Argus CNN Accelerator Based on Kernel Clustering and Resource-Aware Pruning

Damjan M. Rakanovic*, Vuk Vranjkovic, Rastislav J. R. Struharik
Faculty of Technical Sciences, Department of Power, Electronics and Telecommunications, University of Novi Sad,
Novi Sad, Serbia
rakanovic.de2.2015@uns.ac.rs

***Abstract*—Paper proposes a two-step Convolutional Neural Network (CNN) pruning algorithm and resource-efficient Field-programmable gate array (FPGA) CNN accelerator named "Argus". The proposed CNN pruning algorithm first combines similar kernels into clusters, which are then pruned using the same regular pruning pattern. The pruning algorithm is carefully tailored for FPGAs, considering their resource characteristics. Regular sparsity results in high Multiply-accumulate (MAC) efficiency, reducing the amount of logic required to balance workloads among different MAC units. As a result, the Argus accelerator requires about 170 Look-up tables (LUTs) per Digital Signal Processor (DSP) block. This number is close to the average LUT/DPS ratio for various FPGA families, enabling balanced resource utilization when implementing Argus. Benchmarks conducted using Xilinx Zynq Ultrascale + Multi-Processor System-on-Chip (MPSoC) indicate that Argus is achieving up to 25 times higher frames per second than NullHop, 2 and 2.5 times higher than NEURAghe and Snowflake, respectively, and 2 times higher than NVDLA. Argus shows comparable performance to MIT's Eyeriss v2 and Caffeine, requiring up to 3 times less memory bandwidth and utilizing 4 times fewer DSP blocks, respectively. Besides the absolute performance, Argus has at least 1.3 and 2 times better GOP/s/DSP and GOP/s/Block-RAM (BRAM) ratios, while being competitive in terms of GOP/s/LUT, compared to some of the state-of-the-art solutions.**

***Index Terms*—Machine Learning; Accelerator architecture; Convolutional Neural Network pruning; Edge-based computing.**

I. INTRODUCTION

Deep learning [1] has become one of the most powerful tools for solving a wide range of problems in different fields [2], [3]. One of the most used members of the Deep learning field today are Convolutional Neural Networks (CNN). Theoretical foundations of CNNs have been developed twenty years ago [4], but the first successful CNN architecture was the winning algorithm of the Image classification competition in 2012, widely known as AlexNet [5]. From that time, every winning entry in the competition was from the class of CNNs. However, the exceptional accuracy of CNNs comes with high computational and storage costs. One of the most demanding CNNs in terms of computational load and storage is VGG-16 [6]. It performs almost 31 billion operations to classify one image with a resolution of just 224×224 pixels. Although VGG-16 is very regular in terms of kernel size and layer structure, its accuracy is low considering recent, more complex architectures, like Inception [7], ResNet [8], NASNet [9], and MobileNet [10]. The improvements are mainly derived from much deeper network structures compared to only 16 layers of VGG. Even though the number of parameters has dramatically decreased (from 138 million in VGG-16 to 23 million for Inception v3), the additional layers and their structures introduced new complexity for dedicated CNN hardware due to the different data flows required to process each new layer type. Different types of kernels and filter numbers per layer change our view of how underlining CNN hardware should be developed to accommodate current and future improvements in the field. Another layer of complexity was added by demand to efficiently process the compressed CNN [11].

The development of specialized CNN hardware accelerators started almost immediately with the introduction of CNNs. Some of successful Application-specific integrated circuit (ASIC) architectures are Eyeris v2 [12], Cambricon-x [13], Eyeris [14], NullHop [15], DaDianNao [16], SparseNN [17], ENVISION [18], Thinker [19], UNPU [20]. Significant growth in the number of proposed Field-programmable gate array (FPGA) CNN accelerators was mainly driven by the introduction of more flexible and versatile FPGA-based SoCs, like the Xilinx Zynq family. While ASIC solutions almost always deliver the best performance, modern FPGAs offer comparable performance and acceptable power consumption with the advantage of possible reconfiguration, which can help accommodate new CNN layer types.

Most of the dedicated hardware architectures, both ASIC and FPGA, use a 2D array of Processing Elements (PE)

Manuscript received 28 October, 2020; accepted 2 March, 2021.

This project has received partial funding from the European Union's Horizon 2020 research and innovation programme under Grant No. 856967. It has also been partially funded from the Faculty of Technical Sciences Novi Sad, Department of Power, Electronics and Telecommunications, as part of the project "Research in the fields of power, electronics, telecommunications and applied information systems for the modernization of study programs".

which are built around MAC units, with additional local memory for storing intermediate results of computations. Additional hardware is responsible for feeding this array with weights and input feature map (IFM) activations. This approach is very efficient when the layers are large in terms of the number of kernels and IFMs, like in VGG-16 and AlexNet. In this case, there is a big reuse of IFM points, which results in simple broadcast networks over PEs. With the introduction of new layer types, like the Depthwise layer in MobileNet v1 [10], and layers that have a smaller number of kernels than the size of a 2D array of PEs, the efficiency of this approach significantly decrees. For example, we can observe that the greatest improvement in performance between Eyeriss v1 [14] and v2 comes from data routing networks. This is a typical example of the fact that the number of PEs is not the only factor that defines the performance of the architecture, but rather both the number of PEs and the clustering of PEs in smaller groups with dedicated data buffers.

Unlike the approach that uses a 2D array of PEs, Argus has a dedicated PE for every channel of the output feature map (OFM). This approach maximizes data sharing among PEs because all PEs are processing the same part of IFM with different kernels. The main differentiation compared to most of the previously proposed CNN accelerators is Argus's capability to process CNNs that are compressed by a carefully tailored pruning algorithm, which maximizes and balances the utilization of available hardware resources on FPGAs. Compression algorithm clusters similar kernels into groups that have non-zero weights located at the same positions, reducing the skipping logic by cluster size. Furthermore, individual kernels are pruned in a structured manner. To reduce hardware requirements and to evenly distribute computations through PEs, Argus does not skip zeros in IFMs. Zeros in IFMs usually have a highly irregular distribution, which requires additional hardware for balancing the workload between PEs. In addition, Argus base architecture can be easily scaled to a more powerful version by stacking multiple PE modules with a proportional increase in terms of hardware cost. In summary, this work makes the following contributions:

1. Clustering of similar kernels into groups of kernels, which will have non-zero weights located at the same positions. Clustering reduces zero-skipping logic by a factor of 2 and it is independent of the underlining pruning method. Furthermore, it reduces on-chip memory used for storing non-zero weight positions.
2. Improvement of the existing Accelerator-aware pruning algorithm [21], which reduces zero-skipping hardware blocks of the original algorithm by an additional factor of 2. While the base algorithm takes into consideration only the weight magnitude for the decision, which weight to prune, the proposed CNN pruning algorithm also accounts for the LUT size to further constraint the pruning process.
3. Development of a complete accelerator that supports the developed CNN pruning algorithm. To the best of our knowledge, Argus achieves state-of-the-art performance density among FPGA accelerators in terms of GOP/s/DSP and GOP/s/BRAM, while being competitive with the current state-of-the-art considering GOP/s/LUT.

Argus is not the first CNN accelerator that benefits from processing sparse CNNs. Some of the previous works that benefit from sparsity in IFM are NullHop [15] and DaDianNao [16]. Similar to Argus, SparseNN [17] and Cambricon-x [18] take advantage of skipping zeros in CNN weights. Beside mentioned, there are many other high-quality architectures in terms of performance, like Eyeriss v2 [12], ENVISION [18], Thinker [19], UNPU [20], Snowflake [22], Caffeine [23], CoNNa [24], and architectures in [25]–[27].

II. FPGA-AWARE PRUNING ALGORITHM

Let us start by introducing the terminology that will be used in the remainder of this paper. Every layer's input 3D tensor will be called the "input feature map" (IFM), while every output of a layer will be called the "output feature map" (OFM). The IFM bundle designates a local region of IFM with a size of NxMxD that is used for one convolution or pooling computation. IFM bundle is composed of several IFM sticks, as illustrated in Fig. 1.

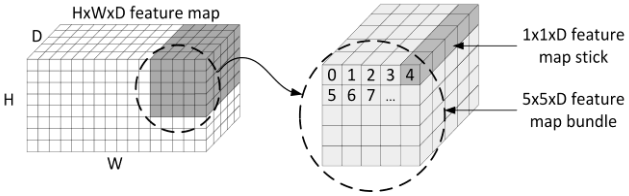

Fig. 1. Illustration of the feature map, feature map bundle, and stick.

The number of network parameters, together with large intermediate tensors (IFM/OFM) and the required number of MAC operations, generate high computational and memory cost of CNN processing. Authors of Eyeriss [12] state that their accelerator expects at least 25 GB/s of memory bandwidth while using 384 MACs to tackle computational complexity. One way of reducing CNN computing and memory requirements is to use CNN pruning (also known as network compression). CNN pruning procedures can be divided into two groups:

– *Fine-grain* approaches, where the algorithm decides which parameter is redundant at the granularity of a single parameter (weight) in each kernel. Han, Mao, and Dally [28] demonstrate a massive reduction in terms of used parameters of up to 9 times for AlexNet using this pruning approach. The fine-grained pruning approach usually results in high, but irregular sparsity. It is very difficult to take advantage of this kind of sparsity with the reasonable cost in terms of additional hardware used for balancing workloads between MACs and zero-skipping logic.

– *Coarse-grain* approaches, in which the pruning algorithm removes complete kernels [29]. This type of pruning does not introduce irregular sparsity patterns in the convolutional layers, which is a big advantage over fine-grain pruning. Almost every CNN accelerator benefits from this approach. The disadvantage is that coarse-grained pruning algorithms cannot achieve the

pruning levels of fine-grain pruning approaches.

Introducing regularity in fine-grain pruning overcomes the main disadvantages of complex zero-skipping patterns compared to coarse-grain while retaining high pruning factors. Argus applies two optimization techniques to reduce hardware requirements and to balance the workloads between PEs. First is kernel clustering, which reduces the complexity of logic by using only one zero-skipping block for the whole cluster of PEs instead of one per PE. One way of kernel clustering is presented in Cambicon-S [30], but the idea was not widely used, especially in synergy with other pruning techniques. The idea of clustering is to group kernels/neurons inside convolutional/fully-connected layers into clusters by the criteria of similarity and to prune all kernels in a cluster in the same way. The pruning outcome is shown in Fig. 2. The output of this pruning will be a sparse CNN, which has clusters of kernels with the same positions of non-zero weights within every cluster. Please notice that the positions of non-zero weights can differ between clusters. Because of this property, the underlining accelerator can use one zero-skipping module for the entire cluster of PEs instead of one module per one PE. The size of the cluster determines the reduction factor of the logic used for skipping zero multiplications.

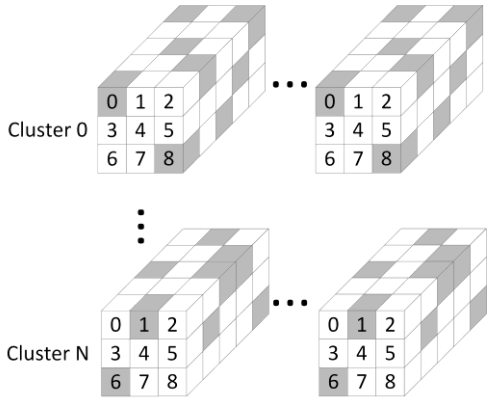

Fig. 2. Clusters of kernels in one layer. Grey represents the remaining non-zero weights.

Algorithm 1 presents the proposed kernel clustering approach and reordering of kernels within cluster groups. In the beginning, the *cluster_and_reorder_CNN* algorithm goes through a CNN model creating clusters for each convolutional and fully-connected CNN layer. For each layer, it calls *cluster_layer* function, which returns clusters for the current layer. *Cluster_layer* takes the weights tensor and creates a kernel similarity matrix (*sim*). The dot product is used as a measure of similarity between two kernels. After the similarity matrix is created, it is passed to the iterative Kerninghan-Lin (KL) clustering algorithm. Because of KL algorithm definition, after the first iteration, the kernels are divided into two groups with an equal number of kernels. For example, if the layer contains 16 kernels, after the first iteration of the KL algorithm, two clusters, each containing eight kernels, will be returned. In the second step, the KL algorithm is applied to these two clusters of eight kernels to further partition kernels into four clusters each having four kernels. After the final step of the KL algorithm, the output will be eight clusters of two kernels each.

After clusters are created, *cluster_and_reorder_CNN* function reorders kernels and channels inside convolutional/fully-connected and batch normalization layers. Reordering CNN model kernels is illustrated in Fig. 3, showing two layers with eight kernels. Imagine that clustering of Layer 0 returns four clusters: [3, 6], [2, 7], [0, 1], and [4, 5]. Before CNN model is deployed to the accelerator, the kernels inside each convolutional layer must be reordered in accordance with the computed clusters. Reordering of kernels will cause different processing order for OFM channels at the output of the accelerator (the orange arrow represents OFM channel stream in Fig. 3). To avoid on-line reordering inside FPGA, the kernel channels of the successor layer need to be reordered in the same way as the kernels are reordered in the predecessor layer (Fig. 3, right).

Algorithm 1. Kernel clustering.

```
func cluster_and_reorder_CNN(CNN_model)
  for layer in CNN_model:
    clusters[layer] = cluster_layer(CNN_model,
                                    layer.name)
  for layer in CNN_model:
    conv_pred = find_pred(CNN_model,layer)
    if(layer.type == Conv):
      reor_kernels(cluster[layer])
      reor_channels(cluster[conv_pred])
    if(layer.type == BatchNorm):
      reor_channels(cluster[conv_pred])

func cluster_layer(CNN_model,layer_name)
  weight_tensor =
get_tensor(CNN_model,layer_name)
  kernel_num = length(weight_tensor)
  for i in range(0, kernel_num):
    for j in range(i+1, kernel_num):
      sim[i][j] = dot_product(weight_tensor[i],
                    weight_tensor[j])
  clusters = Iter_Kerninghan_Lin(sim)
  return clusters
```

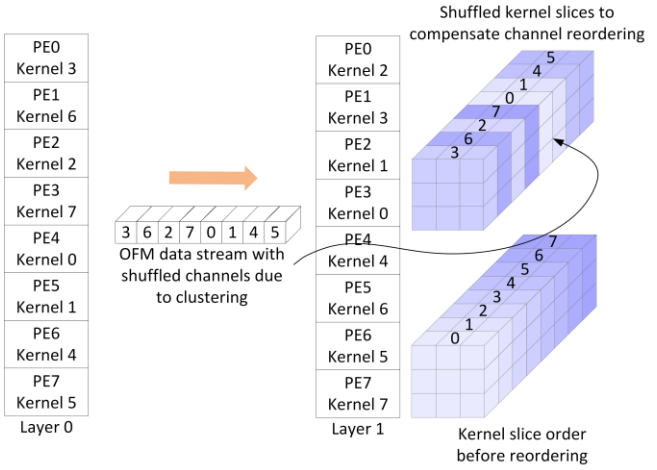

Fig. 3. Kernel channel reordering due to clustering.

In other words, the kernels of the first convolutional layer are reordered in the way in which they are clustered. Successor layers will get reordered OFM, so their channels need to be reordered in the same way as the first layer's kernels.

To further increase regularity (between clusters), Argus uses a modified version of Accelerator-aware pruning

algorithm proposed by Kang in [21]. Accelerator-aware pruning belongs to the fine-grained group of pruning algorithms. It solves the problem of irregular sparsity, which is the major drawback of most fine-grained pruning algorithms. However, it is not optimized for FPGA implementation. The authors did not take into consideration particular characteristics of FPGA resources, namely, LUTs, which will be used to implement zero-skipping logic. Argus modification to the original algorithm [21] takes into account LUT characteristics and further constrains the positions of non-zero weights regarding the available LUT size. As a result, the skipping logic is reduced by half when compared with the algorithm proposed in [21], while CNN accuracy is not degraded. The basic idea of Kang's pruning algorithm is to split the kernel weights into groups with an equal number of weights and then set the same amount of the smallest weights to zero in all groups, as shown in Fig. 4.

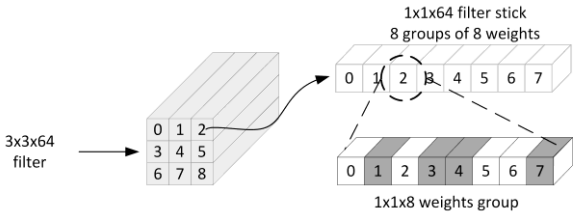

Fig. 4. Accelerator-aware pruning. Group size equals eight, the remaining non-zero weights equal four.

This pruning scheme ensures that every group has the same computational cost. Furthermore, it simplifies the hardware architecture mainly due to a balanced workload on all MAC units. Besides a balanced workload, this pruning approach cuts down the complexity of zero-skipping logic.

Although Kang's approach reduces the complexity for ASICs, it can be seen that the proposed pruning factors and group size are not optimized for FPGAs. The main reason for this is the difference in the granularity of combinatorial logic building blocks between ASIC and FPGA. In ASIC, logic is mapped into a network of individual gates, while in FPGA, the user logic is being mapped into LUTs. Please notice that LUT's level of granularity is much higher compared to gates. This results in step increments of logic utilization, when implementing user logic of increasing complexity. For example, a multiplexer that is mapped into a 6-input LUT will occupy one LUT as long as it has four or fewer data inputs. Once the number of data inputs is increased to five, the multiplexer will be mapped into 2 LUTs. The proposed CNN pruning algorithm minimizes this step increment in skipping logic (multiplexers) by further constraining Kang's pruning scheme. The result of applying additional constraints during pruning is a further reduction of skipping logic by half, compared to the original pruning scheme proposed in [21].

One of the proposed pruning patterns by Kang [21] sets the group size to eight and the number of non-zero weights to four. Further analysis of this pattern has shown that each of the four non-zero weights can be placed on one of the five possible places in a group of eight consecutive weights, as shown in Fig. 5(a). Please observe that the left-most non-zero weight can be located only at positions from 0 to 4 because in the worst case the three remaining non-zero weights must be located at positions 5, 6, and 7. The same applies to all other positions. It can be seen that using Kang's pruning pattern, the zero-skipping logic for one PE unit will be created out of four 5-to-1 multiplexers. Each multiplexer will be responsible for fetching one IFM point that will be multiplied by the associated non-zero weight. Note that in our case, every IFM point is represented by 16 bits. Using the previous example and assuming 6-input LUTs, each cluster of PEs will require IFM multiplexing logic utilizing 128 LUTs. This is because the multiplexing logic is composed of 4 multiplexers where each requires 32 LUTs. CNN accelerator, which has 32 cluster units, would utilize 4096 LUTs for this purpose only. Note that the majority of modern FPGAs have 6-input LUT as the core building block of the programmable logic.

To reduce the high utilization of LUTs, additional constraints can be applied to allowable non-zero positions. As shown in Fig. 5(b), four, instead of five different positions, for each remaining non-zero weight could be permitted. This reduces the multiplexer size to 4-to-1, which leads to a saving of 64 LUTs per PE cluster. In other words, using the same example with 32 clusters, this additional constraining will reduce zero-skipping logic resources from 4096 to 2048 LUTs. Please notice that even when using these additional constraints during CNN pruning, it is still possible to regain most of the accuracy of the unpruned CNN, as can be seen in Table I. Furthermore, the proposed pruning pattern will also be beneficial when implementing skipping logic in ASIC also, but to a slightly lesser degree, reducing the required number of logic gates by about 20 %.

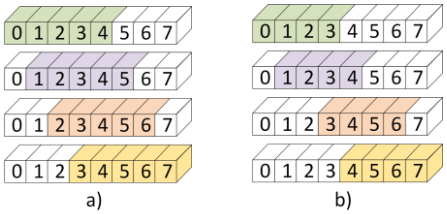

Fig. 5. Possible positions of non-zero weights in a group: a) original pruning proposed in [21] (128 LUTs per IFM selector), b) constrained (64 LUTs per IFM selector).

TABLE I. PRUNING RESULTS FOR LARGE AND COMPACT NETWORKS.

CNN	Unpruned Top-5 accuracy	Constrained (Fig. 3(b))
ResNet50	92.1	92.1
VGG-16	90.1	89.8
MobileNet v1 224 1.0	89.5	89.0

The pseudo-code of the proposed FPGA-aware pruning algorithm is shown in Algorithm 2. At the beginning of the pruning process, CNN's performance is evaluated and stored in the *initial_accuracy* variable. Next, kernels are clustered using the *cluster_and_reorder_CNN* algorithm. The pruning process starts by dividing the kernels into sticks. Every stick is further divided into several groups, each group being eight weights large, as shown in Fig. 4, by calling *split_krns_into_groups* function. The actual pruning

is performed in four steps, removing one weight at a time from a group of eight (incremental pruning). In each step, for every weight group, a list containing the optimal weight pruning order is created by calling the *create_pruning_order_list* function. Creating an optimal weight pruning order is a three-stage process. First, the weights in weight groups are normalized separately for each kernel within the cluster. Next, the absolute values of the normalized kernels are added and stored in the temporary matrix, which has the same shape as every kernel in a cluster. Finally, every group in the temporary matrix is sorted in ascending order and a list of indices inside the groups is returned.

Returned order of indexes will be considered first for pruning, as is the case in the majority of previously proposed pruning algorithms. However, due to additional constraints imposed on the allowable non-zero weight positions, as shown in Fig. 5(b), this will not be always possible. For example, let us assume that in the first two pruning steps the weights at positions zero and one have been pruned. This will prohibit the removal of the weight at position two in the following steps. Allowable weights for pruning in steps three and four, in this case, would be the weights at positions 3–7, but not the weight at position two. Selection of the best possible weight to prune next, while obeying the constraints from Fig. 5(b), is performed within the *set_to_zero* function.

Algorithm 2. FPGA-aware Network Pruning.

```
func compress_cnn(cnn_model, cluster_size):
 initial_accuracy = evaluate_network(cnn_model);
 cluster_and_reorder_CNN(CNN_model);
 kernel_groups=split_krns_into_groups(cnn_model;
 for i in range(4):
  for group in kernel_groups
    pruning_list =
      create_pruning_order_list(group,
clusters);
    set_to_zero(group, pruning_list);
  retrain pruned CNN
```

Kernel group size was selected to be eight-weights large because most of FPGA SoCs limit the width of the Advanced Extensible Interface (AXI) data bus between the DRAM controller and the programmable logic to 128 bits. Since Argus uses 16-bit operand number representation, because of its negligible impact on CNN accuracy [31], [32], at most eight operands can be transferred in one beat of AXI transaction, so selecting a kernel group size of eight would result in the optimal processing performance.

To evaluate the proposed FPGA-aware pruning algorithm, it was used to prune several standard CNN networks pretrained on ImageNet [33], using Keras [34]. Reported accuracy results after pruning were obtained using the validation set. Note that in the performed experiments, the first convolutional layer from every selected CNN network was excluded from pruning due to its small depth of only three IFM points, which seems to be the common approach [21].

As can be seen in Table I, FPGA-aware pruning algorithm results in a negligible loss of pruned network accuracy in the case of compact networks like MobileNet and VGG-16, and no loss in the case of ResNet50. It is worth noting that most hardware architectures used for comparison with Argus use 8-bit precision arithmetic, which almost always degrades CNN accuracy more than in the case of pruned MobileNet v1 [35].

III. ARGUS CNN ACCELERATOR ARCHITECTURE

The most demanding layers in CNNs considering computational time are convolutional. They consume up to 90 % of the time needed for inference [4, 8]. Therefore, the accelerator performance is the most dependent on its efficiency in the processing of convolutional layers. The process of computing a generic convolutional layer is listed in Algorithm 3.

Note that IFM padding and optional bias addition were omitted from Algorithm 3. Time for bias addition can be masked, while padding does not consume additional time because the number of convolutions to compute is determined by OFM size (loops L2 and L3), not by IFM size. For simplicity, convolutional layer processing is split into two functions, *calc_layer_ofm* and *calc_ofm_point*.

Algorithm 3. Pseudo-code of generic convolutional layer processing algorithm.

```
func calc_layer_ofm(IFM, KM):
L1:for (fn = 0; fn<Kernel_Num; fn++)
L2: for (y = 0; y <OFM_Height; y++)
L3:  for (x = 0; x <OFM_Width; x++)
       ifm_h_part = y*Sv:y*Sv+Kernel_Height
       ifm_w_part = x*Sv:x*Sv+Kernel_Width
       ifm_bundle =
IFM[ifm_h_part][ifm_w_part][:]
       OFM[x][y][fn] =
       calc_ofm_point(ifm_bundle, KM[fn])

func calc_ofm_point(ifm_bundle, km):
L4:for (kh = 0; kh<Kernel_Height; kh++)
L5: for (kw = 0; kw<Kernel_Width; kw++)
L6:  for (kd = 0; kd<Kernel_Depth; kd++)
       ofm_point += ifm_bundle[kh][kw][kd]*
                          km[kh][kw][kd]
   return ofm_point
```

Calc_layer_ofm takes IFM 3D tensor and kernels (KM) as input and process IFM by sliding kernels over it. Its task is to prepare the IFM bundle needed for the current OFM point calculation and to call *calc_ofm_point*. The number of convolutions per channel is determined by OFM horizontal and vertical size, which is represented by the L2 and L3 loops. Loop L1 is responsible for creating the depth of OFM. *Calc_ofm_point* takes *ifm_bundle* for the current OFM point and kernel as input and returns a dot product of these 3D tensors. Loops L4 and L5 determine the vertical and horizontal stick coordinates in the kernel. L6 goes through IFM by the channel axis until *Kernel_Depth* is reached. Note that *Kernel_Depth* is equal to IFM depth in all, but Depthwise convolutions [10].

As opposed to architectures that rely on a 2D array of PEs to compute a single convolution, Argus dedicates one PE to calculate all convolutions related to a particular kernel. In other words, one PE is responsible for computing one channel of OFM. Because of kernel clustering, every two (cluster size) adjacent PEs share the same skipping logic. Speaking in terms of Algorithm 3, one cluster of PEs is responsible for executing two *calc_ofm_point* function calls. Note that the hardware implementation of every PE will

have dedicated memory for storing kernel weights. These weights will be stored as an array created by flattening the kernel in stick-first order. Flattening removes any information regarding the kernel shape, which means that the kernel can be of any shape, which is also an important advantage of Argus over many existing solutions. To increase the processing performance, Argus unrolls the L1 loop with a factor of *PE_Num*, the number of available PEs. This means that the Argus is processing *PE_Num* output channels of OFM in parallel. To further speed up processing, Argus also does the unrolling of loop L6 by a factor of four, which means that every PE is capable to execute four MAC operations in a single clock cycle. If a network is compressed, these four MACs are covering all non-zero multiplications inside a group of eight consecutive IFM points, as shown in Fig. 4. When CNN is not compressed, PE takes four consecutive points of IFM, because there are no zero weights that can be skipped. That means that the proposed pruning speeds up the processing by a factor of two in the ideal case. Note that just non-zero kernel weights are stored inside PE memory if the network is compressed.

To achieve high utilization of PE units, selecting the value of *PE_Num* must be done carefully. The vast majority of layers in contemporary CNNs have at least 32 different kernels. Setting *PE_Num* to 32 will lead to high PE efficiency for all layers that have 32 or more kernels. Of course, a higher number of PEs would increase the parallelism and therefore further increase the processing performance of layers with more than 32 kernels, but the hardware will be underutilized while processing layers that have fewer kernels than *PE_Num*. To solve this underutilization problem, Argus uses several groups of PEs, each having 32 PEs, called "Convolutional Cores".

The top-level block diagram of the generic Argus architecture is shown in Fig. 6. Argus is designed as a configurable and scalable heterogeneous multi-core architecture.

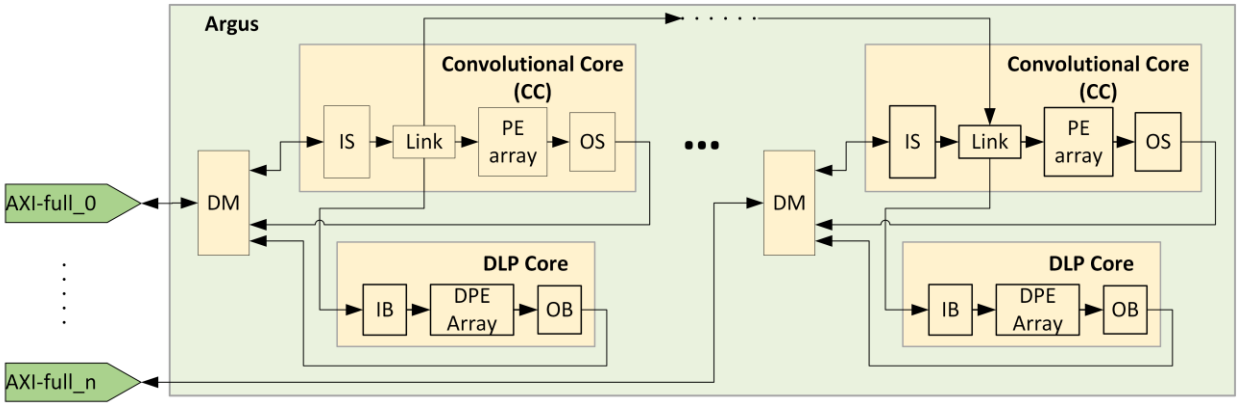

Fig. 6. Top Level Architecture of the Argus CNN Accelerator.

At the top level, Argus is composed of two major components: Convolutional Cores (CCs) and DLP Cores. Besides them, several Data Mover (DM) modules are used to connect CCs and DLPs to the surrounding logic. DMs convert and combine the internal AXI-Stream interfaces, used by CCs and DLPs, into a number of AXI-Full interfaces, which are used to connect the Argus core to the DRAM memory controller. CCs are used to accelerate convolutional and fully-connected layer types from CNN, which can be compressed using the "FPGA aware pruning" algorithm. Please notice that the fully-connected layer type can be regarded as a special version of the convolution layer, where the kernel size equals the IFM size of the fully-connected layer. CCs are specifically designed to operate efficiently on convolutional layers and are therefore ill-suited to be used to accelerate other CNN layer types, like pooling, adding, etc. The purpose of DLP cores is to accelerate the processing of non-convolutional layer types.

Argus architecture is highly configurable, enabling easy creation of different configurations, depending on the selected number of CC and DLP cores, with different performance/area/power tradeoffs. Before the actual implementation, the user can specify the desired number of CC, as well as DLP cores.

CC module, shown in Fig. 7, is composed of Register file, DRAM Arbiter, Input Stream (IS), Link, PE array, and Output Stream (OS) modules. After configuring the core using Register file, IS requests biases, weights, and non-zero indexes through a DRAM Arbiter. When the weights are loaded into the PE array, IS starts streaming IFM while the PE array does the computation. OS is responsible for storing the computed convolutions into the DRAM memory via the associated DM module.

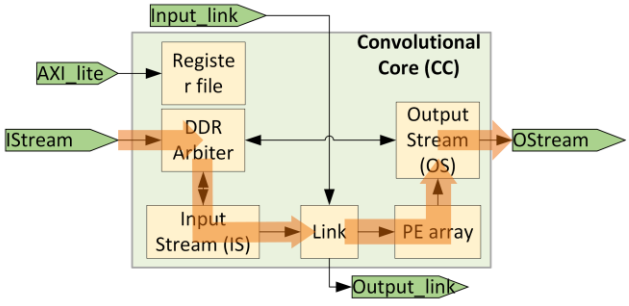

Fig. 7. Single Convolutional Core (CC) top-level architecture.

A. Input Stream

Input Stream plays a major role in reducing data transfer between DRAM and CC. The previously published idea [36] was exploited by Argus IS. IS generates read requests for IFM sticks and stores them in the on-chip cache, which is a part of IS. Reading starts from the upper left stick and continues through the first row of IFM as shown in Fig.

8(a). After *Kernel_Height* (KH) rows are stored, IS starts sending IFM bundles to PE array which computes the first row of OFM, Fig. 8(a) (KH and KW equal three). Meanwhile, IS continues requesting IFM stick data for row number 3.

As can be seen in Fig. 8(a), there is an opportunity for significant data reuse while processing IFM by up to 9 times for kernel size 3×3 with vertical and horizontal stride values of one [37]. The first bundle includes nine IFM sticks from the upper-left corner. These sticks contain the first three sticks from rows 0–2 of IFM. After *PE_Num* OFM points are computed, IS slides over the IFM by moving one place to the right, assuming that the horizontal stride equals one. The second bundle now contains sticks from columns 1–3 in rows 0–2. Note that this second IFM bundle reuses six IFM sticks from the previous IFM bundle (six sticks from columns 1 and 2). The third IFM bundle (dark grey bundle in Fig. 8(a)) reuses sticks from the second column for the third time. After sliding down by one row, row 0 is not needed anymore and can be replaced by row 3 from the IFM.

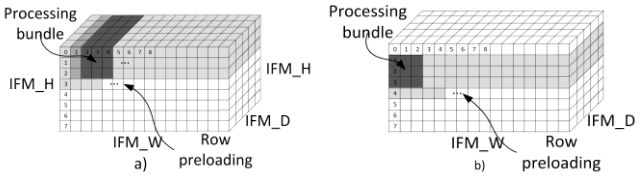

Fig. 8. Processing of IFM by bundles (the grey part of IFM): a) Computing the first OFM row while preloading row 3 into IS cache; b) Loading pattern when the cache size is less than needed to store the whole IFM.

To avoid restrictions on IFM size that can fit into the cache, IS can split IFM vertically into several parts, as shown in Fig. 8(b). Partitioning of IFM along the width axis, known as striping [36], allows setting the cache size according to the available on-chip memory resources rather than according to the IFM size. Please notice that when using striping, some of the sticks on the vertical boundary will be loaded twice, but this will not cause a significant increase in the required throughput because just a few columns will be loaded more than once.

As can be seen in Fig. 9, IS consists of 4 main blocks. The *Stick requester* generates the stick address in DRAM based on information about the IFM position in DRAM. Data from DRAM (response on request) goes through *Cache writer*, whose responsibility is to calculate the stick cache address and write it in the cache. Addresses are created based on a request pattern that is known in advance.

The *Memory* module is built around two-port RAM, with the addition of a valid row status, which indicates which row of the cache is valid and which is free for new sticks. This status line is used by both *Cache writer* and *Cache reader*. If there is no free space in *Memory*, the *Cache writer* will block the DRAM controller by pulling down the ready signal. On the other hand, The *Cache reader* will stop the IFM stick readout process if the requested stick is not yet in *Memory*. *Cache reader*, as the most complex module in IS, is responsible for generating the correct read address of the stick in the cache memory. Besides mentioned, *Cache reader* has information about padding, so it can request zero padding at appropriate moments.

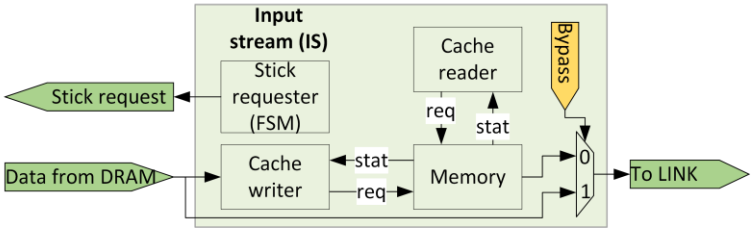

Fig. 9. Input Stream block diagram.

B. DRAM Arbiter and Output Stream

DRAM Arbiter, as shown in Fig. 7, is responsible for the arbitration of read requests to DRAM. Read requests are coming mainly from IS and sometimes from the OS module. IS requests sticks from IFM whenever the internal cache in IS is ready to store a new stick. OS creates requests only when CC is computing partial convolutions, which is the case when CC cannot process a complete convolutional filter in a single pass. Because of the limited on-chip memory resources, CC can split the filter into two or more parts along a channel axis. In the first pass, CC will calculate the first part of the convolution and store it off-chip. Next, CC will load the second slice of each filter and process the rest of the IFM. These two parts have to be added together to compute the final convolution result. To do that, OS pulls the first partial convolution results part through *DRAM Arbiter* and adds them to the second partial convolution results delivered by the PE array. This way, CC masks the time needed for the partial results addition operation to avoid the performance penalty when doing partial convolutions.

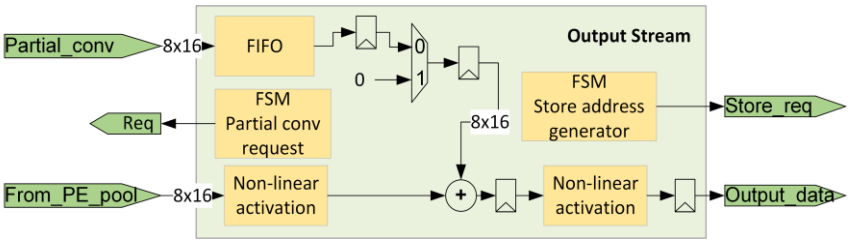

Fig. 10. Output stream block diagram.

The *Output stream* (OS) module, shown in Fig. 10, takes the convolution results from the PE array and passes them to the DRAM memory controller. It creates AXI requests with the appropriate physical address of the OFM stick and transfer size in bytes. In addition, it implements a mechanism for partial convolution completion with a dedicated FSM for requests and an additional adder in the data path for addition. Besides partial convolution, an adder could also be used for adding shortcut connections at the end of each residual block in ResNet networks.

C. Processing Element Array

PE array is the biggest module of CC, composed of 32 PEs grouped into 16 clusters. The internal architecture of the PE array is presented in Fig. 11. PE array uses two data streams, the Input stream, and the Output stream. Both streams use the 128-bit AXI-Stream protocol. The input stream is used for loading bias, weights, non-zero index, and IFM. The output stream is responsible only for moving the convolution results to OS.

The processing sequence starts with bias loading into the Bias Storage module, which is a simple register bank of 32 registers, one per PE. When all biases are loaded, IS delivers weights and non-zero indexes to the *Memory bank*. The *Memory bank* is built of 32 Block RAMs (BRAM) for weight storage. Each BRAM is allocated to one PE, storing 2048 weight values. Alongside BRAMs for weights, there are 4 additional BRAMs for non-zero index values. After the weights are loaded, IFM starts streaming through Input stream to all IFM point selectors (zero-skipping block). Every IFM point selector has four 4-to-1 multiplexers for choosing IFM points that match positions of non-zero weights in a group of 8, as described in Section II ("FPGA-aware pruning algorithm"). The size of the IFM point selector is reduced by using the constraints shown in Fig. 5(b) to only 64 LUTs per selector, 16 LUTs per multiplexer. Note that one IFM point selector is used per cluster, meaning that only one zero-skipping block is used per two PEs.

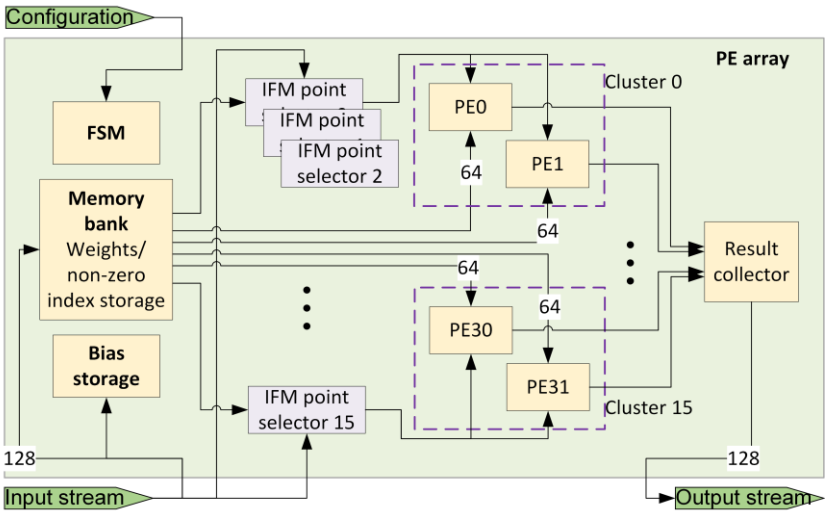

Fig. 11. Architecture of Processing Element array.

All computations in the *PE array* are done in 32 PEs. Every PE is built around 2 DSP blocks that are capable of computing 4 MAC operations in one system clock cycle. To achieve 2 MAC operations per single DSP, a Multi-Pumping technique [38], [39] has been used.

The *Result collector* is the last block in the *PE array* pipeline, which collects results from all PEs. It is capable of processing eight values in a single clock cycle and it is taking results from PEs in a Round-Robin manner, in blocks of eight.

D. Multi-Core Convolutional Engine

Even though a single CC has high MAC utilization over almost all existing CNN architectures, the selected number of PEs (32) can be a limiting factor for achieving the required performance for more complex CNNs. CCs peak performance is 32 GMAC/s, which can effectively be seen as 64 GMAC/s if the network is compressed.

If more performance is needed, the number of used PEs, *PE_Num*, must be increased. To scale up the Argus performance without degrading PE utilization on shallow layers, CC/IS pairs can be replicated several times, as shown in Fig. 12.

In this setup, every CC will have a dedicated IS, which means that every CC can operate on different parts of shallow IFMs, keeping the PE utilization high. On the other hand, while processing layers that have more than *PE_Num* kernels, there is no need to supply every CC with a different IFM part, instead all PEs can now process the same IFM bundle. In this case, only one IS will fetch the IFM from the DRAM and pass the IFM bundles through blocks called *Link* to other CCs, while the other IS modules would be idle.

As an example, let us consider an Argus core composed of four CCs with a total of 128 PEs, as shown in Fig. 12. Also, let us assume that the convolutional layer that is being processed has 64 kernels.

To achieve the maximum utilization of PEs, CC_0 uses 32 kernels that belong to one group of 16 clusters, and CC_1 uses the remaining 32 kernels (belonging to the other 16 clusters). To employ CC_2 and CC_3 modules, Argus also loads the first group of clusters into CC_2 and the second cluster group into CC_3. Finally, CC_0 will have the same copy of kernels as CC_2, and CC_1 will have equal memory

content as CC_3. IS_0 loads the upper half of the IFM and broadcasts it also to CC_1 using the *Link* module (see orange arrows in Fig. 12). This means that CC_0 and CC_1 are computing the upper half of OFM using the same IFM bundles. A similar approach is used with CC_2 and CC_3, but now IS_2 is responsible for streaming the lower half of the IFM to CC_2 and CC_3 modules. Please notice that IS_1 and IS_3 modules are idle, thus reducing DRAM throughput.

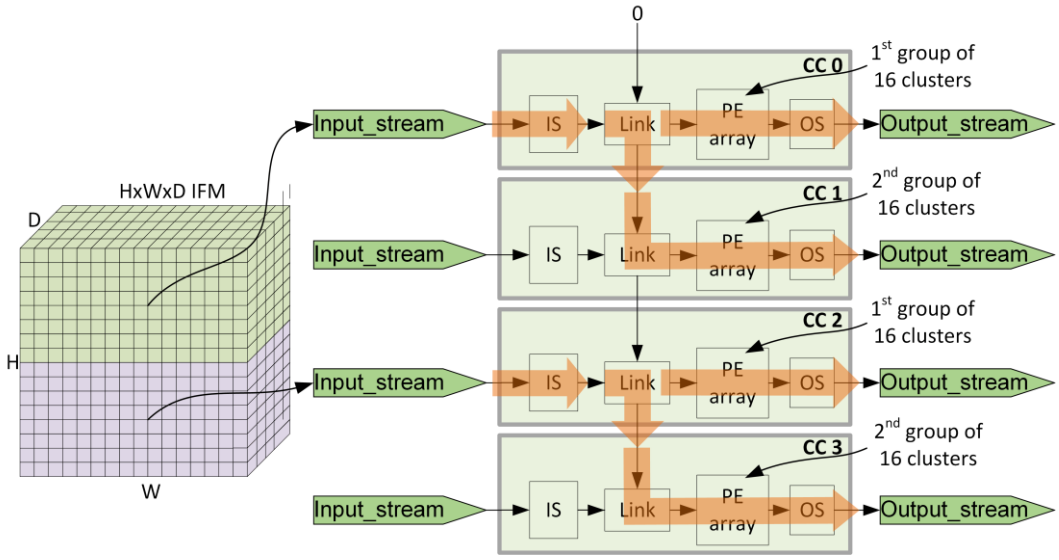

Fig. 12. Scaling up the performance of the accelerator by stacking more CC and connecting them using Link modules.

E. Dense Layer Processing Core

The purpose of the *Dense Layer Processing Core* (DLP) core is to enable Argus to process maximum pooling, average pooling, and adding layer types. The DLP has six pipeline stages, which are controlled by FSM (Fig. 13). The DLP core can work in three different configurations, where some of the stages could be skipped, depending on the layer type. The processing of layers always has three steps. In the first step, the first IFM stick is sent to the *Memory* module through the *Input regs array* block. In the second step, all remaining sticks from the IFM bundle are being processed, storing intermediate results in the *Memory*. When all sticks are processed, in the third step, the final results are sent from the *Memory* module to the output of the DLP core.

Each DLP pipeline stage is vectorized, consisting of eight lanes of identical processing elements, enabling the DLP core to process eight IFM points in parallel. All stages operate on 16-bit numbers, except the *Memory* and *Mul array* modules, which use 24-bit operands. During the calculation in the first step, which is common to all supported layer types, only *Input regs array* and *Memory* pipeline stages are active. The purpose of this phase is to initialize the content of the *Memory* with the values from the first IFM stick.

When processing a maximum pooling layer, during the second step, the first four pipeline stages are active. The *Memory* module stores the current maximum value of each OFM point from the current OFM. The *Add array* stage calculates the difference between a vector of eight consecutive IFM points and the corresponding current maximum OFM vector stored in the *Memory*. Based on this comparison, the *Cmp Mux array* stage updates the content *Memory* module with the appropriate maximum OFM vector. In the third step, the *Memory* and the *Output regs array* modules are active, sending the final maximum pooling results from the *Memory* to the output of the DLP core.

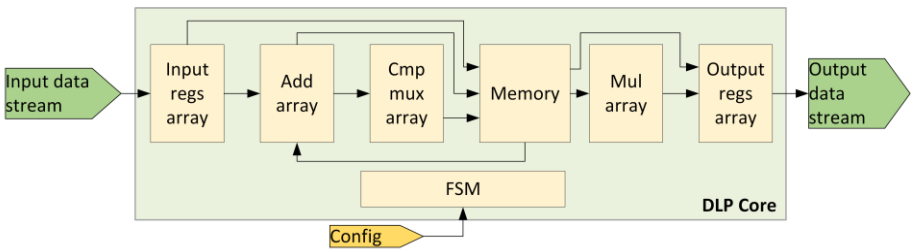

Fig. 13. Architecture of the Dense Layer Processing Core.

In the case of average pooling layer processing, the first step is identical to that of the maximum pooling layer. During the second step, the *Input regs array*, *Add array*, and *Memory* stages are active. The *Memory* module stores the running sum of every OFM point from the current OFM stick. The new IFM stick from the current IFM bundle, fed through the *Input reg array* stage, is added to the OFM running sum stick in the *Add array* stage. In the final step, the *Memory*, *Mul array* and *Output regs array* stages are active. The final OFM point sum is averaged by multiplying it with the value supplied as part of the DLP configuration. After the multiplication, the output of the *Mul array stage* is

the final vector of average values of eight consecutive OFM points, which is sent to the output of the DLP core using the *Output regs array* stage.

When processing an adding layer, the first step is identical to the first processing steps of pooling layers. In the second step, the *Input regs array*, *Add array*, and *Memory* stages are active. The *Memory* contains the running sum of the current OFM stick. *Add array* module accumulates the values of the IFM sticks from appropriate IFMs to this running sum OFM stick. When IFM sticks from all IFMs that are being added together are processed, the final sum value of the current OFM stick is stored in the *Memory* module. In the third step, the OFM stick is sent to the DLP's output.

IV. IMPLEMENTATION RESULTS

To show the trade-off between performance and hardware utilization, Argus was implemented in three different configurations. The most compact version has one CC and one DLP and can be fitted in a wide range of FPGA SoCs. A balanced version in terms of performance and required hardware resources has two instances of CC block and one DLP, showing almost doubled performance compared to the one-CC version. The most powerful version has four CCs and one DLP, and it is meant for mid-range SoCs. All three configurations have been implemented using Xilinx Vivado Suite 2019.1, targeting the ZU7 MPSoC device. The synthesis was performed using *Flow Perf Optimized High*, while *Performance Net Delay High* strategy was used for implementation. Resource utilization is shown in Table II, together with the utilization for some of the previously published accelerators that were used for comparison. Table III shows FPGA devices that can accommodate various accelerators in terms of available resources.

TABLE II. RESOURCE UTILIZATION OF ACCELERATORS USED FOR COMPARISON.

Accelerator	No. of MACs	LUT (FPGA)	BRAM (FPGA)	LUT/ DSP	Arith. (bits)
FpgaConvNet [(a)]	900	218K	615	242	16
Snowflake	256	-	192	-	16
NullHop (FPGA)	128	229K	386	1789	16
NEURAghe	864	88K	320	101	16
CoNNa C4	256	267K	596	1042	16
Caffeine	1058	100K	783	94	16
Depthwise optimized accelerator[(b)]	3283	121K	-	36.8	8
Eyeriss v2	384	-	-	-	8
Thinker	1024	-	-	-	8/16
Envision	512	-	-	-	4/8/16
[25][(a)]	220	13.4	98	61	8
[26][(a)]	1144	252	912	220	16
[27][(a)]	1350	178.1	1460	132	8/16
NVDLA	256	-	-	-	4/8/16
Argus 4CC	272	46K	218.5	168	16
Argus 2CC	140	25K	96.5	177	16-bit
Argus 1CC	74	14K	55	195	16-bit

Note: [(a)]Accelerators are scalable, meaning that utilization can be both lower and higher. Utilization numbers are presented for configurations that are used for performance comparison; [(b)]Specialized accelerator for CNN architectures that exploits Depthwise separable convolutions. All hardware requirements are reported assuming the implementation results presented in the paper [40].

Please notice that some accelerators are scalable, meaning that they can fit into smaller devices than shown in Table III. Both Table II and Table III report the utilization/fitment for the configurations of these accelerators used for performance comparison. FpgaConvNet and Caffeine are implemented using HLS approach, which is flexible but can be inefficient in terms of required hardware resources. Furthermore, HLS-based accelerators cannot support changing the CNN model on-the-fly, which is not the case with Argus. Argus is a general CNN accelerator independent of the CNN model. Both architectures use a big amount of MAC units (equivalent to DSP blocks in FPGA) and the proportionally large amount of available on-chip memory resources, which disqualifies them from being used in the entry-level FPGA SoCs. Besides being inefficient in mapping algorithms to the underlining hardware, HLS does not manage to use the complete computational potential of a DSP block.

An important parameter when comparing various accelerators, which can be skipped at first glance, is the utilization of LUTs per single DSP. Two extremes regarding this criterion are CoNNa C4 and NullHop. In the case of CoNNa, high efficiency and complex zero-skipping logic result in requiring more than 1000 LUTs per one DSP block. Speaking in terms of required DSP blocks, CoNNa can fit almost every FPGA device. On the other hand, high LUT utilization prevents it to be implemented in the entry-level FPGA devices. In addition, CoNNa can not utilize all available DSP blocks. The same is the case with the NullHop.

TABLE III. FPGA-BASED ACCELERATORS COMPARISON FOR DIFFERENT SOCS.

Accelerator	Required LUT/ DPS	Zynq UltraSCALE + MPSoC				
		Zu2	Zu3	Zu4	Zu5	Zu7
Available LUT/DSP		196	196	120	93	133
[25]	60	•	•	•	•	•
Caffeine	94					•
Snowflake[(a)]	-		•	•	•	•
NEURAghe	101				•	•
[27]	131					•[(b)]
Argus 4CC	168		•	•[(a)]	•[(a)]	•
Argus 2CC	177	•	•	•	•	•
Argus 1CC	195	•	•	•	•	•
[26]	220					•[(b)]
FpgaConvNet	242					•[(b)]
CoNNa (C4)	1042					•
NullHop	1789					•

Note: [(a)]Only required DSP and BRAMs are used for fitment comparison because LUT utilization is not reported; [(b)]Accelerators are scalable, meaning that they can fit smaller devices. Table III shows the fitment for the configurations that are used for performance comparison.

On the other hand, accelerator presented in [25] requires the smallest number of LUTs per DSP. However, it does not exploit any type of sparsity while processing CNNs, which

degrades its performance per used DSP block. When the accelerator is scaled up to the limit of the underlining device (all DSP blocks are used), many LUTs remain unused. Argus utilizes these LUTs to perform zero-skipping, decreasing the number of MAC operations performed by DSP blocks, which leads to better GOP/s/DSP by up to 3.3 times.

NEURAghe [42] has an excellent LUT per DSP ratio, but it requires a large number of DSPs and BRAMs, which means that only bigger devices can support it. On the other hand, Argus shows a very good overall LUT/DSP ratio, which is mostly due to the carefully tailored pruning algorithm and omitting to skip zeros in the IFM. As can be seen from Table III, Argus containing one or two CCs can fit into all Zynq UltraSCALE + MPSoC FPGA devices, while the most powerful version cannot be implemented only with the smallest SoCs, ZU2. In contrast with most other architectures, Argus can fit in the largest number of FPGAs, while retaining the best performance among FPGA-based accelerators.

As stated before, Caffeine, CoNNa, and Accelerator architectures in [26] and [27] can only fit into the ZU7 device, which is the largest device used for comparison. NEURAghe can be mapped into smaller devices like ZU5, but still not into cost-optimized SoCs like ZU3. Snowflake shows similar results as the biggest Argus instance, but it should be noted that the LUT utilization is not reported for this architecture, so an FPGA device fitment was calculated considering only the required DSPs and BRAMs. Our analysis of the Snowflake architecture indicates that it should not require a large amount of LUTs.

V. PERFORMANCE RESULTS

Table IV shows a comparison of performance results when accelerating AlexNet, VGG16, MobileNet v1, and ResNet50 for several CNN accelerator architectures.

Performance analysis of AlexNet acceleration shows that the major performance degradation for Argus comes due to the time needed to process large fully-connected layers. These layers consume about 67 % of the inference time. Note that most of this time is spent on weight loading rather than performing calculation. To fully utilize DSP blocks when processing large fully-connected layers, all accelerators require extremely high bandwidth, sometimes more than 200 GB/s [22], which is not achievable in today's low-cost embedded devices. Furthermore, many papers omit the discussion on memory bandwidth requirements when processing these layers, some of them remove them from performance analysis, while others assume that CNN can be compressed [22].

TABLE IV. PERFORMANCE COMPARISON OVER DIFFERENT CNN ARCHITECTURES.

Accelerator	CNN Architecture	LUT (FPGA only)	No. of MAC's	Freq. (MHz)	Frame Rate [frames/s] Conv only	Frame Rate [frames/s]
FpgaConvNet	AlexNet	218K	900	125	-	121.53
NVDLA	AlexNet	-	256	250	-	68.6
Eyeriss v2	AlexNet	-	384	200	287.4[(a)]	234.1[(a)]
Thinker	AlexNet	-	1024	200	-	254.3
Envision	AlexNet	-	512	200	47	-
Snowflake	AlexNet	-	256	250	100.5	-
Argus (4CCs)	AlexNet	46K	272	250	254	82.8
FpgaConvNet	VGG16	218K	900	125	-	4.01
NullHop (FPGA)	VGG16	229K	128	60	-	0.44
NEURAghe	VGG16	88K	864	140	-	5.52
Caffeine	VGG16	100K	1058	200	-	8.67[(b)]
CoNNa	VGG16	267K	256	140	-	7.83
Argus (4CCs)	VGG16	46K	272	250	15.8	11.02
Eyeriss v2	MobileNet v1 (128×128, 0.5)	-	384	200	-	1117
Argus (4CCs)		46K	272	250	-	1090
Depthwise optimized accelerator [40]	MobileNet v1	121K	3283	150	-	231.2
Argus (4CCs)	MobileNet v1	46K	272	250	-	185
NVDLA [41]	ResNet50	-	256	250	-	17.45
NVDLA [41]	ResNet50	-	256	500	-	29.1
Snowflake	ResNet50	-	256	250	17.7	-
Argus (4CCs)	ResNet50	46K	272	250	40.2	36.5

Note: [(a)]Results are recalculated for a maximum throughput of 25 GB/s with a provided degradation of 24 % as the authors stated; [(b)]Performance is calculated using the stated overall throughput divided by 30 GOPS needed for VGG 16. The authors did not provide frames/s results, but we extract their counting rules on the total number of operations needed for VGG-16 from this paper.

To perform a fair comparison with the Snowflake and Envision, only the convolutional layer inference time was measured due to a lack of end-to-end inference results for these accelerators. Argus's performance is more than 5 and 2.5 times better than Envision and Snowflake, respectively. In the case of Snowflake, the performance gain is most probably caused by the fact that the Snowflake does not support zero-skipping. Compared with the FpgaConvNet, Argus is 30 % slower, but FpgaConvNet uses 3.5 and 4 times more DSP blocks and LUTs, respectively. Compared to Thinker, Argus is 3 times slower, but it uses 4 times less MAC units. In addition, the authors of Thinker provided performance results in terms of GOP/s only, without considering memory throughput as a limitation. Because of

that, it could be argued that Thinker's actual performance will be worse than that listed in Table IV. Compared to Eyeriss v2, Argus has a slightly worse performance when processing only convolutional layers, which is mostly due to the higher compression ratio used in Eyeriss v2 [12] paper. Even though the authors of Eyeriss v2 used almost 3.5 higher bandwidth to the DRAM and a higher compression ratio, on-chip buffering implemented within Argus managed to compensate for almost all of these advantages. However, there is no buffering technique that can compensate for the required throughput on the fully-connected layers if the batch size equals 1, which degrades Argus performance when processing the complete AlexNet CNN. Note that such large fully-connected layers are considered obsolete and are not used in modern CNN architectures.

In the case of VGG 16, Argus achieves better results than all other architectures. Besides being faster, Argus requires at least 4 times fewer LUTs compared to FpgaConvNet and NullHop (FPGA implementation), and about 2 and 1.8 times fewer LUTs than Caffeine and NEURAghe, respectively. Moreover, please observe that all other architectures, except NullHop, use from 3 to over 4 times more MAC units. Once more, all previously proposed accelerators require too much FPGA resources, which disqualify them from being used in entry-level FPGAs and edge devices.

Performance comparison on MobileNet v1 was done with Eyeriss v2 and Depthwise separable convolutional engine [40]. In contrast with AlexNet, MobileNet v1 is a modern CNN architecture that does not include large fully-connected layers. Argus shows similar performance as Eyeriss v2 on small MobileNet v1 (width multiplier equal to 0.5) while using at least 3.5 and 10 times lower memory throughput when processing Pointwise and Depthwise layers, respectively. On the other hand, Argus has a 20 % lower performance than the accelerator proposed by Zhao, Niu, and Luk [40], which is highly optimized for Depthwise convolutions. However, please notice that the proposed accelerator [40] uses 12 times more MAC units than Argus to reach the reported performance.

Argus performance for ResNet50 was compared against Snowflake and two different NVDLA [41] configurations. Same as for AlexNet, Snowflake presents results for convolutional layers only, and in this case, Argus is about 2.27 times faster. Compared to Nvidia's NVDLA, Argus is 2.09 faster than NVDLA running at 250 MHz and 1.25 times faster than the configuration that works at 500 MHz.

Besides absolute performance comparison (frames per second), a very important aspect for FPGAs and scalable accelerators is the performance per resource used. Moreover, it is important to develop an accelerator that uses the available resources in a balanced manner. Balanced usage of resources has a great impact on accelerator scaling and utilization of available processing resources on a dedicated FPGA platform. Table V shows the performance comparison among different FPGA accelerators for VGG16 CNN. Besides GOP/s per DSP and LUT, which are commonly used metrics, Table V also lists GOP/s per BRAM used, which represents 36 kb of on-chip memory. Please note that the performance capability of all accelerators was calculated as the total number of operations needed to classify one image using a dense CNN, multiplied by the reported frames per second. This ensures a fair comparison between architectures that exploit CNN sparsity and dense accelerators. In addition, some papers report logic utilization per logic cell instead of LUT. These results are rescaled to match LUT utilization.

As can be seen from Table V, Argus in configuration with 1 CC has the best GOP/s/DSP, with about 30 % better results than the first competitor [27]. CoNNa C4 is 50 % behind Argus, while others show from 2 to 10 times worse GOP/s/DSP results. Considering GOP/s/LUT, [27] is the best competitor with a 10 % difference compared to Argus. Considering GOP/s/LUT, the best is [26], with a 10 % difference compared to Argus. CoNNa shows competitive results, while others have from 3 to 100 times lower GOP/s/LUT compared to Argus.

Argus shows far better utilization of BRAMs than all others, outputting about 2 GOP/s/BRAM. It has 2 times better performance density than the accelerators in [27] and [25], and from 2.7 to 45 times better than others. As two other ratios, (GOP/s/DSP, GOP/s/LUT), GOP/s/BRAM can become a limiting factor for further accelerator scaling, as in the case of [27]. In the most powerful configuration, in [27], accelerator utilize 80 % of BRAMs, while utilizing only 53 % of available DSPs.

Even though Argus has better results than competitors for VGG16, it has the potential to be even more efficient in the case of modern networks, like the MobileNet family, without large fully-connected layers. Because of the limited compression ratio (50 %), the loading of weights in fully connected layers takes about 25 % of the whole processing time.

TABLE V. COMPARISON OF PERFORMANCE DENSITY PER USED HARDWARE RESOURCES.

VGG16	**[42]**	**[23]**	**[24]**	**[15]**	**[25]**	**[26]**	**[27]**	**Argus 1CC**	**Argus 4CC**
DSP:	1728	1058	256	128	220	1144	1350	74	272
LUT (k):	200	100	267	229	13.4	252	178	14	46
BRAM (36 kb):	640	784	596	386	98	912	1460	51	202.5
GOP/s/DSP:	0.28	0.25	0.95	0.14	0.43	0.27	1.1	**1.42**	1.25
GOP/s/k LUT:	2.43	2.66	0.91	0.08	7.13	1.23	**8.34**	7.51	7.41
GOP/s/BRAM:	0.76	0.34	0.41	0.05	0.97	0.34	1.02	**2.06**	1.68

VI. CONCLUSIONS

This paper proposed a novel CNN pruning algorithm, called "FPGA-aware pruning" and a resource-efficient complete CNN hardware accelerator called "Argus". The pruning algorithm exploits two different techniques to achieve high regularity in compressed CNN. Besides the high regularity, the algorithm is specially tailored to be suitable for FPGA-based acceleration. One of the used techniques, kernel clustering, reduces the size of zero-

skipping logic by a factor of 2. Furthermore, the proposed FPGA-aware pruning algorithm reduces the logic resources consumption from 20 % to 50 % in the case of ASIC and FPGA implementations, respectively, when compared to the previously proposed solution [21]. The architecture of Argus, together with the new pruning algorithm, enables very efficient usage of available FPGA resources, enabling Argus to be implemented in the smallest FPGA devices, and still being able to reach high CNN processing performance. Therefore, Argus is best suited to be used in edge-based applications. Argus compares very favorably with some of the previously proposed solutions like FpgaConvNet, Snowflake, NullHop, NEURAghe, Caffeine, CoNNa, Depthwise optimized accelerator, Eyeriss v2, Envision, and NVDLA, usually being able to reach higher frame rates, or achieve similar processing performance results with significantly lower resource consumption. This is particularly the case when the fps-per-MAC metric is being used.

CONFLICTS OF INTEREST

The authors declare that they have no conflicts of interest.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning", *Nature*, vol. 521, pp. 436–444, 2015. DOI: 10.1038/nature14539.

[2] G. Litjens *et al*., "A survey on deep learning in medical image analysis", *Medical Image Analysis*, vol. 42, pp. 60–88, 2017. DOI: 10.1016/j.media.2017.07.005.

[3] A. Luckow, M. Cook, N. Ashcraft, E. Weill, E. Djerekarov, and B. Vorster, "Deep learning in the automotive industry: Applications and tools", in *Proc. of IEEE International Conference on Big Data (Big Data)*, Washington, DC, 2016, pp. 3759–3768. DOI: 10.1109/BigData.2016.7841045.

[4] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, "Object recognition with gradient-based learning", in *Lecture Notes in Computer Science*, vol. 1681. Springer, Berlin, Heidelberg, 1999. DOI: 10.1007/3-540-46805-6_19.

[5] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet classification with deep convolutional neural networks", *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, May 2017. DOI: 10.1145/3065386.

[6] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition", in *Proc. of The 3rd International Conference on Learning Representations (ICLR2015)*, 2015. arXiv: 1409.1556.

[7] C. Szegedy, V. Vanhoucke, S. loffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision", in *Proc. of 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2818–2826. DOI: 10.1109/CVPR.2016.308.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", in *Proc of 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[9] B. Zoph, V. Vasudevan, J. Shlens, and Q. Le, "Learning transferable architectures for scalable image recognition", in *Proc. of 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 8697–8710. DOI: 10.1109/CVPR.2018.00907.

[10] A. G. Howard *et al*., "MobileNets: Efficient Convolutional Neural Networks for mobile vision applications", 2017. arXiv: 1704.04861.

[11] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning Convolutional Neural Networks for resource efficient inference", 2016. arXiv: 1611.06440.

[12] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices", *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019. DOI: 10.1109/JETCAS.2019.2910232.

[13] S. Zhang *et al*., "Cambricon-X: An accelerator for sparse neural networks", in *Proc. of 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Taipei, Taiwan, 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783723.

[14] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks", in *Proc. of 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, Seoul, South Korea, 2016, pp. 367–379. DOI: 10.1109/ISCA.2016.40.

[15] A. Aimar *et al*., "NullHop: A flexible convolutional neural network accelerator based on sparse representations of feature maps", *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 3, pp. 644–656, 2019. DOI: 10.1109/TNNLS.2018.2852335.

[16] Y. Chen *et al*., "DaDianNao: A machine-learning supercomputer", in *Proc. of 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Cambridge, UK, 2014, pp. 609–622. DOI: 10.1109/MICRO.2014.58.

[17] J. Zhu, J. Jiang, X. Chen, and C.-Y. Tsui, "SparseNN: An energy-efficient neural network accelerator exploiting input and output sparsity", in *Proc. of 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden, Germany, 2018, pp. 241–244. DOI: 10.23919/DATE.2018.8342010.

[18] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "14.5 Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable Convolutional Neural Network processor in 28nm FDSOI", in *Proc. of 2017 IEEE International Solid-State Circuits Conference (ISSCC)*, San Francisco, CA, 2017, pp. 246–247. DOI: 10.1109/ISSCC.2017.7870353.

[19] S. Yin *et al*., "A 1.06-to-5.09 tops/w reconfigurable hybrid-neural-network processor for deep learning applications", in *Proc. of Symposium on VLSI Circuits*, Kyoto, Japan, 2017, pp. C26–C27. DOI: 10.23919/VLSIC.2017.8008534.

[20] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "UNPU: A 50.6TOPS/W unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision", in *Proc. of International Solid-State Circuits Conference - (ISSCC)*, 2018, pp. 218–220. DOI: 10.1109/ISSCC.2018.8310262.

[21] H.-J. Kang, "Accelerator-aware pruning for Convolutional Neural Networks", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 30, no. 7, pp. 2093–2103, 2020. DOI: 10.1109/TCSVT.2019.2911674.

[22] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello, "Snowflake: A model agnostic accelerator for deep Convolutional Neural Networks", 2017. arXiv: 1708.02579.

[23] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks", in *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Austin, TX, 2016, pp. 1–8. DOI: 10.1145/2966986.2967011.

[24] R. Struharik, B. Vukobratovic, A. Erdeljan, and D. Rakanovic, "CoNNA - Compressed CNN hardware accelerator", in *Proc. of 2018 21st Euromicro Conference on Digital System Design (DSD)*, Prague, Czech Republic, 2018, pp. 365–372. DOI: 10.1109/DSD.2018.00070.

[25] F. Spagnolo, S. Perri, F. Frustaci, and P. Corsonello, "Energy-efficient architecture for CNNs inference on heterogeneous FPGA", *Journal of Low Power Electronics and Applications*, vol. 10, no. 1, p. 1, 2020. DOI: 10.3390/jlpea10010001.

[26] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang, "An efficient hardware accelerator for sparse convolutional neural networks on FPGAs", in *Proc. of 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, San Diego, CA, USA, 2019, pp. 17–25. DOI: 10.1109/FCCM.2019.00013.

[27] C. Zhu, K. Huang, S. Yang, Z. Zhu, H. Zhang, and H. Shen, "An efficient hardware accelerator for structured sparse convolutional neural networks on FPGAs", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 9, pp. 1953–1965, 2020. DOI: 10.1109/TVLSI.2020.3002779.

[28] S. Han, H. Mao, and W. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization, and Huffman coding", *arXiv: Computer Vision and Pattern Recognition*, 2016. arXiv: 1510.00149.

[29] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. Graf, "Pruning filters for efficient ConvNets, 2017. arXiv: 1608.08710.

[30] X. Zhou *et al*., "Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach", in *Proc. of 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 15–28. DOI: 10.1109/MICRO.2018.00011

[31] J. Qiu *et al*., "Going deeper with embedded FPGA platform for convolutional neural network", in *Proc. of 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'16)*, New York, NY, USA, 2016, pp. 26–35. DOI: 10.1145/2847263.2847265.

[32] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "A quantitative analysis on microarchitectures of modern CPU-FPGA platforms", in *Proc. of 2016 53rd Annual Design Automation Conference (DAC'16)*, New York, NY, USA, 2016, pp. 1–6. DOI: 10.1145/2897937.2897972.

[33] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database", in *Proc. of IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[34] Chollet, Francois, & others, Keras, 2015. [Online]. Available: https://keras.io

[35] P. Colangelo, N. Nasiri, E. Nurvitadhi, A. Mishra, M. Margala, and K. Nealis, "Exploration of low numeric precision deep learning inference using Intel® FPGAs", in *Proc. of 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 73–80. DOI: 10.1109/FCCM.2018.00020.

[36] R. Struharik and V. Vranjkovic, "Stick buffer cache v2: Improved input feature map cache for reducing off-chip memory traffic in CNN accelerators", in *Proc. of 2019 27th Telecommunications Forum (TELFOR)*, Belgrade, Serbia, 2019, pp. 1–4. DOI: 10.1109/TELFOR48224.2019.8971049.

[37] D. Rakanovic, A. Erdeljan, V. Vranjkovic, B. Vukobratovic, P. Teodorovic, and R. Struharik, "Reducing off-chip memory traffic in deep CNNs using stick buffer cache", in *Proc. of 2017 25th Telecommunication Forum (TELFOR)*, 2017, pp. 1–4. DOI: 10.1109/TELFOR.2017.8249398.

[38] R. P. Tidwell, "Alpha blending two data streams using a DSP48 DDR technique", Xilinx, XAPP706 (v1.0) Mar. 31, 2005. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp706.pdf

[39] B. Ronak and S. A. Fahmy, "Multipumping flexible DSP blocks for resource reduction on Xilinx FPGAs", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 9, pp. 1471–1482, 2017. DOI: 10.1109/TCAD.2016.2629421.

[40] R. Zhao, X. Niu, and W. Luk, "Automatic optimising CNN with depthwise separable convolution on FPGA", in *Proc. of 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018. DOI: 10.1145/3174243.3174959.

[41] NVDLA. [Online]. Available: http://nvdla.org/

[42] P. Meloni *et al*., "NEURAghe: Exploiting CPU-FPGA synergies for efficient and flexible CNN inference acceleration on Zynq SoCs", *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, pp. 18:1–1:24, 2018. DOI: 10.1145/3284357.