Journal of Signal Processing Systems (2021) 93:531–544
https://doi.org/10.1007/s11265-020-01606-2

Efficient Design of Pruned Convolutional Neural Networks on FPGA

Mário Véstias[1]

Received: 21 April 2020 / Revised: 21 April 2020 / Accepted: 8 October 2020 / Published online: 14 November 2020

Abstract
Convolutional Neural Networks (CNNs) have improved several computer vision applications, like object detection and classification, when compared to other machine learning algorithms. Running these models in edge computing devices close to data sources is attracting the attention of the community since it avoids high-latency data communication of private data for cloud processing and permits real-time decisions turning these systems into smart embedded devices. Running these models is computationally very demanding and requires a large amount of memory, which are scarce in edge devices compared to a cloud center. In this paper, we proposed an architecture for the inference of pruned convolutional neural networks in any density FPGAs. A configurable block pruning method is proposed together with an architecture that supports the efficient execution of pruned networks. Also, pruning and batching are studied together to determine how they influence each other. With the proposed architecture, we run the inference of a CNN with an average performance of 322 GOPs for 8-bit data in a XC7Z020 FPGA. The proposed architecture running AlexNet processes 240 images/s in a ZYNQ7020 and 775 images/s in a ZYNQ7045 with only 1.2% accuracy degradation.

Keywords Deep learning · Convolutional neural network · FPGA · Block pruning · Edge computing

1 Introduction

Deep neural networks (DNN) have shown very promising achievements in computer vision applications, like object detection and classification [1]. The convolutional neural network (CNN) is a type of DNN used to classify images and one of the most researched and deployed deep neural network. Through the identification of correlations among pixels a CNN is able to classify the object present in an image as belonging to a pre-determined class.

CNNs differ from the other DNN models since they use a particular class of layers known as convolutional. These layers apply a set of 3D convolutions between 3D kernels of weights and the maps of a previous layer to produce a set of output maps for the next layer. A sequence of these layers identifies features of the image whose complexity increases with the depth of the network. In the final layers of a CNN all features are associated to class with a certain probability.

One of the first CNNs was LeNet [2] with a total of 60K weights distributed by five layers. The network was applied for digit classification with small images. AlexNet [3], a deeper and more complex CNN, was presented in the ImageNet Challenge for image classification, with eight layers with a total of 61M weights and 724 MAC (Multiply-ACcumulate) operations to process images of size 224 × 224 × 3. With a Top-5 error rate around 15 % it attracted the attention to deep neural networks as a very promising machine learning method. Other models have followed with better accuracies. VGG-16 [4] with 16 layers, 2.2× more weights than AlexNet and 15.5 GMAC (Giga Multiply-Accumulate) operations improved the Top-5 error to around 7%. GoogleNet [5] introduced the inception module, a new type of layer, 1 × 1 convolutions and other optimizations that lead to improvements in the accuracy for image classification. The number of layers of CNNs kept increasing as a way to improve accuracy. ResNet [6], the first CNN exceeding human level accuracy, has 152 layers.

Running these machine learning models on the edge close to data sources is essential for the future of big data processing since it avoids a high-latency communication of private data for cloud processing and permits real-time decisions on the edge. However, a common feature of CNN models is the high number of weights and operations. Therefore, running algorithms that are quite demanding in terms of computing and storage requirements in devices where these resources are scarce is a challenging task [7].

✉ Mário Véstias
mvestias@deetc.isel.ipl.pt

[1] INESC-ID, Instituto Superior de Engenharia de Lisboa, Instituto Politécnico de Lisboa, Lisbon, Portugal

Two main research directions are being followed for the deployment of CNNs in edge computing platforms: model optimization and new computing architectures and platforms. Model optimization includes all methods used to reduce the number of operations and weights of the network. This will reduce proportionally the computational complexity and the storage requirements of the target computing platform. On the other side, dedicated architectures improve the performance, the energy and the cost of the computing platform. Most of previous proposals in either of these dimensions do not follow a collaborative design approach. Model optimizations do not consider the target platform and computing architectures are designed for specific models. An integrated design approach where both the model and the architecture are designed in common is more effective and produces better results.

In this paper, we propose a hardware-oriented pruning of convolutional neural networks. Pruning is a well-known method for model optimization that prunes connections between layers to reduce the number of operations and weights. The method by itself is quite efficient but introduces sparsity in the matrices of weights, which reduces the computational efficiency of the regular pipelined structures of computing datapaths. To overcome the sparsity problem caused by pruning, a block pruning technique adapted to the pipelined datapath of the architecture is proposed in this paper together with a dedicated architecture with support for pruned networks. The architecture also supports image batch as a complement method of pruning. The correct integration of both methods allows the implementation of efficient architectures with high image throughput and high network accuracy.

The proposed architecture extends a high performance architecture for CNN inference proposed in [8] with block pruning. The following has been considered for the design of the proposed architecture:

- A new coarser grained pruning method of CNNs where weights are removed in groups;
- A deep learning flow based on Caffe [9] and Ristretto [10] to optimize networks with block pruning and data quantization;
- Extension of a baseline architecture with configurable block pruning and batch.

The paper is organized as follows. Section 2 describes the state of art on FPGA implementations of CNNs and optimization methods based on pruning. Section 3 explains the fundamentals of convolutional neural networks. Section 4 describes the baseline architecture. Section 5 describes the block pruning method and the flow used to explore block pruning and quantization. Section 6 describes the proposed architecture with support for both pruning and batch. Section 7 reports the results on inference accuracy and area/performance of the proposed architecture running AlexNet on two different SoC (System on Chip) FPGAs (ZYNQ7020 and ZYNQ7045). Section 8 concludes the paper.

2 Related Work

Computing platforms for deep learning on edge cannot rely on high-performance devices since they either have high energy consumption or low power efficiency and are relatively expensive [11]. Embedded processors are used in many low cost devices but achieve only a few dozen GFLOPs (Giga FLoating-point Operations Per second) with low power efficiency, insufficient for real or almost real-time processing of CNNs. Embedded GPUs (Graphics Processing Units) offer thousands of GFLOPs at acceptable power but usually higher than that available in a low energy embedded platform. Dedicated hardware solutions can achieve the best performance and energy efficiency as long as the power and cost are acceptable for a low cost embedded device. Dedicated hardware solutions with application specific integrated circuits (ASICs) are the most efficient. Edge TPU from Google [12] is a processor for embedded inference applications. It has a peak performance of 4 TOPS and consumes 0.5 TOPS/W. The high performance and high energy efficiency is the result of a dedicated chip for inference. However, the performance efficiency (measured performance over peak performance) greatly depends on the network. For example, the inference of an image (224×224) with VGG16, MobileNetv1 and SqueezeNet takes 343, 2.4 and 2 ms [13], respectively. This corresponds to a performance efficiency of less than 2.3%, 2.6% and 9%, respectively. So, the performance efficiency is low and variable with the network model. The problem is that ASIC-based devices are very limited in terms of configurability and thus unable to adapt the processing datapath to each particular network and model optimizations. Reconfigurable computing is an alternative to ASIC solutions providing configurability and high performance. A few authors have proposed the implementation of CNN on coarse-grained arrays [14], while most of the reconfigurable computing proposals for CNNs target FPGAs (Field Programmable Gate Arrays).

FPGAs permit the design of dedicated hardware accelerators for each specific neural network model. FPGAs are less efficient but very hardware flexible and can be tailored for each particular neural network. FPGAs run CNN inference with high performance efficiency, because they can be reconfigured to best implement each different CNN. Depending on the FPGA family, hundreds or even thousands of sustained GOPs (Giga Operations per second) were already obtained in the execution of a CNN inference. FPGA implementations of CNNs started with small

networks [15, 16] and/or considering only convolutional layers [17] and now full CNN implementations [18] and automatic tools for the generation of CNN accelerators [19] are available.

FPGA implementations with dedicated accelerators for the whole CNN model were proposed in [20, 21]. The former uses an architecture similar to that proposed in [17]. Any of these architectures can run any convolutional layer with different features, including the size of convolution windows that may vary for different layers. To ensure all this flexibility the performance efficiency of the architecture varies with the window sizes of convolutions since a fixed structure is used to run all different convolution sizes. Some authors proposed solutions to eliminate this variation in efficiency. Suda et al. [21] rearrange the input maps of the layer to be executed as matrix multiplications. Convolutions are executed with the same efficiency independently of the window size. However, it has a large overhead associated with the memory accesses and execution times necessary to rearrange the input maps. This overhead was partially eliminated in [22] that also uses an accelerator for matrix multiplication and dedicated units to convert the inputs maps into a matrix.

Instead of using the same hardware block to execute different layers, the authors in [23] proposed an architecture with a dedicated hardware module for each layer in a pipeline datapath. The approach removes efficiencies caused by different window sizes but requires other techniques such as fused layers [24] to account for the extra memory required to store intermediate maps and weights. The solution is not appropriate for models with a large number layers. To overcome this limitation, a mixed approach was proposed in [25], where the architecture has several execution modules, each running a subset of the layers.

To reduce the complexity of the accelerators, several optimization techniques have been proposed. One of the most used simplification is data size reduction where smaller and simpler representations are used for activations and weights [26]. In [27], an analysis of several CNNs have shown that data represented with 8 bit fixed-point preserves the accuracy close to that obtained with data represented with 32-bit floating-point. A deeper optimization allows different fixed-point scales and bitwidths for different layers [28, 29]. Data quantization can be taken to the limit with some or all data represented in binary. In general, there is a trade-off between the network size and precision. In [30] this trade off was studied and a binarized neural network requires 2 to 11 times more operations and weights than a CNN with 8-bit fixed-point weights to achieve a comparable accuracy on a small network.

Another class of model optimizations is known as data reduction, a set of techniques to reduce the number of weights and/or activations of the network model in order to reduce the necessary memory and the number of operations to execute the model. Data reduction uses pruning to reduce the number of weights and data compression to additionally reduce the size of memory storage. In [31] deep neural networks are pruned and compressed with Huffman coding. Applied to fully connected layers of AlexNet permitted a 91% reduction in the number of weights with a negligible effect over the accuracy. Pruning introduces sparsity so in [32] the pruning is adapted to the underlying hardware arithmetic unit. The method was tested with different processors, including a microcontroller, a general-purpose processor and a GPU. Considering a GPU, the pruning method improved the performance by 25% and reduced the model size by 53%.

Pruning applied to fully connected layers is very efficient since it reduces considerably the number of weigths and the number of operations in proportion. In convolutional layers, weight pruning is less efficient. Instead of pruning weights some approaches rely on the fact that the ReLU function reduces many output activations to zero. So instead of weight pruning, they implicitly prune zero valued activations while running the layers. The method permits to reduce the number of operations if the multiplication by zero is skipped. A zero-skiping method was proposed in [33] and implemented in ASIC. The same work additionally considers explicit pruning by dynamic zeroing activation outputs whenever the value is close to zero within a threshold. The problem of the proposed architecture is that it requires large on-chip memory, which is not adequate for low density devices, and only considers optimizations in convolutional layers. In [34] the focus is over fully connected weights skipping both weights and activations that are zero while [35] can skip both weights and activations only in convolutional layers.

In [36], an architecture implemented in FPGA was proposed that dynamically skips computations with zeros similar to what is done in [33]. The authors kept a dense format to store the matrix still requiring that all weights be loaded from memory. The NullHop architecture proposed in [37] also exploits the sparsity of convolutional feature maps, consequence of zero activations generated by the ReLU activation function. The architecture only takes advantage of zero activations to reduce memory size using a sparse matrix compression scheme. The number of operations is kept the same since it does not take advantage of zero activations to avoid the execution of multiplications. In [38] and [39] the proposed method skips zeros present in weights instead of zeros present in the feature maps. The work in [39] considers one weight of each kernel at a time. In [40] both pruning and zero skipping are used to improve the performance of an accelerator for low density FPGAs.

The migration of deep learning to the edge took some authors to move their focus to low density FPGAs [41]. In

[42], small CNNs were implemented in a small ZYNQ7020 with an average performance of 13 GOPs with weights represented as 16 bit fixed-point data. In [43] the authors proposed an architecture to run large CNN models, such as VGG16, in ZYNQ7020 with data represented with 8 bits. The proposal considers a parameterizable hardware module that is run-time reconfigured to run different layers. Data in layers are quantized with the best fixed-point scale and bit width, showing that 8-bits are enough for state-of-the-art networks. The architecture achieves a performance of 84 GOPs. A fully pipelined FPGA accelerator for CNNs with data represented with 16-bit in fixed-point format and a layer-fused technique was proposed in [44] achieving 80 GOPS in a ZYNQ7020 FPGA. A very efficient accelerator for convolutional neural networks was proposed in [45]. The solution achieves 385 GOPs running large CNNs. In [46] pruning of fully connected layers is used to design an efficient accelerator for the inference of CNNs in the small ZYNQ7020 SoC FPGA. The architecture proposed in [47] is able to skip both input feature maps zeros and zero weights that are present in both convolutional and fully connected kernels. The solution requires a data dispatcher module for each processing module and large on-chip memory bandwidth local to each core to search for non-zero weights and activations, reducing hardware efficiency.

The architecture proposed in this paper efficiently integrates fixed-point quantization, pruning and batch in FPGA. The trade off between accuracy, area and performance is determined with an integrated design of both model optimization with pruning and hardware datapath.

3 Convolutional Neural Network

A convolutional neural network is a type of deep neural network used for image classification and object recognition [48]. What differentiates a CNN from the other deep learning models is the utilization of convolutional layers. The convolutional layer considers the spatial correlation between neighbor neurons to establish dependencies between them, that is, the output of a neuron is the result of the convolution between a small window of weights and the associated weights.

A convolutional layer receives several input feature maps (IFM) from the previous layer and generates output feature maps (OFM) to the next layer. There is a window of weights for each input feature map. A stack of windows of weights forms one 3D kernel. In a convolutional layer, a 3D kernel slides over the 3D block of IFMs and produces one OFM. Several 3D kernels are used in each convolutional layer, each extracting different features from the IFMs. The number of output feature maps of each layer is therefore the same as the number of kernels at that layer. Convolutional layers are computationally very demanding since the same kernel has to be applied throughout the whole block of input feature maps and each layer considers many different kernels.

A CNN has several convolutional layers and each convolutional layer may be followed by a pooling layer that determines the average or max of a window of neurons. This layer reduces the complexity of the output feature maps and achieves translation invariance and reduce over-fitting. The last layers of a CNN are the fully connected (FC) or dense layers interconnecting all neurons of previous layers so that complex features extracted by the convolutional layers are globally correlated. In each fully connected layer there are also several kernels. However, in dense layers these kernels are only applied once to the input map. Consequently, dense layers are not computationally intensive since each kernel is only used once. Instead the bottleneck of dense layers is the large number of weights to be transferred from external memory.

CNNs having these three types of layers are known as regular. Examples of regular networks include AlexNet and VGG16. Some proposals consider other types of layers, like combinations of convolutional layers, in their CNNs. Networks like GoogleNet [5] and ResNet [6] are referred to as irregular networks since they contain composite layers different from the three above.

Regular networks, like AlexNet and VGG16, have most of the weights (90%) concentrated in the fully connected layers. In these networks, pruning of fully connected layers is more effective then in convolutional layers. The method reduces the required memory to store weights and allows zero skipping to reduce the required computing resources. The problem of pruning is the introduction of sparsity in the matrix of weights. The block pruning method proposed in this paper reduces the pruning granularity to allow a more suitable hardware implementation of a parallel datapath in order to avoid the need for complex data dispatch units requiring large memory bandwidth.

4 Baseline Architecture Overview

The architecture proposed in this paper modifies a baseline state of the art architecture to support block pruning. This baseline architecture is a follow-up of the work presented in [8] and is described in this section.

Knowing that each type of layer has different characteristics, the baseline architecture has one hardware module for convolutional layers and another for dense layers (see Figure 1).

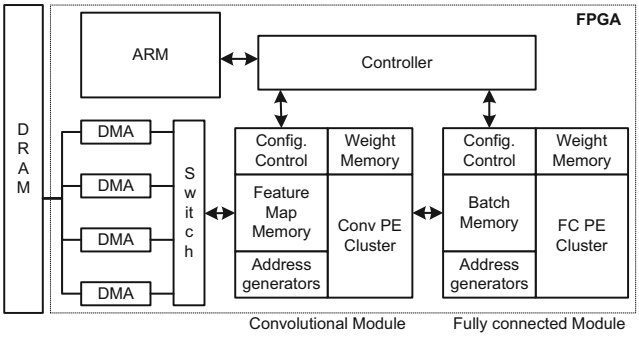

Figure 1 Block diagram of the baseline architecture.

This hardware partition allows different optimization techniques to be applied to convolutional and fully connected layers. In the proposal of this paper, the pruning technique is only applied to fully connected layers.

Each module is independent of each other and executes one layer at a time. Before running a layer, the module is configured with the specific features of the layer, namely the number of kernels, the size of kernels, addresses of feature maps and kernels in the on-chip memory, the optional execution of pooling after a convolutional layer and the fixed-point format that can be different for different layers. The convolutions and inner products of dense layers read the input feature maps from on-chip memory. If the on-chip memory is not enough to store the initial image or the input feature maps, these are divided and processed in parts. The output feature maps are stored in external memory and then reloaded for the next layer. This memory hierarchy permits the execution of large CNNs in FPGAs even when the on-chip memory is insufficient to store the whole maps.

Convolutional and fully connected modules consist of a matrix of processing elements (PE) with local memory to store kernels of weights. Pooling and activation functions run in a central module shared by all processing elements (see Figure 2).

The execution of a CNN in the baseline architecture works as follows:

1. Each module is configured according to the layer features;
2. The image or the input feature maps are loaded to the on-chip memory (Feature Map Memory - FMM);
3. Weigths are loaded into the local memories of the PE cluster;
4. PE clusters execute the convolutions and dot-products;
5. The results are sent to the pooling and activation module;
6. The new OFMs are stored in external memory;
7. Go to step 3 if more kernels need to be loaded;
8. The whole process repeats for each layer.

External memory accesses are done by direct memory access (DMA) blocks that read feature maps and the kernels from external memory and write the feature maps back to external memory.

The PE cluster for convolutional layers explores several forms of parallelism:

1. Inter-output parallelism: Output feature maps are independent and so are calculated in parallel. Different cores of the cluster operating over the same activations with different kernels contribute to different output maps in parallel (line of cores);
2. Intra-output parallelism: Activations of an output feature map are independent and can be calculated in parallel. Different cores of the cluster operating over different activations with the same kernel contribute to different activations of an output map in parallel (columns of cores);
3. Kernel parallelism: The inner product of a single kernel can be calculated in parallel. Each core has several multiply-accumulate (MAC) units that calculate the kernel convolution in parallel. The number of parallel MAC is configurable and determines the data port size of memories.

The baseline architecture calculates convolutions independently of the convolution window size by considering 3D convolutions with activations and weights as long vectors, as explained in [8]. Each activation of an output feature is obtained from the dot product between a 3D kernel $x_k \times y_k \times z_k$ and the correspondent activations of the IFMs of size $x_p \times y_p \times z_p$, where z_p is the number of IFMs. Formally, the dot product to calculate each step of the convolution is given by:

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

where *startAddr* is the address of the first activation of the block of the input feature map being convolved. This operation is used to convolve a kernel with the set of input feature maps sliding the 3D kernel along the feature maps. If a layer is followed by a pooling layer, the output activations of the pooling window are pooled and only the pooling result is stored in the FMM. The advantage of the proposed method is that it is independent of the size of the kernels and of the size of the convolution window.

For example, considering a kernel of size $3\times3\times128$, each output neuron is calculated as the inner product of the kernel with a block of $3 \times 3 \times 128$ activations. The 3D kernel slides over the whole input map spanning the $x_p \times y_p$ space. When there is a stride higher than one, the 3D kernel slides over the input map stepping over some input activations according to

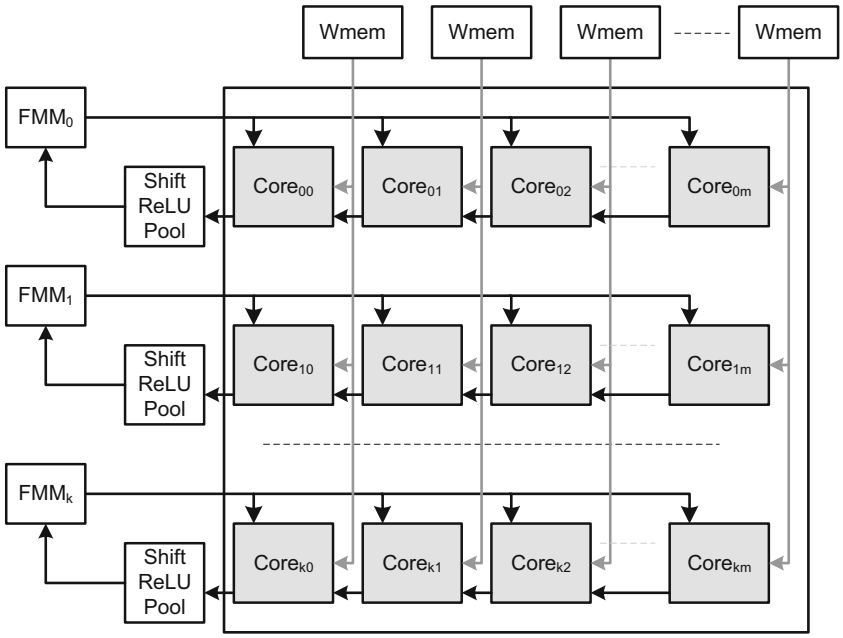

Figure 2 Architecture of the PE cluster for convolutional layers.

the stride size. For example, a stride of two means that the slide shifts by two, reducing the output map to half the size of the input map. The algorithm also includes the optional pooling step. In this case, the output neuron is generated only after determining all neurons of the pooling window, that is, the algorithm sequentially determines all neurons of the pooling window so that the pooling layer can be merged with the convolutional layer.

The PE cluster for dense layers also explores intra-output and kernel parallelism, but inter-output parallelism does not apply to dense layers. However, the baseline architecture supports feature map batch, in which several maps from the convolutional layers are batched before running the first dense layer. When batch is considered, the architecture also explores inter-output parallelism since multiple output maps are generated in parallel.

The fully connected cluster 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 batched in the batch memory. Compared to the convolutional module, there is a major simplification in the address generators since, insteads of an image convolution, there is a single dot product between the whole batched image and the kernel. Multiple dot products between a batched image and different kernels are possible by using multiple cores in a line of the cluster. This number is limited by the available memory bandwidth and is in general much lower than in the PE cluster for convolutions. This architectural difference between the two clusters of PE is the main reason for having independent modules for convolutional and fully connected layers, improving the hardware efficiency.

5 Block Pruning of Dense Layers

Pruning introduces sparsity in the kernels. Sparse kernels of a layer have pruned weights in different indices. This complicates the exploration of kernel and intra-output parallelism since several activations of the input map have to be accessed at sparse addresses at the same time. This requires multi-port memories or memory content replication. Sparsity also introduces an overhead associated with the index information of the sparse vector of weights.

To improve the hardware implementation and the performance of pruned networks we introduce the block pruning method which performs a coarse pruning of blocks of weights. The method reduces the index overhead and the required number of memory ports. The technique permits to prune blocks of weights instead of just single weights (see example in Figure 3).

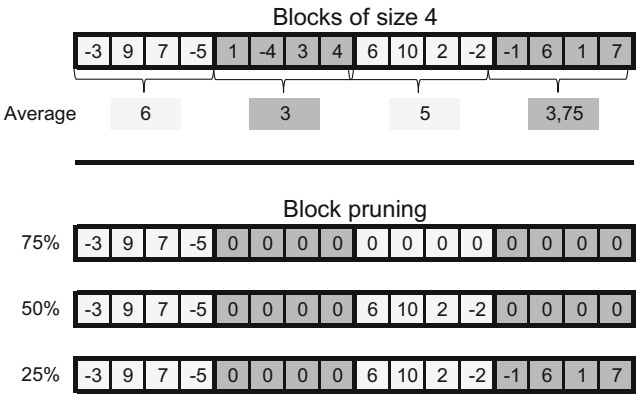

Figure 3 Pruning method for blocks of four weights.

The proposed method determines the average magnitude of a block of weights, sort them and then the blocks with the lowest average magnitude are pruned limited by a pruned percentage. The remaining blocks are stored as a sparse vector where each position contains the block of weights and the index of the next block.

A framework to explore the block pruning of weights in fully connected weights followed by data quantization was developed based on Caffe [9] as the main framework and Ristretto for data quantization.

Pruning can be implemented with different metrics and methods to reduce the number of weights. In this work we have considered the weights magnitude. A percentage of weights whose magnitude is closer to zero is iteratively removed according to the flow in Figure 4.

In the first step we train the network or start with a pre-trained network. After training, consecutive weights of the kernel are grouped in blocks of a predetermined size and the average of each block is determined. Then, starts an iterative pruning of a percentage p of the blocks with the lowest average. While the difference between the original accuracy and the accuracy of the pruned network is bellow a threshold, the algorithm iteratively increases the percentage of pruning. When no more pruning is allowed, a datawidth reduction step follows (the baseline architecture considers 8 bits fixed-point representations). After fine tuning the network, the algorithm outputs the kernels.

Each sparse kernel includes the weights and the indexes of the positions of the weights inside the kernel. Each index has four bits and represents offsets relative to the previous index. The kernel is organized as illustrated in Figure 5.

The number of indexes depends on the size of the pruning blocks as $\frac{Number of weights}{blocksize} \times pruning$. For example, a kernel with 8192 weights pruned by 90% with blocks of eight weights requires $\frac{8192}{8} \times (1 - 0.9)$ indexes. The smaller the blocks and the pruning percentage the more indexes are needed.

6 Proposed Architecture with Support for Block Pruning

The baseline architecture was extended to support block pruned dense layers. The architecture is statically configurable with a block size, that can be 1, 2, 4 or 8. The convolutional module is the same as the baseline architecture, but the fully connected module had to be modified (see Figure 6).

The data width of the weight memories is 64 bits, permitting to read eight 8-bit weights in parallel. The new architecture has a module to store the indexes and to generate the read addresses of the batch memories. Figure 7 illustrates the architecture for a bock size of eight.

Sixteen indexes are read from the 64-bit DMA word followed by sixteen blocks of eight weights each (16×64). The sixteen indexes are serialized and written in a FIFO. Each different kernel has an associated FIFO. During the computation of the inner product, the indexes are read and added sequentially to the initial address of the batch memory to determine the read address of a block of eight activations. The batch memory is a dual port memory with one port for each core in a line (each core in a line runs a different kernel). All batch memories receive the same pair of addresses. To calculate more kernels in parallel the architecture illustrated in the figure is replicated, including

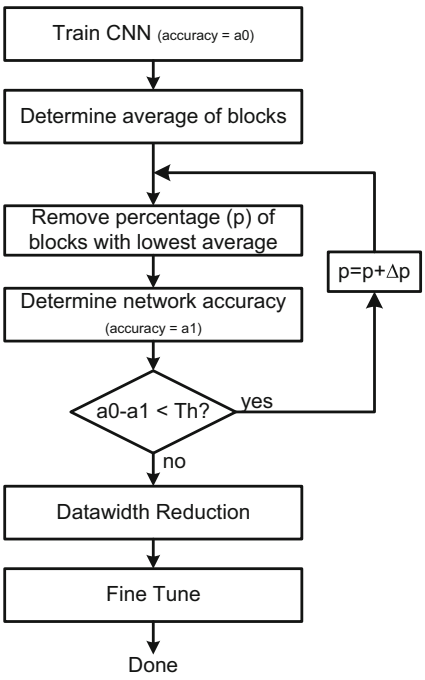

Figure 4 Network pruning flow.

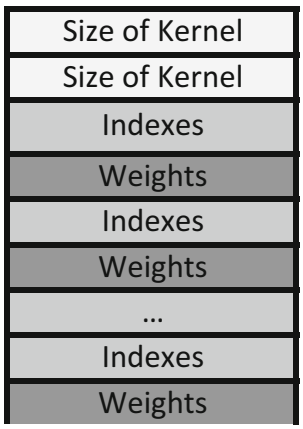

Figure 5 Organization of the kernel in memory.

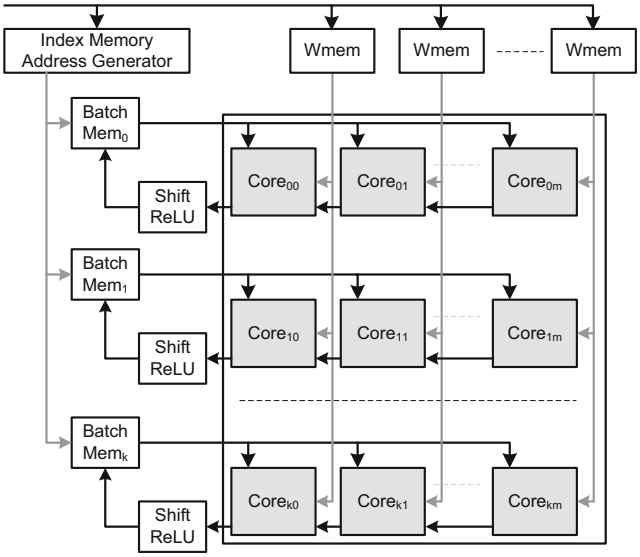

Figure 6 Architecture of the fully connected module with support for pruned weights in dense layers.

the batch memories with copied contents, to provide extra read ports.

The internal products received from the cores go through the fixed-point scale adjustment, the activation function and than concatenated into a 64-bit word to be stored in the batch memory.

The architecture must be adjusted for other pruning block sizes as follows:

1. Block size = 4: Two indexes are stored in parallel in the FIFO since two blocks will be read in parallel from the weight memory in a total of eight weights. Two read addresses are generated from the two parallel indexes, one for each block of the same kernel. Therefore, each dual-port batch memory provides the two blocks of four activations for the same core in a line. In this case, each core in a line requires one dual-port memory;
2. Block size = 2: Four indexes are stored in parallel in the FIFO since four blocks will be read in parallel from the weight memory in a total of eight weights. Four read addresses are generated from the four parallel indexes, one for each block of the same kernel. Therefore, the dual-port batch memory must be replicated to provide four ports of the same batch memory map. Therefore, each core in a line requires two dual-port memories;
3. Block size = 1: This is normal pruning. Eight indexes are stored in parallel in the FIFO since eight blocks (one weight each) will be read in parallel from the weight memory in a total of eight weights. Eight read addresses are generated from the eight parallel indexes, one for each block of the same kernel. Therefore, the dual-port batch memory must be replicated to provide eight ports of the same batch memory map. Therefore, each core in a line requires four dual-port memories.

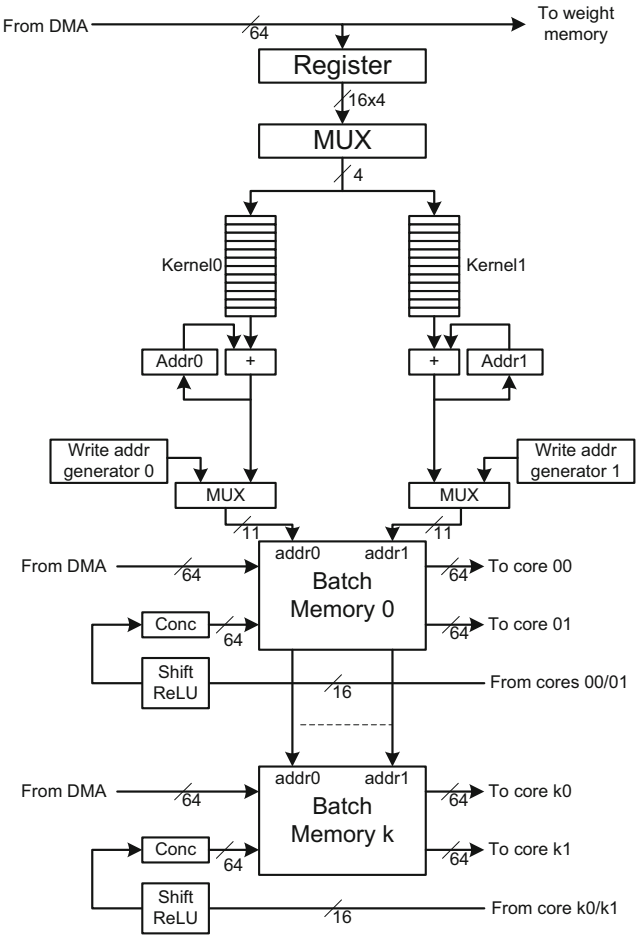

Figure 7 Architecture of the index memory and address generator for a block size fo eight.

Figure 8 illustrates the implementation for a block size of four.

The configuration requires more on-chip memory and more logic to generate multiple addresses in parallel. The number of possible indexes also doubles since the size of the block is half the size.

This solution reduces the volume of data that has to be transferred from external memory and the number of multiply accumulations is the same as the number of weights. The architecture supports unbalanced number of weights in each kernel, since all cores wait for the end of execution of all cores.

7 Results

The new architecture with the proposed pruning method was implemented with Vivado 2019.1 and tested in a ZedBoard

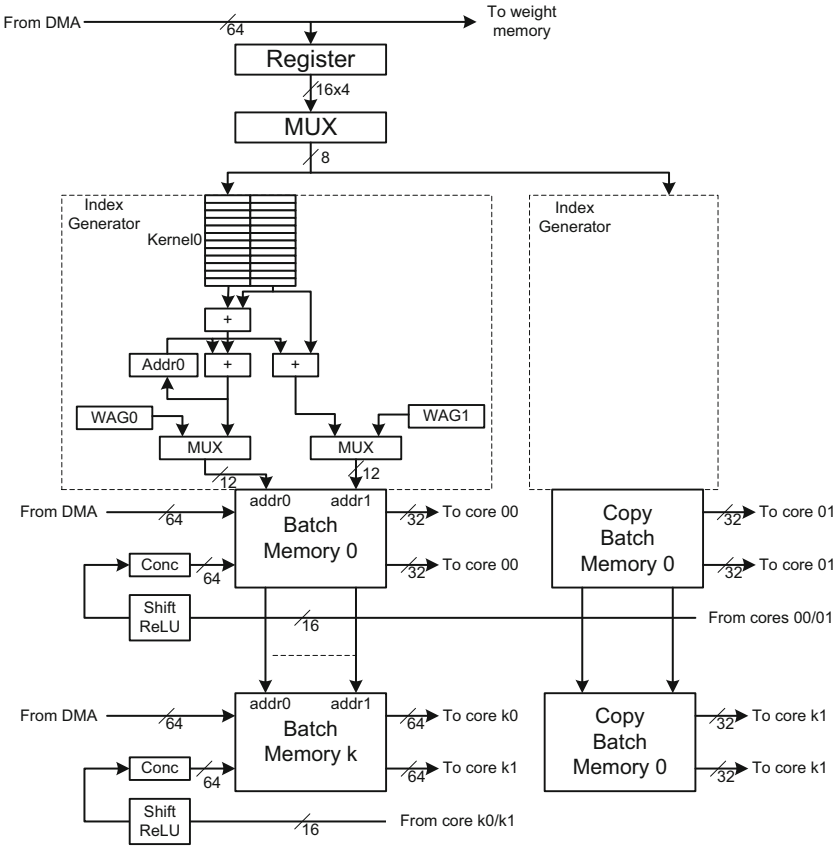

Figure 8 Architecture of the index memory and address generator for a block size of four.

with a small density FPGA - ZYNQ XC7Z7020 (Artix-7 FPGA with a dual ARM Cortex-A9 CPU). The circuit operates at 200 MHz. To demonstrate the scalability of the proposed architecture, it was also mapped in a XC7Z045 FPGA (Kintex-7 FPGA with a dual ARM Cortex-A9 CPU) with an operating frequency of 230 MHz. ZYNQ FPGAs have four 64-bit High-Performance (HP) ports working at working at 150 MHz that gives programmable logic direct access to external memory. The measured external memory bandwidth is 3.3 GBytes/s. All architectures were tested with AlexNet.

The proposed architecture was configured and implemented with block pruning with sizes of 8, 4, 2 and 1. The accuracy of the network for different pruning percentages with different block sizes and 8 bit fixed-point quantization was determined (see Figure 9, where Bx is the configuration with block size x).

From the results, we observe that the size of the pruning block influences the accuracy of the network, but with small variations. Considering a pruning of 90% and a block of size 8, the accuracy difference to a non-pruned network is about 1.8. Similar results were obtained with 16 bit quantization since the accuracy difference between 8 bit and 16 bit quantization is small (around 1.5). Also, considering a pruning of 90 %, the accuracy difference from a block of size one to a block of size 8 is 1.3.

We have synthesized and implemented the best architectures for different pruning and block sizes with the following configurations. We designate the architecture as Arq_B_P, where B indicates the block size and P indicates the pruning percentage. We considered block sizes of 1, 2, 4, 8 and pruning of 90%, 70% and 0%. All architectures were mapped in both FPGAs, the ZYNQ7020 (see Tables 1 and 2) and the ZYNQ045 (see Tables 3 and 4).

The table includes the number of cores in each processing module: convolutional module and fully connected module. With a high pruning of 90%, a batch of two is still required, but the intra-parallelism of the fully connected module is only one because the bandwidth available for the fully connected module is only able to feed one core. The

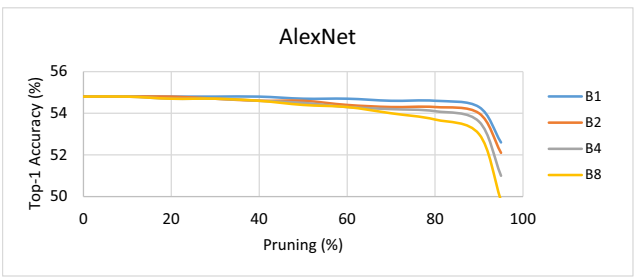

Figure 9 Variation of accuracy with pruning percentage and block size.

Table 1 Results of the architecture for different block sizes and a pruning of 90% and the baseline architecture in a ZYNQ7020 SoC FPGA.

ZYNQ7020					
	Arq_8_90	Arq_4_90	Arq_2_90	Arq_1_90	Arq_1_0
Conv cores	16 × 7	16 × 7	16 × 7	16 × 7	16 × 7
FC cores	1 × 2	1 × 2	1 × 2	1 × 2	2 × 6
Batch	2	2	2	2	6
LUT	43323	43393	43473	43573	46130
DSP	220	220	220	220	220
BRAM(36 Kbits)	110	110	122	130	134
Equivalent GOPs	347	347	347	347	294
Images/s	240	240	240	240	203

architecture achieves the same performance for all sizes of the pruning blocks. The only difference is in the required resources. As we move to a lower block size, the number of LUTs increases slightly and the number of BRAMs also increases. This is a major point to take into consideration when the architecture is mapped on small density FPGAs where the on-chip memory resources is scarce. Compared to the original baseline architecture, the pruned architecture improves the performance 18%. The proposed architecture with a block size of one (only 0.5% percent reduction in accuracy) is 6% smaller in area and 18% faster. We also observe that since the intra-parallelism is only one, the results with a block size of four are identical to those with a block size of eight. If the available memory for the fully connected module was enough to allow a higher intra-parallelism, the implementation with a block size of eight would be smaller. So, the advantage of a higher block size is only evident with a higher memory bandwidth for dense layers.

When the pruning is decreased to 70%, the batch must increase to three to compensate the increase of the number of fully connected weights. The intra-parallelism of the fully connected module is kept at one. The architecture achieves the same performance for the first three sizes of the pruning blocks with an increase in the required resources. However, for a block size of one, the FPGA does not have enough BRAMs to implement the batch size of three. The batch was reduced to two with a redution in performance and still requiring the highest number of BRAMs. Compared to the baseline architecture, the 70% pruned architecture improves the performance by 17%, a slight reduction because of the reduction in pruning.

Considering the larger ZYNQ7045 FPGA, with a pruning of 90%, a batch of sixteen is required, but the intra-parallelism of the fully connected module is still one because of the limited bandwidth of the ZYNQ. The architecture achieves the same performance for all sizes of the pruning blocks with a slight reduction for a unitary block size. The number of LUTs increases slightly with the reduction of the block size. The main increase in resources is the number of BRAMs. Since the batch is high, the architecture with BS=1 needs 73% more BRAMs than the architecture with BS=8. Compared to the original baseline architecture, the pruned architecture improves the performance by 37%, but the performance of the baseline architecture was achieved with a batch size of 32.

With a pruning of 70%, the architectures are identical to those used with a pruning of 90%. The increase in the number of weights is compensated by the data dispatcher that gives more bandwidth to the fully connected module.

Table 2 Results of the architecture for different block sizes and a pruning of 70% and the baseline architecture in a ZYNQ7020 SoC FPGA.

ZYNQ7020					
	Arq_8_70	Arq_4_70	Arq_2_70	Arq_1_70	Arq_1_0
Conv cores	16 × 7	16 × 7	16 × 7	16 × 7	16 × 7
FC cores	1 × 2	1 × 2	1 × 27	1 × 2	2 × 6
Batch	3	3	3	2	6
LUT	43901	43971	44051	43573	46130
DSP	220	220	220	220	220
BRAM(36 Kbits)	118	118	130	138	134
Equivalent GOPs	344	344	344	275	294
Images/s	238	238	238	190	203

Table 3 Results of the architecture for different block sizes and a pruning of 90% and the baseline architecture in a ZYNQ7045 SoC FPGA.

ZYNQ7045					
	Arq_8_90	Arq_4_90	Arq_2_90	Arq_1_90	Arq_1_0
Conv cores	64 × 7	64 × 7	64 × 7	64 × 7	64 × 7
FC cores	1 × 16	1 × 16	1 × 167	1 × 16	1 × 32
Batch	16	16	16	16	32
LUT	170279	170349	170429	170529	181135
DSP	728	728	728	728	748
BRAM(36 Kbits)	262	262	326	454	388
Equivalent GOPs	1123	1123	1123	1114	819
Images/s	775	775	775	769	565

The execution time of both modules increases, with a consequent reduction of the performance. Compared to the original baseline architecture, the pruned architecture improves the performance by 19%, a slight reduction because of the reduction in pruning.

To analyze the importance of batch versus pruning, two architectures with BS = 4 and different small batch (1, 2 and 3) were executed. This establishes the importance of each method and, in particular, of pruning when image batch is not allowed (see results in Figure 10)

Without batch (batch = 1), pruning is fundamental to achieve a performance up to 4× better than an architecture without pruning. Batch introduces an extra improvement over pruning with no accuracy degradation. The influence of batch is higher when pruning is lower. For example, with the highest pruning, the batch has only a slight influence.

It is interesting to observe that the same image throughput can be achieved with different combinations of pruning and batch. For example, the performance of the architecture with batch=1 and pruning=60% is the same of the architecture with batch=2 and prune=20%. The difference is that the last architecture is more accurate since the pruning is lower. Therefore, the best architecture depends on the constraints in terms of batch, accuracy and available resources.

The same experiment was done with the ZYNQ7045 FPGA (see results in F igure 11)

The main difference is that the batch method now has a bigger effect over the performance of the architecture. This has to do with the fact that the memory bandwidth is the same for an architecture with more computational resources. In this case, the computation to communication ratio decreases and the communication bottleneck is more evident. The batch method is important to balance both communication and computation.

We have compared configuration B4 with 8 bit quantization, 90 % pruning and batch with previous works running AlexNet. The overall results are shown in Table 5.

There are only a few works implemented in a small ZYNQ7020 using 8-bit quantization. Compared to [43], the proposed architecture is 4.1× faster and more efficient (GOPs/kLUT and GOPs/DSP) with just 0.3% difference in accuracy. Compared to the solutions with 16-bit data, the improvement is 4× better than the pruned architecture [44] and 12.5× better than [44].

When mapped to a ZYNQ7045, the proposed architecture is better than the previous works with a slight reduction in accuracy. The efficiency of the proposed architecture is also better than the state of the art proposals.

Table 4 Results of the architecture for different block sizes and a pruning of 70% and the baseline architecture in a ZYNQ7045 SoC FPGA.

ZYNQ7045					
	Arq_8_90	Arq_4_90	Arq_2_90	Arq_1_90	Arq_1_0
Conv cores	64 × 7	64 × 7	64 × 7	64 × 7	64 × 7
FC cores	1 × 16	1 × 16	1 × 16	1 × 16	1 × 32
Batch	16	16	16	16	32
LUT	170279	170349	170429	170529	181135
DSP	728	728	728	728	748
BRAM(36 Kbits)	262	262	326	454	388
Equivalent GOPs	979	972	959	929	819
Images/s	676	671	662	641	565

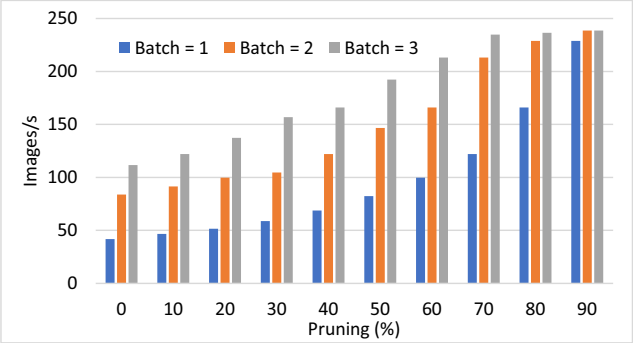

Figure 10 Variation of performance with pruning percentage, considering BS=4 and two different batch sizes, 1 and 2 in a ZYNQ 7020.

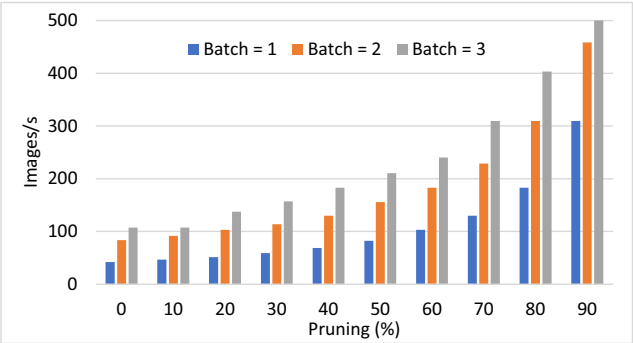

Figure 11 Variation of performance with pruning percentage, considering BS=4 and two different batch sizes, 1 and 2 in a ZYNQ 7045.

8 Conclusions

In this work a block pruning technique and an extended architecture to support pruned networks were proposed. The extended architecture with configurable pruning datapath proposed in this work permits to improve the performance/area efficiency with minimal accuracy reduction. This is fundamental for embedded systems with low resources. The required on-chip memory to support weight pruning depends on the block size of the pruning technique. An integrated design of both pruning and image batch leads to different architectural solutions with different area and different network accuracy.

The results show that the proposed accelerator of convolutional neural networks in low density FPGAs achieves very good accuracy with high image processing throughput.

In the future, we plan to integrate more optimization techniques oriented for best hardware datapath design.

Acknowledgments This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UIDB/50021/2020 and was also supported by project IPL/IDI&CA/2020/TRAINEE/ISEL through Instituto Politécnico de Lisboaa.

Compliance with Ethical Standards

Conflict of interests The authors declare that they have no conflict of interest.

Table 5 Performance comparison of Lite-CNN with other works in low density ZYNQ7020 and ZYNQ7045 SoC FPGAs.

ZYNQ 7020							
Work	Format	Freq (MHz)	GOPs	GOPs/kLUT	GOPs/DSP	Latency (ms)	Acc.
[49]	16 × 16	100	19	0.35	0.08	71.75	a)
[50]	16 × 16	150	20	0.38	0.09	—	a)
[42]	16 × 16	125	38	0.73	0.17	52.4	a)
[44]	16 × 16	200	80	1.5	0.36	16, 7 b)	a)
[43]	8 × 8	214	84	1.6	0.38	17.2	53.9%
This work	8 × 8	200	322	7.4	1.46	4.17	53.6%
ZYNQ 7045							
Work	Format	Freq (MHz)	GOPs	GOPs/kLUT	GOPs/DSP	Latency (ms)	Acc.
[28]	8 × 8	200	493	5.7	0.6	2.94	54.6%
[8]	8 × 8	200	133	2.9	0.6	10.9	—
[51]	8 × 8	30	290 c)	2.8	0.4	4.9	55.9%
This work	8 × 8	230	1123	6.6	1.5	1.3	54 %
This work	8 × 8	230	972	5.7	1.3	1.5	54.3 %

a) Authors assume accuracy close to that obtained with floating-point - 55,9%

b) With pruning and image batch

c) Convolution layers only in ZYNQ Ultrascale+ XCZU7EV

References

1. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., Fei-Fei, L. (2015). Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, *115*(3), 211–252. https://doi.org/10.1007/s11263-015-0816-y.
2. Cun, Y.L., Jackel, L.D., Boser, B., Denker, J.S., Graf, H.P., Guyon, I., Henderson, D., Howard, R.E., Hubbard, W. (1989). Handwritten digit recognition: applications of neural network chips and automatic learning. *IEEE Communications Magazine*, *27*(11), 41–46. https://doi.org/10.1109/35.41400.
3. Krizhevsky, A., Sutskever, I., Hinton, G.E. (2012). Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (pp. 1097–1105). USA: NIPS'12, Curran Associates Inc.
4. Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 3rd International Conference on Learning Representations*.
5. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A. (2015). Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 1–9).
6. He, K., Zhang, X., Ren, S., Sun, J. (2016). Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 770–778).*
7. Véstias, M. (2020). Deep learning on edge: Challenges and trends. In Rodrigues, J.M., Cardoso, P.J., Monteiro, J., Ramos, C.M. (Eds.) *Smart Systems Design, Applications, and Challenges* (pp. 23–42): IGI Global.
8. Véstias, M.P., Duarte, R.P., deSousa, J.T., Neto, H. (2018). Lite-cnn: A high-performance architecture to execute cnns in low density fpgas. In *Proceedings of the 28th International Conference on Field Programmable Logic and Applications*.
9. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. arXiv:1408.5093.
10. Gysel, P., Pimentel, J., Motamedi, M., Ghiasi, S. (2018). Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks. *IEEE Transactions on Neural Networks and Learning Systems*. https://doi.org/10.1109/TNNLS.2018.2808319.
11. Véstias, M. (2020). Processing systems for deep learning inference on edge devices. In Mastorakis, G., Mavromoustakis, C.X., Batalla, J.M., Pallis, E. (Eds.) *Convergence of Artificial Intelligence and the Internet of Things* (pp. 213–240). Cham: Springer International Publishing.
12. Google: Edge TPU. (2019) https://cloud.google.com/edge-tpu/.
13. Coral: EDGE TPU Performance Benchmarks. (2020) https://coral.ai/docs/edgetpu/benchmarks.
14. Mário, V., Lopes, J.D., Véstias, M., deSousa, J.T. (2020). Implementing cnns using a linear array of full mesh cgras. In Rincón, F., Barba, J., So, H.K.H., Diniz, P., Caba, J. (Eds.) *Applied Reconfigurable Computing. Architectures, Tools, and Applications* (pp. 288–297). Cham: Springer International Publishing.
15. Chakradhar, S., Sankaradas, M., Jakkula, V., Cadambi, S. (June 2010). A dynamically configurable coprocessor for convolutional neural networks. *SIGARCH Comput. Archit. News*, *38*(3), 247–257. https://doi.org/10.1145/1816038.1815993.
16. Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., Temam, O. (2014). Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture* (pp. 609–622).
17. Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., Cong, J. (2015). Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '15* (pp. 161–170). New York: ACM.
18. Liu, B., Zou, D., Feng, L., Feng, S., Fu, P., Li, J. (2019). An fpga-based cnn accelerator integrating depthwise separable convolution. *Electronics*, *8*(3), 18.
19. Rivera-Acosta, M., Ortega-Cisneros, S., Rivera, J. (2019). Automatic tool for fast generation of custom convolutional neural networks accelerators for fpga. *Electronics*, *8*(6), 17.
20. Qiu, J., Wang, J., Yao, S., Guo, K., Li, B., Zhou, E., Yu, J., Tang, T., Xu, N., Song, S., Wang, Y., Yang, H. (2016). Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '16* (pp. 26–35). New York: ACM.
21. Suda, N., Chandra, V., Dasika, G., Mohanty, A., Ma, Y., Vrudhula, S., Seo, J.S., Cao, Y. (2016). Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '16* (pp. 16–25). New York: ACM.
22. Qiao, Y., Shen, J., Xiao, T., Yang, Q., Wen, M., Zhang, C. (2017). Fpga-accelerated deep convolutional neural networks for high throughput and energy efficiency. *Concurrency and Computation: Practice and Experience*, *29*(20), e3850–n/a. https://doi.org/10.1002/cpe.3850,cpe.3850.
23. Liu, Z., Dou, Y., Jiang, J., Xu, J., Li, S., Zhou, Y., Xu, Y. (July 2017). Throughput-optimized fpga accelerator for deep convolutional neural networks. *ACM Trans. Reconfigurable Technol. Syst.*, *10*(3), 17:1–17:23. https://doi.org/10.1145/3079758.
24. Alwani, M., Chen, H., Ferdman, M., Milder, P. (2016). Fused-layer cnn accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (pp. 1–12).
25. Shen, Y., Ferdman, M., Milder, P. (2017). Maximizing cnn accelerator efficiency through resource partitioning. *SIGARCH Comput. Archit. News*, *45*(2), 535–547. https://doi.org/10.1145/3140659.3080221.
26. Gonçalves, A., Peres, T., Véstias, M. (2019). Exploring data bitwidth to run convolutional neural networks in low density fpgas. In Hochberger, C., Nelson, B., Koch, A., Woods, R., Diniz, P. (Eds.) *Applied Reconfigurable Computing* (pp. 387–401). Cham: Springer International Publishing.
27. Gysel, P., Motamedi, M., Ghiasi, S. (2016). Hardware-oriented approximation of convolutional neural networks. In *Proceedings of the 4th International Conference on Learning Representations*.
28. Wang, J., Lou, Q., Zhang, X., Zhu, C., Lin, Y., Chen, D. (2018). A design flow of accelerating hybrid extremely low bit-width neural network in embedded fpga. In *28th International Conference on Field-Programmable Logic and Applications*.
29. Véstias, M.P., Duarte, R.P., De Sousa, J.T., Neto, H.C. (2020). A configurable architecture for running hybrid convolutional neural networks in low-density fpgas. *IEEE Access*, *8*, 107229–107243.
30. Umuroglu, Y., Fraser, N.J., Gambardella, G., Blott, M., Leong, P., Jahre, M., Vissers, K. (2017). Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '17* (pp. 65–74). New York: ACM. https://doi.org/10.1145/3020078.3021744.
31. Han, S., Mao, H., Dally, W.J. (2015). Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. CoRR, arXiv:1510.00149.

32. Yu, J., Lukefahr, A., Palframan, D., Dasika, G., Das, R., Mahlke, S. (June 2017). Scalpel: Customizing dnn pruning to the underlying hardware parallelism. *SIGARCH Comput. Archit. News*, *45*(2), 548–560. https://doi.org/10.1145/3140659.3080215.
33. Albericio, J., Judd, P., Hetherington, T., Aamodt, T., Jerger, N.E., Moshovos, A. (2016). Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)* (pp. 1–13).
34. Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M.A., Dally, W.J. (2016). Eie: Efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)* (pp. 243–254).
35. Parashar, A., Rhu, M., Mukkara, A., Puglielli, A., Venkatesan, R., Khailany, B., Emer, J., Keckler, S.W., Dally, W.J. (June 2017). Scnn: An accelerator for compressed-sparse convolutional neural networks. *SIGARCH Comput. Archit. News*, *45*(2), 27–40. https://doi.org/10.1145/3140659.3080254.
36. Nurvitadhi, E., Venkatesh, G., Sim, J., Marr, D., Huang, R., Ong GeeHock, J., Liew, Y.T., Srivatsan, K., Moss, D., Subhaschandra, S., Boudoukh, G. (2017). Can fpgas beat gpus in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '17* (pp. 5–14). New York: ACM. https://doi.org/10.1145/3020078.3021740.
37. Aimar, A., Mostafa, H., Calabrese, E., Rios-Navarro, A., Tapiador-Morales, R., Lungu, I., Milde, M.B., Corradi, F., Linares-Barranco, A., Liu, S., Delbruck, T. (2019). Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps. *IEEE Transactions on Neural Networks and Learning Systems*, *30*(3), 644–656. https://doi.org/10.1109/TNNLS.2018.2852335.
38. Zhang, S., Du, Z., Zhang, L., Lan, H., Liu, S., Li, L., Guo, Q., Chen, T., Chen, Y. (2016). Cambricon-x: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (pp. 1–12).
39. Lu, L., Xie, J., Huang, R., Zhang, J., Lin, W., Liang, Y. (2019). An efficient hardware accelerator for sparse convolutional neural networks on fpgas. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (pp 17–25).
40. Véstias, M.P., Duarte, R.P., deSousa, J.T., Neto, H.C. (2019). Fast convolutional neural networks in low density fpgas using zero-skipping and weight pruning. *Electronics* (8), 11. https://doi.org/10.3390/electronics8111321.
41. Véstias, M., Duarte, R., Sousa, J.T.D., Neto, H. (2020). Moving deep learning to the edge. *Algorithms*, *13*, 125.
42. Venieris, S.I., & Bouganis, C. (2018). fpgaconvnet: Mapping regular and irregular convolutional neural networks on fpgas. *IEEE Transactions on Neural Networks and Learning Systems*, 1–17. https://doi.org/10.1109/TNNLS.2018.2844093.
43. Guo, K., Sui, L., Qiu, J., Yu, J., Wang, J., Yao, S., Han, S., Wang, Y., Yang, H. (2018). Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *37*(1), 35–47. https://doi.org/10.1109/TCAD.2017.2705069.
44. Gong, L., Wang, C., Li, X., Chen, H., Zhou, X. (2018). Maloc: A fully pipelined fpga accelerator for convolutional neural networks with all layers mapped on chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *37*(11), 2601–2612. https://doi.org/10.1109/TCAD.2018.2857078.
45. Véstias, M.P., Duarte, R.P., de Sousa, J.T., Neto, H.C. (2020). A fast and scalable architecture to run convolutional neural networks in low density fpgas. *Microprocessors and Microsystems*, *77*, 103136.
46. Peres, T., Gonçalves, A., Véstias, M. (2019). Faster convolutional neural networks in low density fpgas using block pruning. In Hochberger, C., Nelson, B., Koch, A., Woods, R., Diniz, P. (Eds.) *Applied Reconfigurable Computing* (pp. 402–416). Cham: Springer International Publishing.
47. Struharik, R.J.R., Vukobratović, B.Z., Erdeljan, A.M., Rakanović, D.M. (2020). Conna-hardware accelerator for compressed convolutional neural networks. *Microprocessors and Microsystems*, *73*, 102991.
48. Véstias, M. (2021). Convolutional neural network. In Khosrow-Pour, D.B.A.M. (Ed.) *Encyclopedia of Information Science and Technology, Fifth Edition* (pp. 12–26): IGI Global.
49. Wang, Y., Xu, J., Han, Y., Li, H., Li, X. (2016). Deepburning: Automatic generation of fpga-based learning accelerators for the neural network family. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)* (pp. 1–6).
50. Sharma, H., Park, J., Mahajan, D., Amaro, E., Kim, J.K., Shao, C., Mishra, A., Esmaeilzadeh, H. (2016). From high-level deep neural models to fpgas. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (pp. 1–12).
51. Zhang, M., Li, L., Wang, H., Liu, Y., Qin, H., Zhao, W. (2019). Optimized compression for implementing convolutional neural networks on fpga. *Electronics*, *8*(3), 295. https://doi.org/10.3390/electronics8030295.

Mário P. Véstias is a Coordinate Professor at the Polytechnic Institute of Lisbon, School of Engineering (ISEL), Department of Electronic, Telecommunications and Computer Engineering (DEETC). He is a senior researcher at the Electronic Systems Design and Automation group at the research institute INESC-ID in Lisbon. His main research interests are Computer Architectures and Digital Systems for High-Performance Embedded Computing, with an emphasis on Reconfigurable Computing. He is a PhD in Electrical and Computer Engineering from the Technical University of Lisbon.