ7020.

Ref.	Format	LUTs	DSPs	BRAMs	MHz	CNN	Accuracy	GOPS	image/s	GOp/kLUT	GOp/DSP
[42]	fixed-16	35644	208	9	100	LeNet	97.9%	13	—	—	—
[43]	Binary	46900	3	94	143	BNN model	88.4%	208	168	4.4	69.3
[5]	fixed-8	45781	220	132	200	AlexNet	54.7%	133	92	2.9	0.6
8:8888	fixed-8	46914	212	126	200	AlexNet	54.7%	332	229	7.1	1.6
8:8228	hybrid-82	48224	200	132	200	AlexNet	53.2%	650	448	13.5	3.2
8:8218	hybrid-821	42802	188	124	200	AlexNet	52.7%	662	457	15.5	3.5
8:8118	hybrid-81	42338	200	124	200	AlexNet	51.0%	735	508	17.4	3.7
[35]	fixed-8	29867	190	86	214	VGG16	65.6%	84	3	2.8	0.4
8:8888	fixed-8	48151	216	114	200	VGG16	67.6%	373	12	7.8	1.7
8:8228	hybrid-82	47450	196	114	200	VGG16	65.8%	1012	33	21.3	5.2
8:8218	hybrid-821	48540	180	112	200	VGG16	65.1%	1012	33	20.8	5.6
8:8118	hybrid-81	48736	216	112	200	VGG16	63.2%	1341	43	27.5	6.2

TABLE 13. Performance comparison with previous works on ZYNQ7045.

Ref.	Format	LUTs	DSPs	BRAMs	MHz	CNN	Acuracy	GOPS	image/s	GOp/kLUT	GOp/DSP
[3]	fixed-8	86262	808	303	200	AlexNet	54.6%	493	340	5.7	0.6
[3]	8:8218	103505	550	498	200	AlexNet	52.6%	1240	856	11.9	2.2
[44]	fixed-[4, 7]	176192	900	—	150	AlexNet	—	1191	821	6.8	1.3
8:8888	fixed-8	186502	690	406	240	AlexNet	54.7%	1132	781	6.1	1.6
8:8228	hybrid-82	180050	796	386	240	AlexNet	53.2%	2070	1429	11.5	2.6
8:8218	hybrid-821	183336	840	394	240	AlexNet	52.7%	2375	1639	13.0	2.8
8:8118	hybrid-81	192123	850	394	240	AlexNet	51.0%	2375	1640	12.4	2.8
[45]	12:Mixed	58860	576	—	172	VGG16	55.8%	316	9	5.4	0.5
[30]*	8 BFP	231761	1027	913	200	VGG16	68.3%	761	24	3.3	0.7
[46]	fixed-8	114521	680	542	200	VGG16	69.2%	524	55 (pruned)	4.6	0.8
8:8888	fixed-8	181432	572	372	240	VGG16	67.6%	1405	45	7.7	2.5
8:8228	hybrid-82	175188	774	364	240	VGG16	65.8%	2540	82	14.5	3.3
8:8218	hybrid-821	175240	776	362	240	VGG16	65.1%	2594	84	14.8	3.3
8:8118	hybrid-81	186280	796	362	240	VGG16	63.2%	2513	81	13.5	3.2

* Virtex7 VX690T, data format: BFP - block floating-point

C. COMPARISON WITH THE STATE-OF-THE-ART

The performance and area of the proposed hybrid architectures have been compared with the implementations of previous works using the same FPGAs. The overall comparison results are shown in table 12.

For the ZYNQ7020 device, the proposed hybrid architectures increase the image throughput by more than 5.5× compared to the best state-of-the-art approaches for running AlexNet with 8-bit fixed-point data. All the proposed hybrid architectures achieve a measured performance which is 2.5× higher than the binary network architecture for classifying CIFAR-10 images proposed in [43]. For running the VGG16 network, the proposed 8:8228 and 8:8218 architectures achieve an image throughput which is 12× higher than the architecture proposed in [35], while keeping the same accuracy. The 8:8118 architecture achieves a slightly higher throughput (13.7×), at the cost of a 3.5% accuracy degradation.

The architectures have also been compared with previous works implemented on a ZYNQ7045 device, as shown in table 13. For the AlexNet network, the proposed solutions almost double the image throughput with better performance efficiencies. For the VGG16 network, the proposed hybrid architectures also show better performances than previous works, achieving a throughput 1.5× higher than the pruned solution proposed in [46], and 9.3× higher than the solution proposed in [45].

VI. CONCLUSIONS

This work proposes a new configurable architecture for the execution of CNNs, which efficiently supports hybrid data quantization. The architecture targets low-cost FPGAs, and constitutes an advantageous trade-off between performance (or performance efficiency) and accuracy: it significantly boosts performance and performance efficiency in exchange for a low accuracy degradation. Furthermore, its scalability allows taking advantage of larger FPGAs where more cores can be deployed with a proportional performance increase.

Running the AlexNet network, the proposed hybrid quantization architecture achieves a performance of 735 GOPS on a ZYNQ7020 FPGA, and 2.375 TOPS on a ZYNQ7045 FPGA. For the VGG16 network, the measured performance increases to 1.341 TOPS on a ZYNQ7020 FPGA, and 2.513 TOPS on a ZYNQ7045 FPGA. These results clearly show that it is possible to run large high-performance CNNs on low-density FPGAs for embedded devices, which is a technological contribution to enable the deployment of large CNNs on edge nodes and end devices.

To extend the applicability of the proposed solution to irregular networks, the architecture is now being modified to support particular convolutional layers that exist in irregular networks. Two interesting such layers are the GoogleNet Inception and the ResNet Residual layers. Also, hybrid quantization of both weights and activations is being studied in terms of network accuracy and architectural design.

REFERENCES

[1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, ''ImageNet large scale visual recognition challenge,'' *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, Dec. 2015.

[2] M. Véstias, *Deep Learning on Edge: Challanges and Trends*. Hershey, PA, USA: IGI Global, 2020, pp. 23–42.

[3] J. Wang, Q. Lou, X. Zhang, C. Zhu, Y. Lin, and D. Chen, ''Design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA,'' in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2018, pp. 163–1636.

[4] Y. Zhao, X. Gao, X. Guo, J. Liu, E. Wang, R. Mullins, P. Y. K. Cheung, G. Constantinides, and C.-Z. Xu, ''Automatic generation of multi-precision multi-arithmetic CNN accelerators for FPGAs,'' in *Proc. Int. Conf. Field-Program. Technol. (ICFPT)*, Dec. 2019, pp. 45–53.

[5] M. Véstias, R. P. Duarte, J. T. de Sousa and H. Neto, ''Lite-CNN: A high-performance architecture to execute CNNs in low density FPGAs,'' in *Proc. 28th Int. Conf. Field Program. Logic Appl. (FPL)*, Dublin, Ireland, Aug. 2018, pp. 399–3993, doi: 10.1109/FPL.2018.00075.

[6] Y. Le Cun, L. D. Jackel, B. Boser, J. S. Denker, H. P. Graf, I. Guyon, D. Henderson, R. E. Howard, and W. Hubbard, ''Handwritten digit recognition: Applications of neural network chips and automatic learning,'' *IEEE Commun. Mag.*, vol. 27, no. 11, pp. 41–46, Nov. 1989.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, ''ImageNet classification with deep convolutional neural networks,'' in *Proc. 25th Int. Conf. Neural Inf. Process. Syst. (NIPS)*, vol. 1. Red Hook, NY, USA: Curran Associates, 2012, pp. 1097–1105.

[8] K. Simonyan and A. Zisserman, ''Very deep convolutional networks for large-scale image recognition,'' in *Proc. 3rd Int. Conf. Learn. Represent.*, 2015.

[9] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, ''Going deeper with convolutions,'' in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1–9.

[10] K. He, X. Zhang, S. Ren, and J. Sun, ''Deep residual learning for image recognition,'' in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[11] M. P. Véstias, ''A survey of convolutional neural networks on edge with reconfigurable computing,'' *Algorithms*, vol. 12, no. 8, p. 154, Jul. 2019.

[12] M. A. Dias and D. A. P. Ferreira, ''Deep learning in reconfigurable hardware: A survey,'' in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2019, pp. 95–98.

[13] M. Alawad and M. Lin, ''Scalable FPGA accelerator for deep convolutional neural networks with stochastic streaming,'' *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 4, no. 4, pp. 888–899, Oct. 2018.

[14] J. Zhang and J. Li, ''Improving the performance of opencl-based FPGA accelerator for convolutional neural network,'' in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2017, pp. 25–34.

[15] S. I. Venieris and C. Bouganis, ''fpgaConvNet: Mapping regular and irregular convolutional neural networks on FPGAs,'' *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 2, pp. 326–342, Feb. 2018.

[16] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He, ''OPU: An FPGA-based overlay processor for convolutional neural networks,'' *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 1, pp. 35–47, Jan. 2019.

[17] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, ''A dynamically configurable coprocessor for convolutional neural networks,'' *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 247–257, Jun. 2010.

[18] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, ''DaDianNao: A machine-learning supercomputer,'' in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2014, pp. 609–622.

[19] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, ''Going deeper with embedded FPGA platform for convolutional neural network,'' in *Proc. 2016 ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, New York, NY, USA, 2016, pp. 26–35.

[20] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-S. Seo, and Y. Cao, ''Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks,'' in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, New York, NY, USA, 2016, pp. 16–25.

[21] Y. Qiao, J. Shen, T. Xiao, Q. Yang, M. Wen, and C. Zhang, ''FPGA-accelerated deep convolutional neural networks for high throughput and energy efficiency,'' *Concurrency Comput. Pract. Exper.*, vol. 29, no. 20, p. e3850, Oct. 2017.

[22] X. Hu, Y. Zeng, Z. Li, X. Zheng, S. Cai, and X. Xiong, ''A resources-efficient configurable accelerator for deep convolutional neural networks,'' *IEEE Access*, vol. 7, pp. 72113–72124, 2019.

[23] A. Gonçalves, T. Peres, and M. Véstias, *Exploring Data Size to Run Convolutional Neural Networks in Low Density FPGAs*. Cham, Switzerland: Springer, 2019, ch. 27, pp. 387–401.

[24] T. Peres, A. Gonçalves, and M. Véstias, *Faster Convolutional Neural Networks in Low Density FPGAs Using Block Pruning*. Cham, Switzerland: Springer, 2019, ch. 28, pp. 401–402.

[25] Z. Liu, Y. Dou, J. Jiang, J. Xu, S. Li, Y. Zhou, and Y. Xu, ''Throughput-optimized FPGA accelerator for deep convolutional neural networks,'' *ACM Trans. Reconfigurable Technol. Syst.*, vol. 10, no. 3, pp. 17:1–17:23, Jul. 2017.

[26] D. T. Nguyen, T. N. Nguyen, H. Kim, and H.-J. Lee, ''A high-throughput and power-efficient FPGA implementation of YOLO CNN for object detection,'' *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 8, pp. 1861–1873, Aug. 2019.

[27] Y. Shen, M. Ferdman, and P. Milder, ''Maximizing CNN accelerator efficiency through resource partitioning,'' in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, Jun. 2017, pp. 535–547, doi: 10.1145/3079856.3080221.

[28] L. Gong, C. Wang, X. Li, H. Chen, and X. Zhou, ''MALOC: A fully pipelined FPGA accelerator for convolutional neural networks with all layers mapped on chip,'' *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2601–2612, Nov. 2018.

[29] P. Gysel, M. Motamedi, and S. Ghiasi, ''Hardware-oriented approximation of convolutional neural networks,'' in *Proc. 4th Int. Conf. Learn. Represent.*, 2016, pp. 1–27.

[30] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji, ''High-performance FPGA-based CNN accelerator with Block-Floating-Point arithmetic,'' *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 8, pp. 1874–1885, Aug. 2019.

[31] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, ''HAQ: Hardware-aware automated quantization with mixed precision,'' in *Proc. CVPR*, 2018, pp. 8612–8620.

[32] M. P. Vestias, R. Policarpo Duarte, J. T. de Sousa, and H. Neto, ''Hybrid dot-product calculation for convolutional neural networks in FPGA,'' in *Proc. 29th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2019, pp. 350–353.

[33] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, ''FINN: A framework for fast, scalable binarized neural network inference,'' in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, New York, NY, USA, 2017, pp. 65–74, doi: 10.1145/3020078.3021744.

[34] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, ''FP-BNN: Binarized neural network on FPGA,'' *Neurocomputing*, vol. 275, pp. 1072–1086, Jan. 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231217315655

[35] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, ''Angel-eye: A complete design flow for mapping CNN onto embedded FPGA,'' *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 1, pp. 35–47, Jan. 2018.

[36] B. Khabbazan and S. Mirzakuchaki, ''Design and implementation of a low-power, embedded CNN accelerator on a low-end FPGA,'' in *Proc. 22nd Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2019, pp. 647–650.

[37] M. P. Véstias, R. P. Duarte, J. T. de Sousa, and H. C. Neto, ''Fast convolutional neural networks in low density FPGAs using zero-skipping and weight pruning,'' *Electronics*, vol. 8, no. 11, p. 1321, Nov. 2019.

[38] O. Macsorley, ''High-speed arithmetic in binary computers,'' *Proc. IRE*, vol. 49, no. 1, pp. 67–91, Jan. 1961.

[39] E. G. Walters, ''Array multipliers for high throughput in Xilinx FPGAs with 6-input LUTs,'' *Computers*, vol. 5, no. 4, p. 20, 2016. [Online]. Available: http://www.mdpi.com/2073-431X/5/4/20

[40] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, ''Caffe: Convolutional architecture for fast feature embedding,'' 2014, *arXiv:1408.5093*. [Online]. Available: http://arxiv.org/abs/1408.5093

[41] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, ''ImageNet: A large-scale hierarchical image database,'' in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2009, pp. 248–255.

[42] S. I. Venieris and C.-S. Bouganis, ''FpgaConvNet: A framework for mapping convolutional neural networks on FPGAs,'' in *Proc. IEEE 24th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2016, pp. 40–47.

[43] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional neural networks with software-programmable FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2017, pp. 15–24, doi: 10.1145/3020078.3021741.

[44] A. Kouris, S. I. Venieris, and C.-S. Bouganis, "Cascadeˆ CNN: Pushing the performance limits of quantisation in convolutional neural networks," in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2018, pp. 155–1557.

[45] J. Wang, J. Lin, and Z. Wang, "Efficient hardware architectures for deep convolutional neural network," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 6, pp. 1941–1953, Jun. 2018.

[46] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-M. Hwu, and D. Chen, "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, New York, NY, USA, Nov. 2018, p. 56, doi: 10.1145/3240765.3240801.

MÁRIO P. VÉSTIAS received the Ph.D. degree in electrical and computer engineering from the Technical University of Lisbon. He is currently a Coordinate Professor with the Department of Electronic, Telecommunications and Computer Engineering (DEETC), School of Engineering (ISEL), Polytechnic Institute of Lisbon. He is also a Senior Researcher with the Electronic Systems Design and Automation Group, INESC-ID, Lisbon. His main research interests are computer architectures and digital systems for high-performance embedded computing, with an emphasis on reconfigurable computing.

RUI P. DUARTE (Member, IEEE) received the Ph.D. degree from Imperial College London, U.K., in 2014. He is currently a Researcher with the Electronic Systems Design and Automation (ESDA), INESC-ID, Lisbon, Portugal. His research interests include reconfigurable computing, fault-tolerant, and low-power architectures.

JOSÉ T. DE SOUSA (Member, IEEE) is currently a Lecturer with the Department of Electrical and Computer Engineering, School of Engineering (IST), University of Lisbon, and a Senior Researcher with the Electronic Systems Design and Automation research group, INESC-ID, a research institute associated with IST, since 1999. He is also a tech entrepreneur in the area of semiconductor intellectual property, having founded and managed three companies: Coreworks (2001–2013, co-founder and CEO), IPbloq (2017–2019, co-founder and CEO), IObundle (2018–present, owner and founder). His main interests are digital systems design and computer architecture, with emphasis on reconfigurable computing. He holds four international patents, is coauthor of one book, and has published more than 70 technical papers in international journals and conferences. He was a General Chair of the 2013 Field Programmable Logic and Applications Conference, and a co-editor of its proceedings and a related special issue on the IEEE TRANSACTIONS ON COMPUTERS journal.

HORÁCIO C. NETO received the Ph.D. degree in electrical and computer engineering from the Technical University of Lisbon. He is currently an Associated Professor with the Department of Electrical and Computer Engineering (DEEC), School of Engineering (IST), University of Lisbon. He is responsible for the Electronic Systems Design and Automation (ESDA) research group, INESC-ID, a research institute associated with the engineering university, IST. His main research interests are digital systems design and computer architecture, with emphasis in reconfigurable computing.