Microprocessors and Microsystems 77 (2020) 103136

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

A fast and scalable architecture to run convolutional neural networks in low density FPGAs

Mário P. Véstias[a,*], Rui P. Duarte[b], José T. de Sousa[b], Horácio C. Neto[b]

[a] INESC-ID, ISEL, Instituto Politécnico de Lisboa, Portugal
[b] INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal

ARTICLE INFO

Article history:
Received 15 February 2019
Revised 8 March 2020
Accepted 14 May 2020
Available online 21 May 2020

Keywords:
Deep learning
Convolutional neural network
Smart edge devices
FPGA

ABSTRACT

Deep learning and, in particular, convolutional neural networks (CNN) achieve very good results on several computer vision applications like security and surveillance, where image and video analysis are required. These networks are quite demanding in terms of computation and memory and therefore are usually implemented in high-performance computing platforms or devices. Running CNNs in embedded platforms or devices with low computational and memory resources requires a careful optimization of system architectures and algorithms to obtain very efficient designs. In this context, Field Programmable Gate Arrays (FPGA) can achieve this efficiency since the programmable hardware fabric can be tailored for each specific network. In this paper, a very efficient configurable architecture for CNN inference targeting any density FPGAs is described. The architecture considers fixed-point arithmetic and image batch to reduce computational, memory and memory bandwidth requirements without compromising network accuracy. The developed architecture supports the execution of large CNNs in any FPGA devices including those with small on-chip memory size and logic resources. With the proposed architecture, it is possible to infer an image in AlexNet in 4.3 ms in a ZYNQ7020 and 1.2 ms in a ZYNQ7045.

1. Introduction

Recently, the convolutional neural network has emerged as a method for many artificial intelligence tasks including image classification [1], useful for computer vision applications, like object detection and image segmentation. When used in edge devices, these methods and algorithms turn embedded devices into smart embedded devices that can take decisions based on the smart analysis of collected data.

Deep neural networks (DNN) have already achieved the accuracy of humans. For this reason, they are being applied in many automatic classification tasks. Convolutional neural network (CNN) a type of DNN is very effective on these tasks since it can identify correlations among data inputs and mix them to classify images. Similar to the human brain, CNNs are made of a series of layers of connected neurons with associated weights. This network of layers is trained to classify unknown data not used during the training process.

What distinguishes CNNs from other DNNs is the fact that the first hidden layers known as convolutional execute 3D convolutions between groups of weights (3D kernels) and the initial image or input maps. Each convolution with a 3D kernel produces an output feature map that is the input map of the next layer. Many kernels are used at each layer producing as many output feature maps. This large number of convolutions permits to discover features of the image that are then correlated to identify classes of objects. The final layers of a CNN are the fully connected (FC), where all nodes of a layer are connected to all nodes of the previous layer. The last fully connected layer outputs the result of the inference with each node corresponding to a class probability.

CNNs may have any number of layers and kernels producing maps with different sizes. This diversity has led to diverse networks during the last years applied to different problems and achieving increasing classification accuracy. The first CNNs were regular networks based on the main two layers described. LeNet [2] was one of the first CNNs used for digit classification represented with images of size 32×32. It has two convolutional layers and three fully connected layers with a total of 60K weights. A larger CNN, AlexNet [3], was presented in the ImageNet Challenge for image classification. It consists of five convolutional layers followed by three fully connected layers with a total of 61M weights requiring a total of 724 MAC (Multiply-ACcumulate) operations to

* Corresponding author.
E-mail addresses: mvestias@deetc.isel.pt (M.P. Véstias), rui.duarte@tecnico.ulisboa.pt (R.P. Duarte), jose.desousa@inesc-id.pt (J.T. de Sousa), hcn@inesc-id.pt (H.C. Neto).

https://doi.org/10.1016/j.micpro.2020.103136
0141-9331/© 2020 Elsevier B.V. All rights reserved.

process images of size 224 × 224 × 3 with a top-5 error rate for ImageNet around 20%. Other well-known regular CNN models have followed including the VGG-16 [4] with 16 layers, 2.2 × more weights than AlexNet and 15.5 GMAC operations with a top-5 error rate around 10%. GoogleNet [5] an irregular CNN with 57 convolutional layers and a new type of layer (the inception module that consists of parallel convolutions) needs seven million weights with a total workload of 1.58 GOPs (Giga Operations) to achieve a top-5 error of 7%. Several versions of ResNet [6] with a number of convolutional layers ranging from 53 to 155 with up to 11.3 GOPs of workload reduced the top-5 error to between 5.8% and 6.7%.

The first CNN exceeding human level accuracy was ResNet [6], that won the ImageNet Challenge in 2016. ResNet introduced a new module that contains an identity connection to reduce the complexity of the training and, like GoogleNet, also uses 1 × 1 convolutions.

Several other CNNs were proposed in the last years, some regular and some irregular with layers different from the usual convolutional and fully connected layers. Implementing any of these networks in FPGA with the best performance is a difficult design task, even more challenging for low density FPGAs with strict performance and memory constraints. Therefore, it is important to have parameterizable architectures that can be optimized for each particular CNN without compromising the performance.

In this paper, a parameterizable architecture for inference of regular CNNs in FPGA is proposed. It deals efficiently with the diversity of layers and kernels and can be implemented in low density FPGAs to run large CNNs in low cost FPGA-based embedded systems. The proposed architecture considers a set of architectural optimizations: fixed-point quantization, that is, activations and weights are represented with fixed-point format (fixed-point representation in different layers can have different scale factors); an efficient method to calculate the convolutional layers that is independent of the size of the convolution window; image batch where convolutional layers are run for multiple images before running the fully connected layers; and parallel execution of convolutional and fully connected layers to improve the implementation of each type of layer and increase throughput.

The paper is organized as follows. Section 2 describes the related work on FPGA implementations of CNNs and optimization methods. Section 3 describes the fundamentals of convolutional neural networks. Section 4 describes the proposed architecture for CNN inference. Section 5 describes area and performance models of the architecture to help designing the architecture for best performance. Section 6 describes the area and the performance results of the architecture running well-known CNNs. Section 7 concludes the paper.

2. Related work

FPGAs are promising platforms for the acceleration of CNN inference because they have higher energy efficiency when compared to GPU (Graphics Processing Units) and CPU (Central Processing Units) and higher performance when compared to CPU. The reconfigurability of FPGAs is also an advantage because it permits to quickly adapt the hardware architecture to the specific needs of the CNN. Using operands and operations with the strictly required size improves the resource utilization of the architectures. While having all these advantages, FPGAs require hardware design and implementation expertise. To overcome this, a few works have proposed automatic frameworks and high-level synthesis (HLS) to automatically convert a high-level specification of the network into a synthesizable architecture to be implemented in FPGA [7]. While HLS tools allow fast design of architectures for CNN inference, they are not optimized for best performance and most efficient resource usage. A different approach is to consider configurable cores or architectural templates that can be configured for a particular network [8]. This is the approach followed in this work.

Depending on the target FPGA device, hundreds or thousands of GFLOPs were already obtained in the execution of CNN inference. FPGA implementations of CNNs started to consider small networks [9,10]. The first approaches to the implementation of CNNs in FPGA considered only convolutional layers [11]. In [12] and [13] FPGA implementations of complete CNN models were proposed. The former uses an architecture similar to that proposed in [11]. These works consider a flexible architecture that can run any convolutional layer with different shapes and sizes of convolution windows. The problem of this architecture is that the performance efficiency varies with the window sizes of convolutions. To eliminate this performance variability with the window size, Suda et al. [13] implement convolutions as matrix multiplications by rearranging the input maps of the layer. However, the solution introduces a large overhead associated with the memory accesses and execution times necessary to rearrange the input maps. This overhead was partially eliminated in [14] using an accelerator for matrix multiplication and dedicated units to convert the inputs maps into a matrix.

Reusing a hardware module for different layers improves resource efficiency, but may lead to low throughput if not carefully designed since different layers require different computing patterns and the number of parallel computing cores may not match the number of operations. Another mismatch aspect is associated with the utilization of external memory bandwidth. The distribution of memory bandwidth throughout the computing resources must be carefully designed so that there is a correct balance between computation and communication.

A different direction was followed by Liu et al. [15] that instead of a flexible structure to run any layer proposed a pipeline architecture with a layer at each pipeline level. The work achieves 445 GOPs (fixed-point 8-16b) in a Virtex7 VX690T. The approach permits the optimization of the architecture for each layer but requires other techniques like fused layers [16] to account for the extra memory required to store intermediate maps and weights. The solution reduces on-chip memory requirements but still requires some memory that increases with the number of layers which may not be available in low density FPGAs. A trade-off between a single HW module for all layers and a pipeline structure with a module for each layer was proposed in [17] where different subsets of convolutional layers are mapped to different computing layer modules. The solution trade-offs resource efficiency by performance.

It is known that CNNs have high redundancy permitting a significant simplification. The simplifications have been used by several authors looking for hardware simplification. Data quantization includes a set of techniques that reduce the size and arithmetic type of activations and weights. In [18] the author has shown that fixed-point data representations with 8 bits guarantee an accuracy close to that obtained with 32-bit floating point. The size of data can be fixed for all layers or optimized for each layer [19]. In extreme implementations, CNNs are converted to BNN (Binary Neural Networks) where weights or both activations and weights are represented with a single bit reducing memory requirements [20, 21]. The limitation of BNNs is that to obtain an accuracy comparable to that obtained with a floating-point based architecture it needs from 2 to 11 × more weights and operations [20]. Also, the first and last layers require full precision and so the architecture must support both representations. Binary networks use binary weights but batch normalization parameters and bias values still need full arithmetic representations. Binarized networks with 1-bit weights have some accuracy drop that for large networks can be over 10%. This is even worst when both weights and activations are represented with a single bit. BNNs can be efficiently implemented with LUTs of the FPGA, leaving DSPs for addition only, which reduces resource utilization.

Another class of optimizations considers data reduction. A first approach to these methods was proposed in [22] where DNNs are compressed using pruning and Huffman coding. Results show that pruning the fully connected layers of AlexNet by 91% have a negligible effect over the network accuracy. In [23] the pruning is adapted to the underlying hardware matching the pruning structure to the data-parallel hardware arithmetic unit. The method is applied to CPU and GPU. Pruning is typically not applied to convolutional layers since the percentage of weights in these layers is quite below the number of weights in fully connected layers. Pruning introduces sparsity in the kernels of weights and unbalanced computation of different output feature maps. Sparsity requires irregular accesses to on-chip memory of activations. To keep dot-product parallelism, these memories are replicated.

Some optimization techniques are more hardware friendly since they allow keeping the regular structure of computing nodes. Many sources of parallelism exist in the CNN computation. There is parallelism between different output feature maps (inter-output parallelism), between different convolutions of the same layer (intra-output parallelism) and between dot-product operations (intra-kernel parallelism). All sources of parallelism can be explored in a CNN implementation [24].

Batching is used to reduce memory bandwidth requirements [25]. Several output feature maps of the last convolutional layer are batched before being executed. The process increases kernel reuse in fully connected layers since the same kernel is used for all batched maps.

Recently, authors started to consider low density FPGAs for the implementation of CNNs. In [8] small CNNs were implemented in a ZYNQ XC7Z020 with an average performance of 13 GOPs with weights represented in 16 bit fixed-point format. The same FPGA was then used to implemented bigger CNN models, like VGG16, with data represented with 8 bits [26] achieving performances of 84 GOPs. In [27] the authors implemented a pipelined architecture with weight pruning in fully connected layers in a ZYNQ XC7Z020 with data represented with 16-bit fixed point achieving 76 GOPs.

Optimization techniques are necessary to run large CNNs in low density FPGAs. Some techniques like data quantization, parallelism exploration and batching do not compromise the regularity of hardware computing modules. Others, like pruning, introduce irregular computations that reduce the performance and resource effectiveness of the architecture. Extreme quantization still degrades accuracy and fully pipelined layer modules are not appropriate for low density FPGAs.

Therefore, a parameterizable hardware module for CNN inference in FPGA that optimizes performance and resource utilization without accuracy degradation of the network model and that can be mapped on low density FPGAs is proposed. The architecture considers 8-bit data quantization, batching and two separate modules to process different types of layers (convolutional and fully connected) in a pipelined model of computation. Performance and area models are derived for the architecture to help designers choose the best configuration for a particular network.

Compared to previous CNNs, the proposed architecture improves the inference performance and the resource efficiency.

3. Convolutional neural networks

Convolutional neural networks consist of a series of processing layers of different types. Each layer receives a set of input feature maps (IFM) from the previous layer and generates Output Feature Maps (OFM) to the next layer. In regular CNN there are convolutional, fully connected and pooling layers. Some works consider other type of layers in their CNNs. For example, GoogleNet [5] has the inception layer and [6] introduced a new layer that contains an identity connection to reduce the complexity of training. CNNs with specific layers are usually known as irregular CNNs.

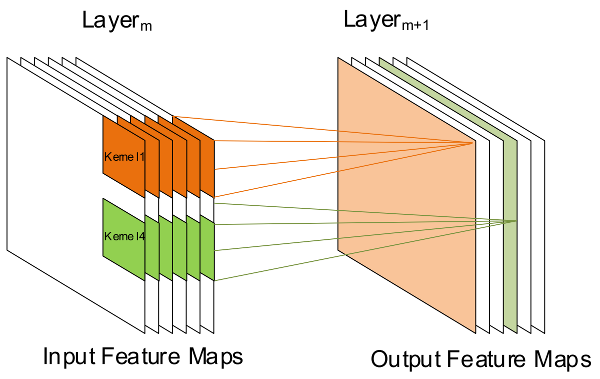

Fig. 1. Input and output feature maps of convolutional layers.

The most computational intensive part of a CNN are the convolutional layers in which a set of 3D kernels are convolved with the IFMs to generate the OFMs. In this paper nodes of feature maps are referred to as activations (see Fig. 1).

Each convolution of a 3D kernel over the IFM produces one OFM. Therefore, the number of output feature maps generated in a convolutional layer is the same as the number of kernels at that layer. Some convolutional layers are followed by pooling layers that sub-sample the OFMs by merging neighbor activations into a single one using a max or average function.

The set of convolutional layers is followed by one or more fully connected (FC) layers. A node in a fully connected layer is connected to all nodes of the previous layer. The last FC layer outputs the probabilities of each class of objects with one node for each class. These layers contain most of the weights of a CNN which must be stored and transferred from external memory to on-chip memory. Hence, they are very demanding in terms of memory space and memory bandwidth. Also, while in the convolutional layers the same kernel is used many times, in a FC layer each kernel is used only once.

In all layers each output is followed by an activation function. Several functions exist but recently the Rectified Linear Unit (ReLU) function is commonly used for its simplicity and good results. This function keeps the positive activations unaltered and zeroes the negative ones.

Knowing that convolutional layers are computation intensive and that FC layers are memory bandwidth intensive, several optimization techniques apply to both types of layers while others are more appropriate to only one type of layer.

The most used optimization technique that reduces the computational complexity, the required memory bandwidth and the memory size is to use fixed-point computation instead of floating-point and to reduce the bit width of weights and activations. Different fixed-point representations can be used in different layers which is known as dynamic fixed-point quantization. In this paper we refer to it as mixed fixed-point.

One technique considered only in a few works is zero-skipping. The ReLU function converts negative values to zero and consequently many output activations are zero. Multiplying a weight by zero is useless so the zero-skipping technique does not run these multiplications reducing the processing time. In [28] the authors show for several known networks that an average of up to 50% of input activations are zero. The number of zeros can be increased by applying dynamic pruning to the convolutional layers, which sets activations to zero if their values are below a threshold. The same work has shown that within certain thresholds it is possible to dynamically prune activations without affecting the network accuracy.

Static pruning is also used as an optimization technique to reduce the number of weights. During training, weights below a certain threshold are cut. The technique is normally applied to the fully connected layers where the number of weights is very high and cut percentages of up to 90 % can be used without affecting the network accuracy. Another technique used to reduce the high memory transfer times of weights in the FC layers is the image batch. In this technique, several outputs of the last convolutional layer are calculated and batched before running the FC layers. This way, the weights of FC layers read from memory are then used for a batch of input maps amortizing the transfer time.

Zero-skipping and pruning introduce storage sparsity which complicates the hardware implementation and increases the on-chip memory bandwidth requirements. The image batch technique increases latency but can be used instead of weight pruning keeping the regular structure of the architecture. Therefore, the architecture proposed in this paper implements mixed fixed-point quantization and image batch producing very efficient computing systems for embedded computing in low density FPGAs.

4. Architecture for CNN inference

The proposed architecture targets all types of FPGAs, with an emphasis on low density FPGAs with low memory and computational resources. Therefore, implementations in which all layers are implemented in a pipelined structure that uses on-chip memory to store OFM are not considered, since the on-chip memory resources of low density FPGAs are scarce. Instead, the proposed architecture has two main modules executing in a pipelined fashion: one to execute convolutional layers and another to execute fully connected layers. Both convolutional and fully connected layers could be executed within a single block since the main arithmetic operations of both are dot-products. However, the computational and memory bandwidth requirements of these types of layers are different. The convolutional layers have higher computational requirements, while the fully connected layers have higher memory bandwidth requirements. Consequently, for a better performance efficiency, they are implemented separately.

The hardware modules for the execution of convolutional and fully connected layers run in parallel and include a set of configurable registers to set them for each specific layer. To run a layer, each module is configured with the characteristics of the layer, including the number and size of kernels, source and destination addresses of activations and kernels, the existence of a pooling layer and the fixed-point format.

The input image and the intermediate feature maps are stored in on-chip memory to be processed. The ideal situation is when the on-chip memory is enough to store the whole input image and intermediate feature maps. Since the layers are executed one at a time, this is possible when the on-chip memory can hold the IFMs and OFMs of any layer. In those cases where the on-chip memory is not enough to store the whole initial image, the image is cut into pieces which are convolved separately. If the output feature maps do not fit in the on-chip memory, they are stored in external memory and then reloaded in pieces as IFMs for the next layer. These feature of the architecture permits the execution of large CNN in low density FPGAs.

The convolutions and dot-products are all done by a cluster of processing elements (PE) that run in parallel. Each processing element is responsible for calculating an output feature map. So, each PE processes one kernel at a time. The execution of the CNN in the proposed architecture works as follows:

1. The convolutional and the fully connected modules are configured for the first layers: number of kernels, size of kernels, source and destination addresses of activations and kernels, the existence of a pooling layer and the fixed-point format;
2. The image is loaded to the feature map memory;
3. Kernels are read from external memory and stored on the local memories of the PE cluster. Kernel transfer and kernel processing can be executed at the same time, that is, while one set of kernels are used by the processors, the next set of kernels can be transferred at the same time. This is fundamental in the fully connected layers where the number of computations is the same as the number of weights;
4. After loading the kernels, the image or the IFM are broadcast to all PEs to determine the next output feature maps. In those cases where the number of kernels of a layer is higher than the number of processing elements, output feature maps are calculated in groups and so the process repeats for each new group of kernels. Each output activation associated with a convolution of a kernel is stored back in the feature map memory to be used by the next layer. When a layer is followed by pooling, the activation is stored locally and waits for the other members of the pooling window. Only the pooling result is stored in memory for the next layer;
5. The whole process repeats until finishing the convolution between the image or the IFMs and all the kernels. After that, the next kernels are loaded from memory and the process repeats until running all kernels of a layer, finishing the processing of a layer;
6. The process repeats with the configuration of the next layer.

The architecture proposed in this work implements the functionality described above (see Fig. 2).

The architecture has two clusters of processing elements (PE), one for the convolutional layers and another for the fully connected layers. There is a send/receive activations module for each cluster associated with on-chip memories to store activations. These modules are responsible for reading/writing activations from/to the on-chip memories and send/receive them to/from the PE clusters. They also establish the connection with the data dispatch and interconnect module to access external memory. All data from and to external memory goes through a data dispatch and interconnect module responsible to read and buffer data from external memory, send it to the on-chip memories and write data back to external memory.

4.1. PE Clusters

The PE cluster for convolutions consists of a matrix of cores and an interconnection network to forward input and output activations (see Fig. 3).

Each column of cores is connected to a local memory of the weight memory block. The weights stored in this memory are broadcast to all cores of a column. To allow overlapping of calculation and communication, memories of the weight memory block are dual-port. Each line of cores is connected to a port of the feature map memory (FMM). Activations read from a port of the FMM area broadcast to all cores of the line of cores connected to this port. The number of lines of cores equals the number of ports of the FMM. The number of cores per line of the cluster is limited by the available memory to implement the local memories and by the resources to implement each core. The number of cores per line and per column are statically configurable and therefore the matrix of cores is the same for all layers.

All cores of a column receive the same kernel and therefore contribute to the same OFM (intra-output parallelism). Adding more cores in a column only requires more computing resources,

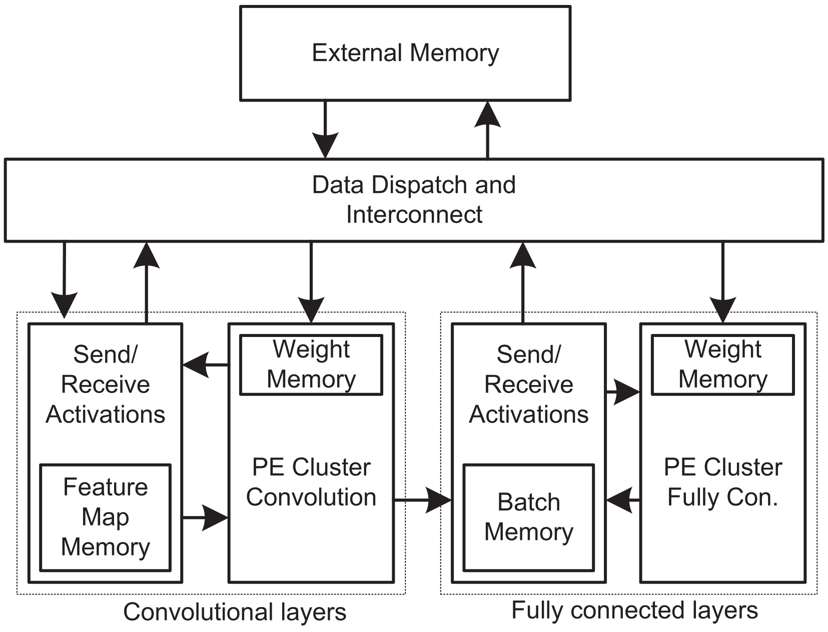

Fig. 2. Block diagram of the proposed architecture.

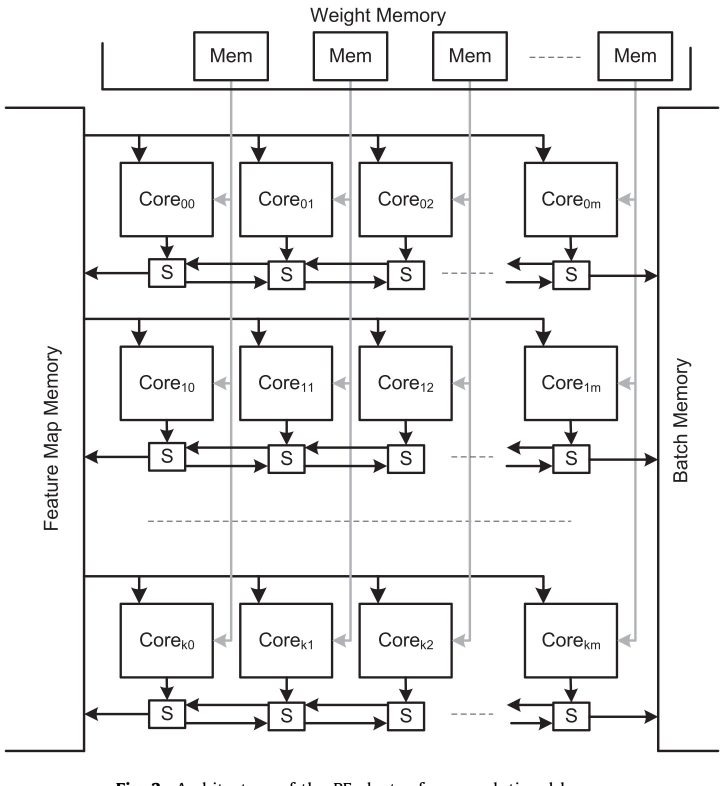

Fig. 3. Architecture of the PE cluster for convolutional layers.

since the local memory is the same independently of the number of cores in its column. Cores in the same line receive different kernels and therefore each produces a different OFM. The more cores there are in a line the more OFM are generated in parallel (inter-output parallelism). The activations calculated by the cores are sent back to the feature map memory if the next layer is convolutional. Otherwise, they are sent to the batch memory to be processed by the fully connected layers. Data transfer from the cores to the on-chip memories is done by small switches, represented in the figure with letter *S*.

The PE cluster for fully connected layers has the same structure of the cluster for convolutions but the number of lines of cores in the cluster is now determined by the batch size. In the fully connected layers the kernels have the same size of the IFM, which is the same as having a single IFM. Each kernel is applied to the whole IFM and produces a single activation. The intra-output parallelism is explored only when batch of OFM of the last convolutional layer is considered. Several OFM of the last convolutional layer can be stored before running the first fully connected layer. In this case, the same kernel can be reused for several batched OFM

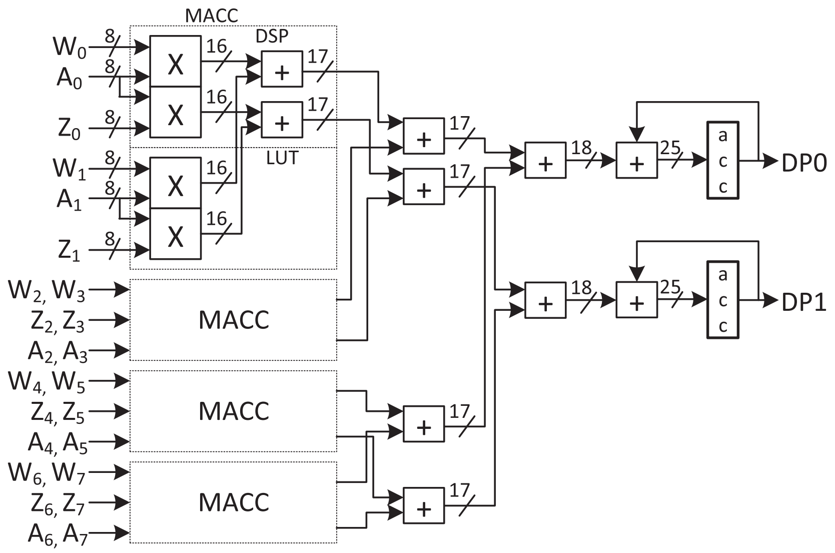

Fig. 4. Architecture of a pair of cores in a line.

reducing the memory bandwidth requirements at the cost of some additional on-chip memory to accumulate OFM and more cores for parallel processing.

The activations generated by the final fully connected layer are sent back to the dispatch and interconnect module to be stored in the external memory.

A core calculates the dot product between the activations and the weights with a multiply-accumulate unit. The clusters of cores are the modules that occupy most of the resources of the architecture and so any area optimization of the core has a great impact in the final area. MAC units are implemented for 8-bit unsigned activations and 8-bit signed weights. Multiplication results are accumulated without bit loss. The final result is shifted according to the fixed-point scale factor and truncated centrally in the send/receive activations modules.

The core has dot-product parallelism to improve performance, that is, eight 8-bit weights and activations are read in parallel in a single memory access. To efficiently use all resources of the FPGA, LUTs and DSPs are both used to implement MACs. One DSP can implement two 8-bit multiplications with one operand in common and two additions [29]. So, one DSP has to be shared between two cores. To implement 8-bit multiplications with LUTs the architecture proposed in [30] is followed. The datapath of two cores sharing DSPs of the same line is illustrated in Fig. 4.

The architecture in the figure determines the dot-product between a common vector of eight activations, $A = [A_0, .., A_7]$, and two vectors of weights, $W = [W_0, .., W_7]$ and $Z = [Z_0, .., Z_7]$, to produce two dot products, $DP0 = A \cdot W$ and $DP1 = A \cdot Z$. Each MAC is implemented with DSPs and LUTs. When there are no more DSPs available, the core is only implemented with LUTs. The architecture is pipelined (not shown in the figure) to improve throughput.

4.2. Send/Receive activations module

The send/receive activations module in the convolutional layers is responsible for reading and writing data from/to the feature map memory. The module transfers data between the external memory and the FMM, reads activations from the FMM and sends them to the PE cluster, and receives activations from the PE cluster and writes them in the FMM. The module contains configurable read and write memory address generators for the FMM. The order of reads and writes of activations from the FMM must take into consideration the window size of the convolution, the size of the maps and the pooling size. Previous approaches calculate 2D convolutions and accumulate the results for all feature maps. The method is inefficient when a single architecture is used for all layers and different layers have different window sizes since the underlying architecture would have to be flexible to support this diversity.

Our architecture follows a different approach. Instead of running multiple 2D convolutions, a single 3D convolution is transformed into a long dot product and so the window size of the convolution is transparent for the processing cores. Pixels of the initial image or activations of the input feature maps and weights of kernels are stored in order $z - x - y$ (see Fig. 5).

Each activation of an output feature is obtained from the dot product between the 3D kernel $x_k \times y_k \times z_k$ and the corresponding activations of the IFM of size $x_p \times y_p \times z_p$ (see Fig. 5b), where z_p is the number of IFMs. The weights of a kernel are all read sequentially from the memories of weights since they are stored in this order. The activations are read in sequence from the FMM but after $x_k \times z_k$ activations the address has to jump to the next y_k adding an offset to the address of the input feature memory being read. For a layer without stride nor followed by pooling, the offset is $x_p \times z_p$. Formally, the dot product to calculate each step of the convolution is given by:

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

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 kernels and the size of the convolution window.

Considering a 3D input feature map of size $x_p \times y_p \times z_p$, a 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 m, the convolution of a kernel, kl, with the 3D input feature map is given by Algorithm 1.

poolFunction is the function to be used in the pooling operation, like maximum or average. The *startAddr* function adds the correct offset to the address pointer of the feature map memory, depending on the next activation to be calculated.

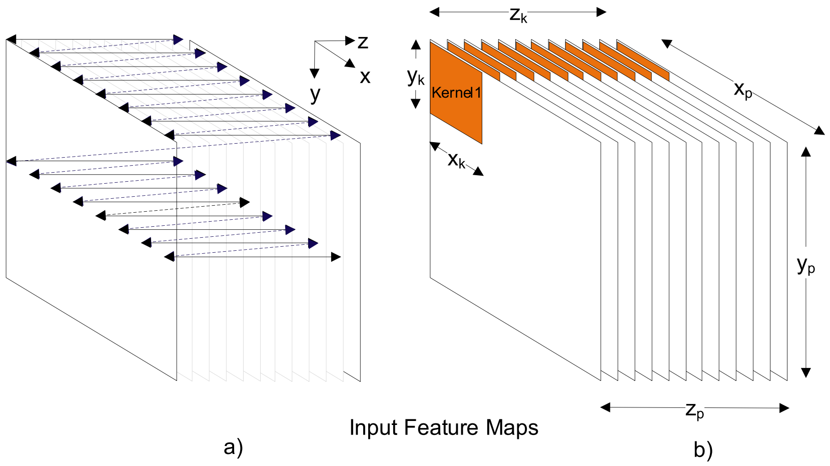

Fig. 5. Reading mode of images, feature maps and weights.

Algorithm 1: Convolution with a 3D kernel.

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

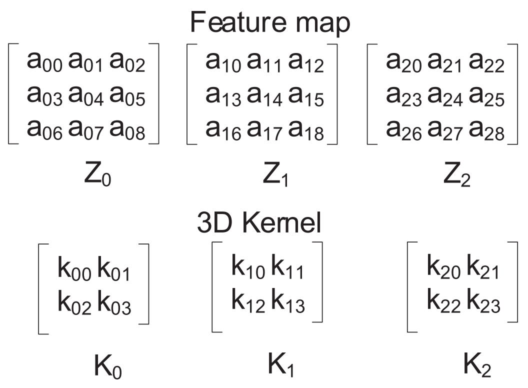
Feature map
a_{00} a_{01} a_{02}
a_{03} a_{04} a_{05}
a_{06} a_{07} a_{08}
a_{10} a_{11} a_{12}
a_{13} a_{14} a_{15}
a_{16} a_{17} a_{18}
a_{20} a_{21} a_{22}
a_{23} a_{24} a_{25}
a_{26} a_{27} a_{28}
Z_0
Z_1
Z_2
3D Kernel
k_{00} k_{01}
k_{02} k_{03}
k_{10} k_{11}
k_{12} k_{13}
k_{20} k_{21}
k_{22} k_{23}
K_0
K_1
K_2

Fig. 6. Example of a 3D feature map and a kernel to be convolved.

Let's consider an example with three input feature maps, Z_0, Z_1, Z_2, and a 3D kernel, K_0, K_1, K_2 (see Fig. 6).

The first kernel convolution is given by:

$$\sum_{i=0}^{1}\sum_{j=0}^{2} a_{ji} \times k_{ji} + \sum_{i=3}^{4}\sum_{j=0}^{2} a_{ji} \times k_{j(i-1)}$$

The next convolution is given by:

$$\sum_{i=1}^{2}\sum_{j=0}^{2} a_{ji} \times k_{ji} + \sum_{i=4}^{5}\sum_{j=0}^{2} a_{ji} \times k_{j(i-1)}$$

and so on with the other two convolutions.

The order of read and write addresses of the activations from/to the FMM are generated by the address generators of the send/receive activations module. The address generators are configured for each layer (there is no hardware reconfiguration, only the registers that store the count limits of the generators are initialized with new values). To improve the performance of the architecture, the FMM explores intra-output parallelism where several activations of an output feature map are generated in parallel (see the block diagram of the FMM in Fig. 7).

Different activations of an output feature map are generated in parallel using the same kernel and different blocks of the IFM. To read different blocks of an IFM, the memory of the FMM is partitioned in several smaller memories. The number of memories is the same as the number of blocks to be processed in parallel. The read and write addresses are common to all memory blocks. At the edges of the IFM blocks, the convolution window needs data stored in a neighbor memory. The output multiplexers permits to select the memory to read data from.

Finally, the result of the activation is shifted (to support mixed fixed-point quantization) and truncated to the number of bits of the activations.

The send/receive activations module for the fully connected layers is simpler since there are no convolutions, only a long dot-product. Data from the batch memory is read and written only once for each kernel. When there is IFM batching, the batch memory consists of several separated memories, one for each map batch. The read and write addresses are common for all memories. The output activations from the cores are also shifted and truncated, like in the convolutional layers, before being written back in the batch memory. Batch memories are dual port to allow simultaneous reading from the fully connected cores and writing from the convolutional cores.

4.3. Data dispatch and interconnect module

The data dispatch and interconnect module contains DMA (Direct Memory Access) blocks to transfer data from the external memory to the on-chip memories of the architecture and vice-versa. Depending on the memory bandwidth requirements of each module, DMAs can be dedicated or shared. DMAs are

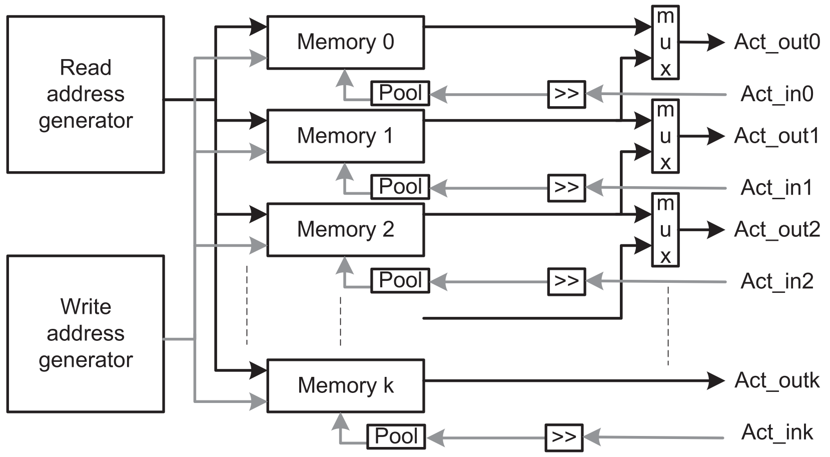

Fig. 7. Architecture of the feature memory buffer.

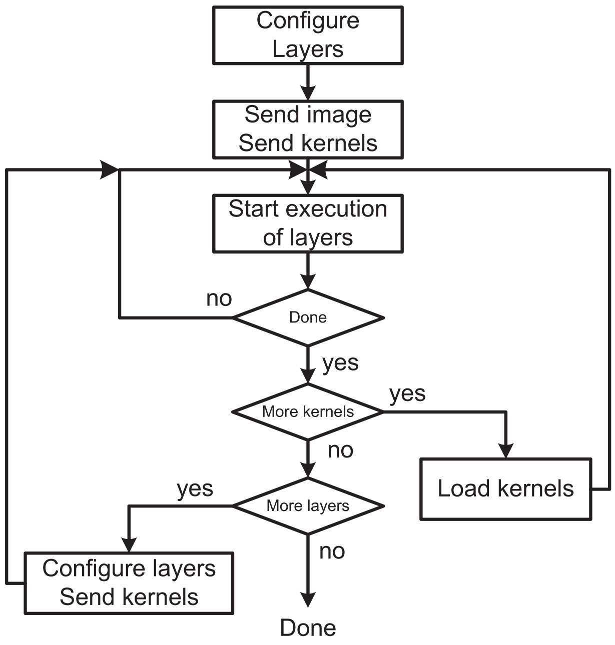

Fig. 8. Execution flow of the CNN model implemented by the ARM processor.

initially programmed for a sequence of transfers. The transferred information contains a header to identify the destination of the data to be transferred.

The dispatch of data is controlled by the ARM processor helped by the internal controller of the architecture. The internal controller informs the processor whenever a layer has finished its execution. The processor is responsible for the configuration of DMAs and to signal the controller when data (image and weights) are loaded and ready to be processed. The processor is also responsible for the configuration of the controller and the architecture for each new layer.

The complete execution of a CNN consists of the execution of a sequence of layers. The ARM processor is programmed according to each CNN model following the execution flow (see Fig. 8).

The overlapping of communication and computation depends on the available on-chip memory and the CNN model. Since this is known before runtime, the optimization is considered when programming the flow in the ARM.

5. Performance and area models of the architecture

The convolutional and the fully connected clusters work in parallel in a pipelined fashion. To obtain the best throughput, delays of both PE clusters should be close to each other as much as possible. The execution times of the clusters depend on the number of cores, which depends on the available hardware resources, and the external memory bandwidth. In the following sections, a performance and an area model of the architecture is described that helps the designer to design the architecture guided by performance and area.

5.1. Performance model

The execution time of the complete CNN, $Texec_{CNN}$ is the sum of the execution time of each convolutional layer, $Texec_{CL}$ plus the time to transfer the image to be inferred and the result, $Tcomm_{image}$, that is

$$Texec_{CNN} = Tcomm_{image} + \sum_{i=1}^{i=CL} Texec_{CL_i} + \sum_{i=1}^{i=FL} Texec_{FL_i} \tag{2}$$

where CL is the number of convolutional layers of the CNN and FL is the number of fully connected layers.

The execution throughput (images/s), Thr_{CNN}, of the architecture is given by

$$Thr_{CNN} = \frac{1}{\max\left(Tcomm_{image} + \sum_{i=1}^{i=CL} Texec_{CL_i}, \frac{\sum_{i=1}^{i=FL} Texec_{FL_i}}{fcCoreL}\right)} \tag{3}$$

where $fcCoreL$ is the number of lines of cores in the FC cluster, that is, the batch size. This throughput is in fact an average throughput since $fcCoreL$ results are produced for each execution of the FC layers.

The time to read the image from external memory depends on the image size, $imageS$, and the memory bandwidth to external memory, BW (bytes/s), as follows:

$$Tcomm_{image} = \frac{imageS}{BW} \tag{4}$$

The time to execute a convolutional layer depends on the configuration of the architecture and on the features of the layer. Associated with the architecture there is the number of cores in each line of the convolutional cluster, $convCoreL$, the number of ports of the FMM, $convCoreC$, and the operating frequency of the architecture, $freq$. The features associated with the layer are the number of 3D kernels, *nKernel*, and the size of the 3D kernels, *kernelSize*, which is the same for all kernels.

It is assumed that each weight memory of the fully connected cluster has enough space to hold two kernels, so that kernel communication can overlap with kernel computation. The storing of the new output activations generated in a layer overlaps with the calculation of the activations. This is essential in the fully connected layers since the weight kernels are not reused like those used in the convolutional layers.

Another aspect has to do with the size of the image and feature maps. If any of these do not fit in the FMM the layer is executed in steps, that is, the image or the IFM are divided and processed separately. For example, an image may have to be divided in two and the half images are processed one after the other. Also, if the OFM do not fit the FMM, then it is stored in external memory and then reloaded to be processed in steps. It means that the weights have to be read from external memory as many times as the number of image or IFM partitions.

Considering these aspects, the execution time a convolutional layer depends on the number of cycles to do the convolutions, the time to transfer the IFM when it had to be stored in external memory, the time to transfer *convCoreL* kernels and the time to write the last *convCoreL* activations.

The number of cycles to execute a convolutional layer, *convCycle*, is given by Eq. (5).

$$convCycle = \frac{1}{convCoreC}\left\lceil \frac{nKernel}{convCoreL} \right\rceil \times \frac{nConv \times kernelSize}{nMAC} \tag{5}$$

where *nConv* is the number of 3D convolutions and *nMAC* is the number of parallel multiply-accumulations of each core.

The volume of non-overlapping data to be transferred in the layer, *dCommCL*, depends on the necessity to use external memory (EM) to store the OFM, OFM_{size}, that is

$$dCommCL = convCoreL \times kernelSize \qquad \text{without EM} \tag{6}$$

$$dCommCL = convCoreL \times kernelSize \times nPartial + OFM_{size} \quad \text{with EM} \tag{7}$$

where *nPartial* is the number of partitions of the input feature map.

From these equations, the total execution time of a layer is given by Eq. (8).

$$Texec_{CL} = \frac{convCycle}{freq} + \frac{dCommCL}{BW} \tag{8}$$

In the case of FC layers, the IFM and OFM are small, compared to those of the convolutional layers, and so it is assumed that there is enough batch memory to hold them. The number of cycles to execute a FC layer, *fcCycle*, is given by Eq. (9).

$$fcCycle = \left\lceil \frac{nKernel}{FCCoreC} \right\rceil \times \frac{kernelSize}{nMAC} \tag{9}$$

where *fcCoreC* is the number of cores in each column of the fully connected cluster.

The volume of data to be transferred in a FC layer, *dCommFL*, is given by Eq. (10)).

$$dCommFL = nKernel \times kernelSize \tag{10}$$

From these equations, the total execution time of a FC layer, $Texec_{FL}$, is given by Eq. (11).

$$Texec_{FL} = \max\left(\frac{fcCycle}{freq}, \frac{dCommFL}{BW}\right) \tag{11}$$

The performance model just described does not include the time to configure the layers. It depends on the unit that does this configuration but the configuration data is negligible compared to all weights and activations to be transferred. Therefore, it was not included in the model.

5.2. Area model

For the area model, the computational and the on-chip memory resources were considered.

The total computational area, A_{comp}, is given by:

$$A_{comp} = AC_{ctrl} + AC_{interC} + convCoreL \times convCoreC \times AC_{cCore} + fcCoreL \times fcCoreC \times AC_{fcCore} + AC_{SRAconv} + AC_{SRAfc} \tag{12}$$

where AC_{ctrl} is the area of the system controller, AC_{interC} is the area of the data dispatch and interconnect module, $AC_{SRAconv}$ is the area of the send/receive activations module of convolutional cluster, AC_{SRAfc} is the area of the send/receive activations module of fully connected cluster, AC_{cCore} is the area of the convolutional core and AC_{fcCore} is the area of the FC core.

The area is computed as the number of LUTs and DSPs. A_{comp} considers DSPs and LUTs as separate resources, that is, two instances of A_{comp} are considered, one for DSPs and another for LUTs.

The total on-chip memory resources, A_{mem}, is given by

$$A_{mem} = AM_{interC} + AM_{SRAconv} + AM_{SRAfc} + convCoreL \times AM_{cCore} + \tag{13}$$

$$+ fcCoreL \times AM_{fcCore} + fcCoreC \times AM_{batch} \tag{14}$$

where AM_{interC} is the memory size of the data dispatch and interconnect module, $AM_{SRAconv}$ is the memory size of the send/receive activations module of the convolutional cluster, AM_{SRAfc} is the memory size of the send/receive activations module of the fully connected cluster, AM_{cCore} is the local memory size of the convolutional cluster, A_{fcCore} is the local memory size of the FC cluster and AM_{batch} is the memory required to store an instance batch of the batch memory. In this case, the memory unit is one 4KBytes BRAM.

5.3. Designing with the models

From the performance and area models just described, the number of cores of each cluster and the batch size is explored restricted by the total resources of the FPGA and the external memory bandwidth searching for the best performance. This design space exploration is still manual.

The designer must guarantee that each local memory of the fully connected cluster is enough to hold two kernels to guarantee overlapping of computation and communication. Also, when possible, guarantee that the FMM can hold the IFM and the OFM of any convolutional layer and that the batch memory is enough to hold two OFM of the previous convolutional layer for each batch instance, and one IFM and one OFM of the fully connected layer.

The number of cores of the proposed architecture is configurable and depends on the available resources. Both clusters of cores of the proposed architecture run in parallel with a pipeline structure. Hence, to obtain a balanced pipeline, the execution times of both must be equal, that is:

$$Tcomm_{image} + \sum_{i=1}^{i=CL} Texec_{CL_i} = \sum_{i=1}^{i=FL} Texec_{FL_i} \tag{15}$$

This condition is constrained by the available on-chip memory, *OCM*, the available computing resources, *COMP*, and the available external memory bandwidth, *BW*. The required on-chip memory A_{mem} must be lower than OCM and the necessary computation resources, A_{comp} must be lower than *COMP*. The available bandwidth must be distributed according to the communication requirements of each cluster. The ratios between the data communication of convolutional layers, *convComm*, and the data communication of the FC layers, *fcComm*, are determined and then the bandwidth is distributed with the same ratio.

Given the available resources and the available memory bandwidth to external memory, the manual design space exploration starts with a particular batch and a particular number of convolutional and fully connected cores whose total number of resources (LTs, DSPs and BRAMs) do not exceed the available resources, according to the area model. From each particular configuration of the architecture, the performance is estimated. The performance model gives us the execution time of convolutional layers and of fully connected layers. To balance these figures according to Eq. (15), we change the number of cores used to run each type of layers and the memory bandwidth reserved for each set of cores. Two performance metrics are used to assess the quality of the solution: image throughput and performance efficiency (ratio between estimated performance and peak performance). Given the limited memory bandwidth, increasing the number of cores reduces the performance efficiency whenever there is a bottleneck associated with the memory bandwidth.

6. Results

All architectures were tested with two regular CNNs: AlexNet and VGG16. All architectures were implemented with Vivado 2019.1 targeting a ZedBoard with a ZYNQ XC7Z020 (Artix-7 FPGA with a dual ARM Cortex-A9 CPU) and a Xilinx SoC ZC706 Evaluation Kit with a XC7Z7045 FPGA (Kintex-7 FPGA with a dual ARM Cortex-A9 CPU). The programmable logic has four 64-bit High-Performance (HP) ports with direct access to external memory working at 150 MHz with a total bandwidth of 4.8 GByte/s. The real data transfer in the ZedBoard with these ports was measured and achieved around 3.3 GBytes/s and this is the bandwidth considered during the design of the architecture. In the ZC706 board, the DRR3 memory connected to the programmable logic side supports up to 4.8 GBytes/s. The real data transfer of a ZC706 board is 4 GBytes/s. The runtime configuration of the architecture is done by the ARM processor. The architecture runs at 200 MHz in the Artix-7 technology and at 250 MHz in the Kintex-7 technology. In all implementations, 8-bit mixed fixed-point is used to represent weights and activations that guarantees very low accuracy drop (1% and below) [26].

To test and assess the quality of the proposed configurable architecture, the next steps were followed:

1. Two instances of the architecture for two different CNNs were designed using the performance and area models targeting two different FPGAs. The models are used to help the designer find an appropriate number of cores constrained by the area of the target FPGA. This process is still manual. However, the hardware description of the architecture uses generics that facilitate the design of the architecture. Even the type of the cores are specified with generics and so there is no extra effort to design the architecture in order to use different types of cores;
2. The designs were tested on the board and the area and performance were determined;
3. The measured area and performance of these four architectures were compared with the results obtained from the models to access their precision for the same four architectures;
4. The area results are quite close to the values obtained with the area model since the model considers the area of each individual block of the architecture obtained from the implementation. The results obtained with the performance model are within 4% of the measured results. Considering this to be an acceptable error, a set of studies based on the models were done to determine some important tradeoffs: performance versus memory bandwidth and communication versus computation.

Table 1
Resources occupation of three different implementations of the core.

Type	Core 0	Core 1	Core 2
LUTs	305	411	517
DSPs	2	1	0

Table 2
Configuration and Area of the proposed architecture running the inference of AlexNet in both FPGA platforms.

	ZYNQ7020			ZYNQ7045		
ConvCores	16×7			48×8		
FCcores	2×6			1×40		
Cores Type 0/1/2	108/4/12			280/104/40		
BATCH	6			40		
Feature Mem. Size	$28K \times 8bytes$			$56K \times 8bytes$		
Batch Mem. Size	$6 \times 2K \times 8bytes$			$40 \times 2K \times 8bytes$		
Weight Mem. Size	$1K \times 8bytes$			$1K \times 8bytes$		
	LUT	DSP	BRAM	LUT	DSP	BRAM
Convolutional Cluster	34584	220	0	128144	664	0
FC Cluster	6204	0	0	20680	0	0
Feature Memory	957	0	56	1068	0	112
Batch Memory	535	0	24	2613	0	160
Data Dispatch + Control	3530	0	18	3832	0	18
Weight Mem	120	0	34	360	0	98
Total	46130	220	134	156951	664	388

Table 3
Performance results of the proposed architecture for high throughput running the inference of AlexNet in both FPGA platforms.

	ZYNQ7020	ZYNQ7045
Images/s	227	774
Performance (GOPs)	329	1120
% Peak Performance	83	66

All results are reported in the following sections.

6.1. Characterization of the proposed configurable architecture using AlexNet

The convolutional and the fully connected cores occupy most of the area of the architecture. To balance the utilization of both LUT and DSP resources of the FPGA, different implementations of the cores are considered with two, one or no DSPs (see the resources occupation of the cores in Table 1).

The proposed architecture has been configured for high throughput (images/s) in the inference of AlexNet in both FPGA platforms. Using the performance and area models introduced in the previous section, the number of convolutional cores, fully connected cores and batch size were calculated for high throughput. The architectures were then implemented and executed on each FPGA platform. The number of resources and the performance of the designed systems are described in Tables 2 and 3.

In the ZYNQ7020 FPGA a throughput of 227 images/s was achieved that corresponds to a real performance of 329 GOPs, around 83% of the peak performance. Therefore, it has a high performance efficiency given that a single programmable module is being used to run all layers. In the ZYNQ7045 there are around $4\times$ more hardware resources and so it was expected a much higher performance. In this device our architecture runs the AlexNet inference with a throughput of 774 images/s, which corresponds to a performance of 1120 GOPs. This is $3.4\times$ the per-

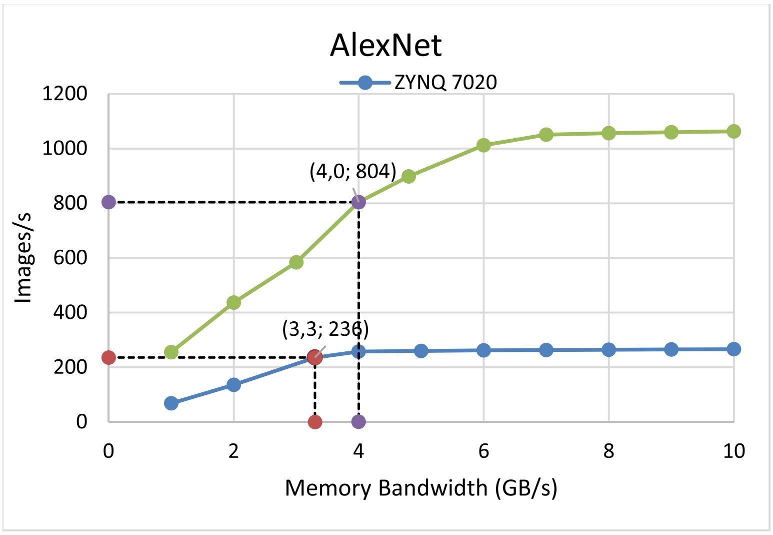

Fig. 9. Estimated throughput of the architecture for different memory bandwidths.

formance achieved in the ZYNQ7020. The percentage of peak performance is lower (66%) because the increase in resources was not followed by an increase in the external memory bandwidth. The ZYNQ7045 still have resources to improve the performance, but the higher FPGA occupation leads to a lower operating frequency and a lower performance efficiency. With 512 convolutional cores the architecture processes 837 images/s with a peak performance over two TOPs but the performance efficiency reduces to around 55%. It means that the communication becomes the bottleneck and the cores are underutilized. The almost linear increase of performance with the number of resources shows that the proposed architecture is scalable and can be efficiently implemented in FPGAs with more resources.

To determine the influence of the external memory bandwidth over the performance of the system, the throughput performance of the architecture was estimated considering FPGA boards with a different memory bandwidth (see Fig. 9).

The points identified in the figure correspond to the performance points obtained with the memory bandwidth of the tested boards. According to the results illustrated in the figure, the bandwidth limits the performance of the system in both platforms. How much is this limitation depends on the type of FPGA. With the ZYNQ7020 the performance improvements achieved with a memory bandwidth higher than 4 GBytes/s are negligible, that is, the available resources limit the performance improvement. The performance achieved with the real memory bandwidth, 3.3 GBytes/s, of the ZedBoard is only 12% above the maximum achievable performance. For the ZYNQ7045, as expected, a higher memory bandwidth is needed to take advantage of all the available resources of the FPGA. In this case, a memory bandwidth of 7 GBytes/s permits to achieve a performance of only 1% from the peak performance. Also, the real memory bandwidth of the board limits the performance to 25% from the peak performance.

Considering the implementations of Table 3, the transfer and processing times of each layer have been determined using the performance and area models (see Fig. 10).

For a perfect processing and communication balancing both delays in each layer should be equal. The processing only dominates in the first two layers. So, in general, the communication dominates the execution time of the architecture, that is, the cores are underutilized. To improve the performance efficiency of the architecture, the number of cores can be reduced to balance the execution time with the communication time. However, reducing the number of cores also reduces the total performance of the system. The designer must choose an appropriate point depending on the project requirements.

The results presented so far are for high throughput. However, a specific application may simply require a particular throughput. Considering this, several implementations were generated with specific estimated image processing throughputs and the occupation of resources in terms of LUTs and DSPs was determined (see Fig. 11).

The number of LUTs and DSPs are related to each other since the arithmetic operations in the cores can be implemented with LUTs or DSPs. The implementations whose occupation of resources are presented in the graphs consider a balanced occupation of LUTs and DSPs. As can be observed, it is possible to achieve real-time processing (assuming real-time processing as 30 images/s) with only one fourth of the resources of a ZYNQ7020. This means that there is a considerable margin to increase the complexity of the network in terms of weights and number of computations and still achieve real-time processing. Also, the proposed architecture can be used as a core within a more complex image processing application using only a fraction of the FPGA.

6.2. Implementation of the architecture for VGG16

To show the usability of the architecture to run larger CNN, an architecture to run the inference of VGG16 was implemented and configured. This is a much larger network when compared to AlexNet, as described in the introduction, with more weights and more computations. One important difference with a great impact over the performance is the fact that while in the AlexNet the IFM plus the OFM of any layer fit in the feature map memory, in the case of VGG, the feature maps of the first layers do not fit the internal memory and have to be stored in external memory. This forces the feature maps to be processed in smaller maps with a consequent increase in the volume of data transferred between the core and the external memory. VGG16 was implemented in both FPGAs for high throughput (images/s) (see results in Tables 4 and 5).

In the ZYNQ7020 FPGA, a throughput of only 12 images/s was achieved, which corresponds to a real performance of 385 GOPs, around 93% of the peak performance. This is a very high performance efficiency. In the ZYNQ7045 it was possible to achieve 53 images/s with a performance of 1632 GOP/s and a perfor-

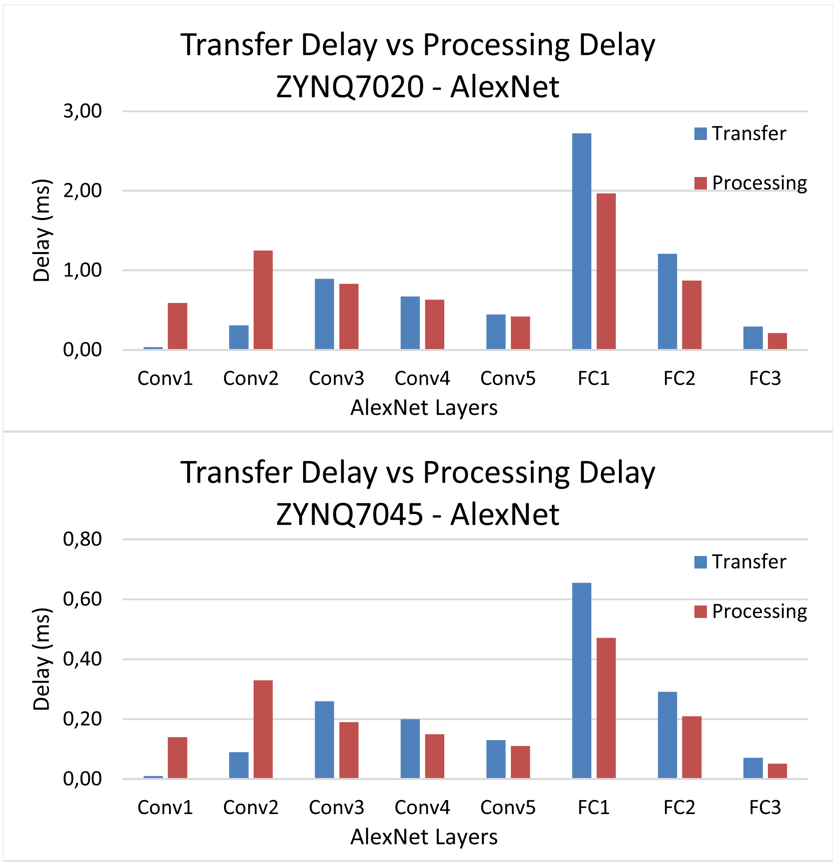

Fig. 10. Transfer and processing delays for each layer of AlexNet.

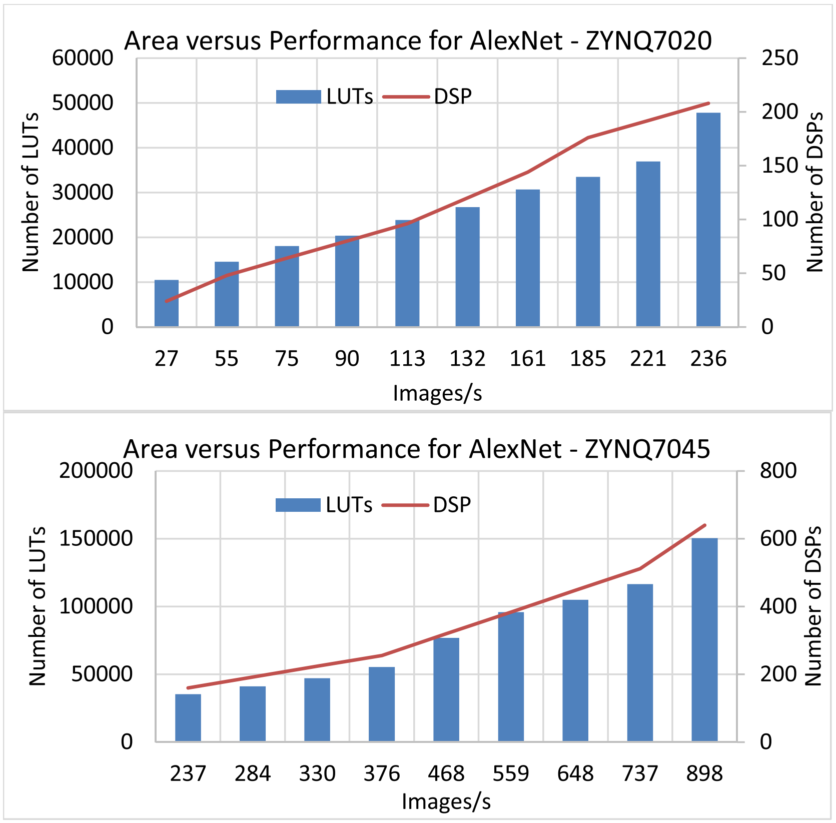

Fig. 11. Trade-off between occupation of resources and image throughput for inference of AlexNet in ZYNQ 7020 and ZYNQ7045.

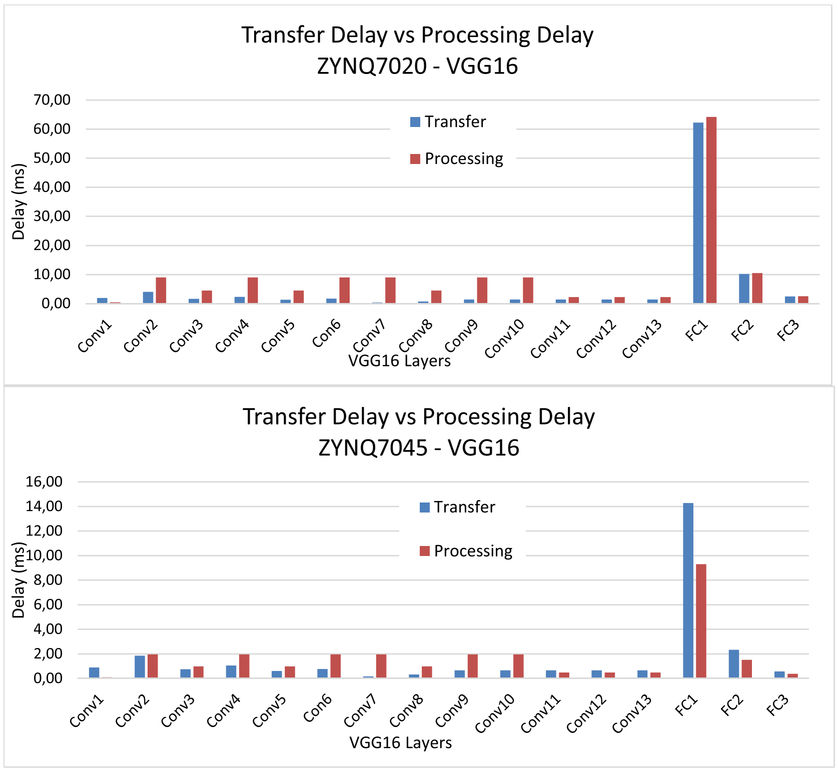

Fig. 12. Transfer and processing delays for each layer of VGG16.

Table 4
Configuration and area of the proposed architecture running the inference of VGG16 in both FPGA platforms.

	ZYNQ7020			ZYNQ7045		
ConvCores	16×8			64×8		
FCcores	1×1			1×6		
BATCH	1			6		
Feature Mem. Size	$32K \times 8bytes$			$144K \times 8bytes$		
Batch Mem. Size	$1 \times 2K \times 8bytes$			$6 \times 2K \times 8bytes$		
Weight Mem. Size	$1K \times 8bytes$			$1K \times 8bytes$		
	LUT	DSP	BRAM	LUT	DSP	BRAM
Convolutional Cluster	42856	220	0	178632	812	0
FC Cluster	517	0	0	1830	12	0
Feature Memory	1048	0	64	1308	0	288
Batch Memory	230	0	4	539	0	124
Data Dispatch + Control	3530	0	18	3940	0	18
Weight Mem	120	0	34	480	0	130
Total	48535	220	120	187007	824	460

Table 5
Performance results of the proposed architecture for best throughput running the inference of VGG16 in both FPGA platforms.

	ZYNQ7020	ZYNQ7045
Images/s	12	53
Performance (GOPs)	385	1632
% Peak Performance	93	86

mance efficiency of 86 %. Note that the operating frequency of the ZYNQ7045 implementation decreased slightly to 230 MHz due to the higher occupation of the FPGA.

The transfer and processing times of each layer of VGG16 was also determined (see Fig. 12).

The detailed delays of each layer of VGG16 help us understand the high performance efficiency obtained. In the ZYNQ7020 all data transfers overlap with data processing, except in the first layer. So, almost all cores are always working during the whole inference processing. For the ZYNQ7045, the transfer time is higher than the processing delays. So, the fully connected cores are underutilized and consequently reduces the performance efficiency.

Table 6
Performance comparison of the proposed architecture with previous works running AlexNet and VGG in ZYNQ7020 and ZYNQ7045.

	This Work	This Work	[19]	This Work	[26]	This Work
CNN	AlexNet	AlexNet	AlexNet	VGG16	VGG16	VGG16
FPGA	7020	7045	7045	7020	7020	7045
LUTs	46130	156951	86262	48535	29867	187007
BRAMs	134	388	303	116	86	460
DSPs	220	664	808	220	190	824
BW	3.3	4.0	10.8	3.3	4.2	4.0
Freq	200	250	200	200	214	230
Images/s	227	774	340	12	2.7	53
GOPs	329	1120	493	385	84	1632
GOPs/kLUTs	7.1	7.1	5.7	7.9	2.8	8.7
GOPs/DSPs	1.5	1.7	0.6	1.8	0.44	2.0

6.3. Comparison with the state of the art

The proposed architecture was compared with previous works. Only a few works consider 8-bit mixed fixed-point representations and the ZYNQ7020 as the target device (see Table 6).

Compared to related work our architectures have better performance and better area efficiency in both types of networks and FPGAs.

7. Conclusions and future work

In this work, a configurable architecture for the inference execution of CNNs was proposed. The architecture targets both low and high density FPGAs. To improve performance and reduce hardware with negligible accuracy loss 8-bit mixed fixed-point representation for weights and activations was considered.

The architecture is scalable and has high performance and area efficiencies. Compared to state of the art works over the same FPGA devices, the proposed architecture improves the image inference throughput. Also, it has been shown that it is possible to run large networks in a low density FPGA with acceptable performance.

The next step is to study the impact of hybrid quantization (different fixed-point datawidths in different layers) over the area and the performance of the architecture. It is also planed to extend the architecture with new modules to support irregular CNNs.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020 and through IPL, Instituto Politécnico de Lisboa, under project IPL/IDI&CA/2019/inCNeuralN/ISEL.

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, L. Fei-Fei, Imagenet large scale visual recognition challenge, Int. J. Comput. Vis. 115 (3) (2015) 211–252.
[2] Y.L. Cun, L.D. Jackel, B. Boser, J.S. Denker, H.P. Graf, I. Guyon, D. Henderson, R.E. Howard, W. Hubbard, Handwritten digit recognition: applications of neural network chips and automatic learning, IEEE Commun. Mag. 27 (11) (1989) 41–46.
[3] A. Krizhevsky, I. Sutskever, G.E. Hinton, ImageNet classification with deep convolutional neural networks, in: Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1, ser. NIPS'12, Curran Associates Inc., USA, 2012, pp. 1097–1105.
[4] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, in: Proceedings of the 3rd International Conference on Learning Representations, 2015.
[5] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015, pp. 1–9. June
[6] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), June 2016, pp. 770–778.
[7] J. Zhang, J. Li, Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network, in: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA '17, 2017, ACM, New York, NY, USA, 2017, pp. 25–34.
[8] S.I. Venieris, C. Bouganis, fpgaConvNet: mapping regular and irregular convolutional neural networks on FPGAs, IEEE Trans. Neural Netw. Learn.Syst. (2018) 1–17.
[9] S. Chakradhar, M. Sankaradas, V. Jakkula, S. Cadambi, A dynamically configurable coprocessor for convolutional neural networks, SIGARCH Comput. Archit. News 38 (3) (2010) 247–257.
[10] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, O. Temam, DaDianNao: a machine-learning supercomputer, in: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, Dec 2014, pp. 609–622.
[11] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, J. Cong, Optimizing FPGA-based accelerator design for deep convolutional neural networks, in: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA '15, ACM, New York, NY, USA, 2015, pp. 161–170.
[12] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, H. Yang, Going deeper with embedded FPGA platform for convolutional neural network, in: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA '16, ACM, New York, NY, USA, 2016, pp. 26–35.
[13] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, Y. Cao, 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, ser. FPGA '16, ACM, New York, NY, USA, 2016, pp. 16–25.
[14] Y. Qiao, J. Shen, T. Xiao, Q. Yang, M. Wen, C. Zhang, FPGA-accelerated deep convolutional neural networks for high throughput and energy efficiency, Concurrencyand Computation: Practice and Experience 29 (20) (2017) e3850–n/a. E3850 cpe.3850
[15] Z. Liu, Y. Dou, J. Jiang, J. Xu, S. Li, Y. Zhou, Y. Xu, Throughput-optimized FPGA accelerator for deep convolutional neural networks, ACM Trans. Reconfigurable Technol. Syst. 10 (3) (2017) 17:1–17:23.
[16] M. Alwani, H. Chen, M. Ferdman, P. Milder, Fused-layer CNN accelerators, in: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Oct 2016, pp. 1–12.
[17] Y. Shen, M. Ferdman, P. Milder, Maximizing CNN accelerator efficiency through resource partitioning, in: Proceedings of the 44th Annual International Symposium on Computer Architecture, ser. ISCA '17, ACM, New York, NY, USA, 2017, pp. 535–547. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080221.
[18] P. Gysel, M. Motamedi, S. Ghiasi, Hardware-oriented approximation of convolutional neural networks, in: Proceedings of the 4th International Conference on Learning Representations, 2016.
[19] J. Wang, Q. Lou, X. Zhang, C. Zhu, Y. Lin, D. Chen, 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, 2018.
[20] Y. Umuroglu, N.J. Fraser, G. Gambardella, M. Blott, P.H.W. Leong, M. Jahre, K.A. Vissers, FINN: A framework for fast, scalable binarized neural network inference, CoRR (2016) abs/1612.07119.
[21] S. Liang, S. Yin, L. Liu, W. Luk, S. Wei, FP-BNN: Binarized neural network on FPGA, Neurocomputing 275 (2018) 1072–1086. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231217315655.
[22] S. Han, H. Mao, W.J. Dally, Deep compression: compressing deep neural network with pruning, trained quantization and huffman coding, CoRR (2015) abs/1510.00149.
[23] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, S. Mahlke, Scalpel: customizing DNN pruning to the underlying hardware parallelism, SIGARCH Comput. Archit. News 45 (2) (2017) 548–560. [Online]. Available: http://doi.acm.org/10.1145/3140659.3080215.
[24] M. Motamedi, P. Gysel, V. Akella, S. Ghiasi, Design space exploration of FPGA-based deep convolutional neural networks, in: 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC), Jan 2016, pp. 575–580.
[25] Y. Shen, M. Ferdman, P. Milder, Escher: a CNN accelerator with flexible buffering to minimize off-chip transfer, in: 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), April 2017, pp. 93–100.
[26] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, H. Yang, Angel-Eye: a complete design flow for mapping CNN onto embedded FPGA, IEEE Trans. Comput.-Aided Des.Integr. Circuits Syst. 37 (1) (2018) 35–47.
[27] L. Gong, C. Wang, X. Li, H. Chen, X. Zhou, MALOC: A fully pipelined FPGA accelerator for convolutional neural networks with all layers mapped on chip, IEEE Trans. Comput.-Aided Des.Integr. Circuits Syst. 37 (11) (2018) 2601–2612.
[28] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N.E. Jerger, A. Moshovos, Cnvlutin: ineffectual-neuron-free deep neural network computing, in: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), June 2016, pp. 1–13.
[29] M. Véstias, R.P. Duarte, J.T. de Sousa, H. Neto, Parallel dot-products for deep learning on FPGA, in: 2017 27th International Conference on Field Programmable Logic and Applications (FPL), Sept 2017, pp. 1–4.
[30] E.G. Walters, Array multipliers for high throughput in Xilinx FPGAs with 6-input luts, Computers 5 (4) (2016). [Online]. Available: http://www.mdpi.com/2073-431X/5/4/20.

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 Ph.D. in Electrical and Computer Engineering from the Technical University of Lisbon.

Rui P. Duarte is an Assistant Professor at the University of Lisbon, School of Engineering (IST), Department of Electrical and Computer Engineering (DEEC). He is a senior researcher at INESC-ID, a research institute associated with the Engineering University, IST. His main research interests are Digital Systems Design and Computer Architecture. He has a Ph.D. in Electrical and Computer Engineering from the Technical University of Lisbon.

José T. de Sousa is an Assistant Professor at the University of Lisbon, School of Engineering (IST), Department of Electrical and Computer Engineering (DEEC). He is a senior researcher at INESC-ID, a research institute associated with the Engineering University, IST. His main research interests are Digital Systems Design and Computer Architecture. He has a Ph.D. in Electrical and Computer Engineering from the Technical University of Lisbon.

Horácio C. Neto is an Associated Professor at the University of Lisbon, School of Engineering (IST), Department of Electrical and Computer Engineering (DEEC). He is responsible for the Electronic Systems Design and Automation (ESDA) research group at 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. He has a Ph.D. in Electrical and Computer Engineering from the Technical University of Lisbon.