A streaming accelerator of Convolutional Neural Networks for resource-limited applications

Moisés Arredondo-Velázquez\textsuperscript{1}, Javier Díaz-Carmona\textsuperscript{1,3}, Cesar Torres-Huitzil\textsuperscript{2}, Alejandro-Israel Barranco-Gutiérrez\textsuperscript{1}, Alfredo Padilla-Medina\textsuperscript{1}, and Juan Prado-Olivarez\textsuperscript{1}

Abstract Convolutional Neuronal Networks (CNN) implementation on embedded devices is restricted due to the number of layers of some CNN models. In this context, this paper describes a novel architecture based on Layer Operation Chaining (LOC) which uses fewer convolvers than convolution layers. A reutilization of hardware convolvers is promoted through kernel decomposition. Thus, an architectural design with reduced resources utilization is achieved, suitable to be implemented on low-end devices as a solution for portable classification applications. Experimental results show that the proposed design has a competitive processing time and overcomes resource utilization when compared with state-of-the-art related works.

Keywords: Convolutional Neural Networks, streaming architecture, Layer Operation Chaining

Classification: Devices, circuits and hardware for IoT and biomedical applications

1. Introduction

Deep learning algorithms allow computational modelling for data representation learning at different abstraction levels, which have remarkably improved the state of the art in application areas like image and voice recognition [1]. Since the outstanding breakthrough shown in 2012 [2, 3, 4, 5, 6], CNNs are the most used tools to deal with image classification problems, providing a broad variety of possibilities to Internet of Things (IoT) applications, such as: license plate detection [7], face recognition [8], autonomous vehicles [9], etc. since it is possible to compute all data on-chip and to obtain a higher level information for sending to the servers instead of raw data to be online processed. Local processing allows the devices to operate with greater efficiency and have a higher autonomy, therefore reducing the workload and bandwidth for local servers. Nevertheless, the CNN computational complexity is usually higher than traditional algorithms [10, 11, 12], so the use of graphics processing units and online servers are a frequent choice to accelerate operations in CNNs. Thus, there is an increasing demand for real-time processing solutions for CNNs, mainly within embedded implementations. State-of-the-art works are focused on reducing the amount of resources through two major design strategies. Single Convolver Architectures (SCA) is one of such strategies, where convolution layer are sequentially processed by adapting resources to the different layers hyperparameters [13, 14, 15, 16]. Multiple Convolver Architectures (MCA) are the second strategy that employs one convolver for each CNN convolutional layers, where a higher performance is achieved when multiple images are processed, but at the expense of a resources increment [17, 18]. A new design approach is proposed in this article, which is based on using Layer Operation Chaining (LOC) through convolutional layers where it is possible to process a single image using multiple convolvers, such that the number of convolvers $N_c$ is smaller than convolutional layers $N_l$, i.e., $1 < N_c < N_l$. There are two essential motivations for using LOC. The first one is to achieve a smaller CNN processing time than the two identified design strategies for a single image processing. The second one is to eliminate the intermediate memories between convolvers.

The main contributions of this work are the following:

- An architecture based on LOC, where a single input image is processed by reusing $N_c$ convolvers in the CNN with $N_l$ convolutional layers, and $1 < N_c < N_l$.
- A strategy to decompose filter size to obtain a programmable convolver able to process different layers.
- The validation of the proposed design strategy by implementing a naive CNN with lower resource utilization compared to the state-of-the-art and a competitive processing time.

The rest of paper is organized as follows. Section 2 gives a brief overview of CNNs. Section 3 describes major state-of-the-art design strategies. Section 4 introduces design approach and proposal architecture. Experimental results and conclusion are shown in sections 5 and 6, respectively.

2. CNN background

A CNN is mainly composed by convolutional layers and a Fully Connected Network (FCN) as a classification stage.
2.1 Convolutional layer

Each convolutional layer receives a feature map $\mathcal{F} \in \mathbb{R}^{M \times N \times CH}$ to be convolved with multiple kernels $\mathcal{K} \in \mathbb{R}^{w \times h \times CH \times K}$ and produces new feature maps $\mathcal{G} \in \mathbb{R}^{M' \times N' \times CH' \times K'}$, expressed as: $\mathcal{F} \otimes \mathcal{K} = \mathcal{G}$. The layer hyperparameters are: the image rows (N), columns (M), filters (K), and filter size (w). A superindex (l) denotes the layer to which every hyperparameter belongs.

Other parameters are Stride (S) and Padding (P). S is used to avoid the kernel operate on whole image. Kernel operation on input maps edges is controlled by P, which lets output maps preserve the same dimension.

Depending on CNN topology, a convolutional layer can be associated with a pooling stage, where the output map dimension is decreased through maximum or mean computing (Maxpooling or Menpooling) of certain image region.

2.2 Classification stage

A FCN is composed by $n_l$ layers, where the initial is denoted as $L_1$ and $L_o$, represents the output layer. The $S_l$ and $C$ values represent the number of neurons in the $l$ and output layer, respectively. The output value of a neuron (Activation) $z_l^{(i)}$ is given by $f(z_l^{(i)}) = a_l^{(i)}$, where $a_l^{(i)}$ is equivalent to $x_l^{(i)} = x_l^{0(1)} + x_l^{1(1)} + \ldots + x_l^{n_l(1)}$. Additional term $x_l^{0(1)} = b$ lets $f(z_l)$ be a non linear function. Note that $i$ depends on $x_l^{(i-1)}$. The LOC can only be applied in a classification stage with $n_l = 2$.

3. Related work

Two major design strategies can be identified. SCA speeds up the whole CNN computing through the use of only one programmable convoler to sequentially process all the layers by adapting to the different layers hyperparameters [20, 21, 22, 23]. In [15] a programmable convoler reuses data through operations chaining between convolution and pooling processing elements (PE), thereby a reduction of external memory accesses is achieved. PEs are clustered in a module called collection and the whole architecture consists of four collections. Convolution inside each collection can be configured to perform one $12 \times 12$, four $6 \times 6$ or sixteen $3 \times 3$ convolutions. This solution dealing with different kernel sizes is also adopted in [24] and [25].

MCA implements one convoler for each convolutional layer, where after a certain latency different input images are parallel processed by all the CNN convolutional layers. The works [18, 26, 27, 28] are examples of this strategy. In [18] convolvers are optimized exclusively to their corresponding hyperparameters, thus a throughput increment is achieved for multiple input images. In [27] a streaming architecture is employed, therefore, operations for different layers are processed in parallel. On-chip buffers store activations while weights are stored on off-chip memory. In addition, the batch-based processing method is applied in the classification stage, thereby, it is possible to process multiple images in parallel using the same parameters of the CNN.

Algorithm 1: Pseudo code of a convolutional layer

\begin{align*}
\text{for } Ch \text{ from } 1 \text{ to } CH^{(l)} \text{ do} \\
\text{for } k \text{ from } 1 \text{ to } K^{(l)} \text{ do} \\
\text{for } y \text{ from } 1 \text{ to } N^{(l+1)} \text{ do} \\
\text{for } x \text{ from } 1 \text{ to } M^{(l+1)} \text{ do} \\
\phantom{\text{for }} \text{perform } \mathcal{G}[x,y,k]^{+} = \sum_{i=0}^{w-1} \sum_{j=0}^{h-1} \mathcal{F}[x \times S + i, y \times S + j, ch] \otimes K[i, j, ch, k] \\
\end{align*}

The main characteristic of SCA is the adaptation to different hyperparameters, such as kernel size (w), Stride, Padding, as well as the number of filters and channels to be parallel processed ($nf, nf_{ij}$, respectively). The total number of used convolvers in this strategy is one ($Nc = 1$). MCA maximizes the throughput by parallel processing multiple input images through a distribution of the convolvers workload based on the convolutional layer complexity. Thus, a CNN with $N_l$ convolutional layers is processed using $N_{c}$ convolvers ($N_c = N_l$). A significant trend for convolution computing based on Algorithm 1 is found in the state-of-the-art works, which may vary by interchanging the two external loops ($CH$ and $K$). Such change directly influences on the hardware implementation technique like the use of line buffers [25, 29, 26] or Tiling maps [16, 20, 15] as input buffers, or using systolic array [16] or multipliers arrays [25, 18] to solve convolution operation.

LOC is used together with multiple convolvers for single-image processing to reduce processing time, at the cost of an increment of resource utilization in contrast to SCA. On the other hand, our approach is not able to parallel compute multiple images as the MCA does. Nevertheless, a reduction of intermediate memories is achieved. Although there are some related works using multiple convolvers [19], where $1 < N_c < N_l$, they are not focused on speeding up a single-image processing as LOC does.

4. Hardware accelerator

4.1 Overall operation

The proposed architecture based on LOC, for a three convolutional layers CNN, is shown in Fig. 1. Owing to the LOC a loops reordering in Algorithm 1 is used, since $K^{(l)}$ kernels must be processed by each convolver on a particular mask position before the position update of such
mask. Hence, the operation of this design follows Algorithm 2.

The architecture consists of \( N_c \) convolvers, where \( 1 < N_c < N_l \). A single image is pipeline processed through the \( N_c \) convolvers and storage for intermediate results between convolvers is not needed, only results of the last convolver are stored. Main control fetches stored data once current layers processing ends. Yet, some components are reused to process more than one layer. Fig. 1 shows a three convolutional layer CNN, processed by two convolvers as follows:

**Algorithm 2: Pseudo code of a convolutional layer**

```plaintext
for y from 1 to \( N' \) do
  for x from 1 to \( M' \) do
    for k from 1 to \( K' \) do
      for \( Ch \) from 1 to \( CH' \) do
        perform \( \mathcal{G}[x, y, k] = \sum_{f=0}^{w_f-1} \sum_{i=0}^{h_i-1} \mathcal{F}[x \times S + i, y \times S + j, ch] \otimes \mathcal{K}[i, j, ch, k] \)
      \end{algorithm}
```

**STAGE 1** To setup a valid mask defined by \( w' \) and \( N' \), the input image is fed to the programmable buffers \( C_{LB}(k) \). The number of buffers for the initial convolver is \( CH' = 1 \) (in Fig. 1). Once a valid mask is generated, a Multiplier-Accumulator array (MAC array) convolves such mask with \( n_f \) kernels. This MAC array is programmable for six \( 5 \times 5 \) or twelve \( 3 \times 3 \) kernels (programmability is described in Section 4.3). Results are pipelined sent to the pooling PEs through buffers \( P_{LB}(k) \). This pipeline process ends when a valid mask exists in the line buffers for the second convolver. Note that layer 2 convolver and pooling line buffers are not programmable (for the described CNN) since second convolutional layer is processed by the second convolver.

**STAGE 2** A valid mask is available on the convolver line buffers \( C_{LB}(1)^{(1)}, C_{LB}(1)^{(2)}, \ldots, C_{LB}(K)^{(1)} \) and the second layer is computed using a similar architecture of the initial convolver. Results are stored in \( K'^{(2)} = CH'^{(3)} \) FIFO memories.

**STAGE 3** Once all first convolver outputs are computed, data of the FIFO memories (Layer 3 FIFOs) can be fetched by the control to the \( CH'^{(3)} \) line buffers to be convolved by the first convolver when a valid mask is available. The programmable line buffer and convolver are setup with the appropriate hyperparameters. The convolver results are then send to the classification stage.

**STAGE 4** Components within this stage receive data generated from convolver that compute layer \( N_l \). A collection of \( C \) MACs (\( C \) output neurons) multiplicate and accumulate the corresponding data and weights, hence, a partial prediction during overall processing is obtained.

The operation sequence for the three convolutional layers CNN is shown in Table I. At a time \( T_1 \), the input image is fetched, stored in line buffers, and then processed by the pre-configured convolver. The first layer results are pooled in \( T_2 \). In \( T_3 \) the pooling results are processed by the second convolver, and the first and second layers are parallel processing the same input image. The second layer results are then processed by the Layer 2 Pooling component in \( T_4 \), and the transition to the next time point occurs when all the results of the first convolutional layer have been calculated. The Layer 1/3 convolver is setup according to third layer hyperparameters and it is started in \( T_5 \), then the convolver output is sent to the classification stage to update the partial prediction. The classification stage is fed with the first results from Layer 1/3 convolver in \( T_6 \). After this time every received data is multiplied and accumulated by \( C \) MACs. At \( T_7 \) only layer three and the classification stage operate. Finally, the CNN result is obtained in \( T_8 \) plus the classification stage latency. The operation for the proposed architecture would be similar for CNNs with a greater number of layers: the results of each component can be sent to the subsequent PEs and the results for new layers would be stored in FIFO memories until the component ends the computing of current layer.

### Table I. Timing diagram for architecture shown in Fig. 1.

| Stage | Component | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 |
|-------|-----------|----|----|----|----|----|----|----|----|
| 1     | CONV1     |    |    |    |    |    |    |    |    |
| 2     | POOL1     |    |    |    |    |    |    |    |    |
| 3     | CONV2     |    |    |    |    |    |    |    |    |
| 4     | POOL2     |    |    |    |    |    |    |    |    |
| 5     | CONV3     |    |    |    |    |    |    |    |    |
| 6     | CLASS     |    |    |    |    |    |    |    |    |

#### 4.2 Programmable line buffer

Since the number of convolvers is fewer than the number of layers in the proposed accelerator, programmable line buffers (shown in Fig. 2) are used to generate the mask of the processed layer. We employ line buffers [14], but add derivations (shown in blue in Fig. 2) to adjust the data path (selected by the central control using multiplexers).

![Fig. 2. Programmable line buffer for \( w = 3 \) and \( w = 5 \).](image)

#### 4.3 Programmable convolver

As a core computation engine, we use a MACs Array. Inspired by [15, 25], a set of MACs subarrays compute \( 3 \times 3 \) kernels. To find a suitable parallelism level, the number of multipliers \( nm \) used for the convolution is analyzed. A convolution with a kernel size of \( w \times w \) requires \( (w^2/nm) \) cycles to complete the operation, where the total number of multiplications are \( Cmp = w^2 \). The operation parallelism \( OP \) represents such operation cycles to compute a \( w \times w \) convolution. Given the hyperbole function \( OP(nm) = [(c_{mp}/nm)] \), we state a cost function based
on the number of multiplications and the number of operation cycles, \( cost = nm \times OP \). Thus, there are three points on \( OP(nm) \) where the cost is the minimum; \( nm = 1 \), \( nm = w \), and \( nm = w^2 \). In addition, an ideal solution consists of using the fewest multipliers and operation cycles as possible. The \( nm \) value closest to this is \( nm = w \), and corresponds to nearest distance value to origin, which is defined by the function point having a slope of \(-45^\circ\) as:

\[
dOP \over dnm = -\text{Cmp} \over nm^2 = \tan \Theta = -1, \quad \therefore nm = \sqrt{\text{Cmp}}. \tag{1}
\]

Each convolution uses \( w \) multipliers and \( w \) operation cycles. To convolve over a larger kernel, the MACs subarray is concatenated until the desired kernel size is covered. Two subarrays are concatenated for a \( 5 \times 5 \) kernel, where one MAC is idle. We use \( 3 \times 3 \) and \( 5 \times 5 \) kernel as they are common kernel sizes. Each MAC array row is composed of two subarrays of three MACs each; a MACs array depth of six was established. The number of parallel processed kernels depends on the \( w \) value. A MACs array that computes two convolutional layers is shown in Fig. 3, where \( w^{(1)} = 5 \), \( nf^{(1)} = 6 \) and \( w^{(3)} = 3 \), \( nf^{(3)} = 12 \). The array moves the convolution mask row to each MACs according to \( w \).

Fig. 4 shows the MACs array row datapath. For \( w = 3 \) a higher number of kernels is parallel processed and there are no idle processing elements as in the case with \( w = 5 \), where one MAC of each row is not used along the processing.

5. Experimental results

The proposed architecture was validated by implementing a three-layer CNN with the characteristics shown in Table II and applied to the MNIST dataset. The architecture was designed in VHDIL and implemented on the ALTERA De2i-150 platform. To evaluate the architecture performance, the obtained results are compared with the obtained from a CNN using a number of convolvers equal to the number of convolutional layers, \( N_c = N_i \). The CNN was implemented using two convolvers, according to the proposed architecture.

5.1 Multipliers usage

In this work the considered resource utilization types are \textit{intra-utilization} and \textit{inter-utilization}. The first one is related to the number of multipliers within the MACs array. Our MACs array have an \textit{intra-utilization} of 69.44% and 100% for kernel sizes of \( 5 \times 5 \) and \( 3 \times 3 \), respectively. In our proposal the \textit{intra-utilization} for a \( 5 \times 5 \) kernel size can be enhanced to 97.22% by adding datapaths for parallel computing an extra kernel, since only one MAC would be idle. Nevertheless, the efficient resources reutilization inside a convolver is the main concern of SCA, which is not of the scope of this work.

Regarding to \textit{inter-utilization}, two main affecting aspects must be considered. First, the ratio between the number of convolutional layers and the number of convolvers. Second, the convolutional layers computational complexity. Overall, certain number of convolvers will be idle on the last operation iteration. e.g., for our proposal, in first iteration convolver 1 computes layer 1, and convolver 2 computes layer 2 for the same input image. Then, in the second iteration convolver 1 computes layer 3 whereas convolver 2 keeps idle. Consequently, \textit{inter-utilization} is 100% and 50% for the first and second iterations, respectively. For the first iteration, this fact is true for T3 to T5 in Table I, where both convolvers are being parallel computed. Thus, an average \textit{inter-utilization} of 75% is attained. These shown percentages do not contemplate initial and final latency in each layer. i.e. T1 to T3 and T5 to T7 in Table I.

Previous example represent the worst case, since a CNN with an odd number of convolutional layers will not fit well in our proposal. Instead, a CNN with an even number of layers will have an increased \textit{inter-utilization}. The first convolver can read data from memory as required, meaning that the first convolver output data is given until all kernels and input maps are processed for a given mask position (this according to Algorithm 2). Thus, subsequent convolvers have a slower rate input data than previous ones. Thus, if the first convolver processes a more complex layer than the subsequent convolver, a balance between their processing time will happen. The worst case scenario,
is presented when one convolver must compute a more complex layer than the assigned to the previous one. Once the less complex layer is computed, the subsequent convolver will continue computing data and consequently first convolver will be idle. This idle time corresponds to the final latency. In this context, authors recommend a further analysis to find out an adequate parallelism degree to maximize resource utilization.

5.2 Processing time acceleration

The processing time ($P_t$) and latency ($Lat$) for both implementations are presented in Table III. The processing time for the layers and classification stage is similar for both architectures, but the acceleration is different. The processing time for architecture where $N_c = N_l$ can be estimated by:

$$T_{arch1} = T_{Conv}^{(j)} + Lat_{FC} + \sum_{l=1}^{N_l} Lat_{Pool}^{(l)} + Lat_{Conv}^{(l+1)}.$$

(2)

where $T_{Conv}^{(j)}$ is the layer $j$ processing time, which is the largest for $j = 2$. The latency values corresponding to classification stage, convolutional layer $l$ and its respective pooling component are: $Lat_{FC}$, $Lat_{Pool}^{(l)}$ and $Lat_{Conv}^{(l+1)}$, respectively. The processing time $T_{Conv}^{(l)}$ is given by:

$$T_{Conv}^{(l)} = \left(\frac{(w(l))^2}{nm}\right) \times \left(\frac{K(l)}{nfo}\right) \times CH(l) \times N(l+1) \times T.$$

(3)

where $T$ is the clock period, and $Lat_{Conv}^{(l+1)}$ is the latency time of the convolutional layer $l + 1$ for each channel:

$$Lat_{Conv}^{(l)} = \frac{T_{Conv}^{(l)}}{N(l+1)}.$$

(4)

On the other hand, the processing time for architecture with $1 < N_c < N_l$ is defined as:

$$T_{arch2} = T_{Conv}^{(1)} + Lat_{Conv}^{(2)} + T_{Conv}^{(3)} + Lat_{FC}.$$

(5)

The architecture performance parameters are related to $N_c$ and $N_l$. For $N_c = N_l$ the LOC is effective along all the CNN layers. Hence, the processing time for $N_c = N_l$ is smaller than the corresponding for $1 < N_c < N_l$. Such result is confirmed with the total processing times shown in Table III, where a reduction of a 6.5% is achieved for $N_c = N_l$.

The comparison of resources utilization for both architectures is shown in Table IV. As it is observed, a notably resources reduction is achieved by the proposed architecture ($1 < N_c < N_l$), since the use of a third convolver is omitted.

### Table IV. Resources utilization for $N_c = N_l = 3$ and $N_l = 2$.

| Method            | $N_c = 3$, $N_l = 3$ ($N_c = N_l$) | $N_c = 3$, $N_l = 2$ ($1 < N_c < N_l$) |
|-------------------|-----------------------------------|---------------------------------------|
| FPGA              | Cyclone IV (EP4CGX150DF31C7N)     | Cyclone IV (EP4CGX150DF31C7N)         |
| LE                | 57678/149760 (39%)                | 20655/149760 (14%)                    |
| Registers         | 38587                             | 10808                                 |
| Memory (bits)     | 215436/6635520 (3%)               | 137812/6635520 (2%)                   |
| Multipliers       | 290/720 (40%)                     | 138/720 (19%)                         |

The architecture performance comparison with other works is shown in Table V considering $N_c = 1$ and $N_c = N_l$, for different versions of the CNN LeNet-5. The original CNN LeNet-5 [30], is implemented in [23] and [18]. The CNN model in [25] uses a third convolutional layer and an input image size of $32 	imes 32 	imes 3$. Two main upsides can be gained by the proposed design strategy. First, the reutilization of first convolver allows that the number of multipliers (DSP blocks) and Logic Elements be smaller than the work in [18]. Furthermore, a precision analysis [23] might reduce even more the number of multipliers. Second, the intermediate memory buffer required by the architecture using $N_c = N_l$ is not required owing to LOC. The memory (Memory bits) for the proposed implementation is much smaller than the case $N_c = N_l$, but it is greater than for $N_c = 1$. In addition, a smaller energy consumption (comparable with the ASIC implementation) is observed compared with the second design strategy. Finally, a competitive processing time per image ($P_t$/Image) is achieved by the the 8-bit CNN implemented on a low-end FPGA running at 100 MHz.

### Table V. Architecture performance comparison.

| Work               | [23] | [25] | [18] | This work |
|--------------------|------|------|------|-----------|
| Platform           | LeNet5 | Modified LeNet | LeNet5 | Three layer LeNet |
| Cyclone V          | ASCI 65 nm | Virtex 7 | Cyclone IV |
| Frequency (MHz)    | 65 | 500 | 100 | 100 |
| Design strategy    | SCA | SCA | MCA | MCA/LOC |
| GOPS               | 318.48 | 152 | 424.7 | 0.6271 |
| Power (W)          | - | 0.35 | 25.2 | 0.736 |
| Energy Efficiency (GOPS/W) | - | 434 | 16.85 | 0.85 |
| Pt/Image (us)      | - | 1093 | 1318 | 399 |
| Precision (bits)   | 3 | 16 | 8 | 8 |
| Logic Elements     | 8967 (7%) | - | 232315 (54%) | 20655 (14%) |
| DSP Blocks         | 0 (1%) | - | 2907 (80.8%) | 158 (19%) |
| Memory (bits)      | 376 (1%) | - | 1717200 (52.4%) | 137812 (2%) |
6. Conclusion

This paper describes a novel design strategy for CNN based on LOC, where hardware reuse is exploited by using fewer convolvers than convolution layers ($1 < N_c < N_l$). A three-layers CNN was implemented on a low-end FPGA. Resources reductions of 64.18%, 71.99%, 36% and 52.41% for logic elements (LE), registers, memory and multipliers, respectively, were achieved when compared with the implementation of same the CNN with $N_c = N_l$. Yet, a small processing time increment of 6.9% was observed. A smaller processing time than reported works using either $N_c = 1$ or $N_c = N_l$ is obtained. Also a reduction of multipliers was achieved compared with the design strategy that uses $N_c = N_l$. The design strategy is a suitable solution for CNN applications where resources utilization are limited.

References

[1] Y. LeCun, et al.: “Deep learning,” Nature 521 (2015) 436 (DOI: 10.1038/nature14539).

[2] A. Krizhevsky, et al.: “Imagenet classification with deep convolutional neural networks,” Adv. Neural Inf. Process. Syst. (2012) 1097 (DOI: 10.1145/3065396).

[3] K. Simonyan and A. Zisserman: “Very deep convolutional networks for large-scale image recognition,” arXiv:1409.1556 (2014).

[4] C. Szegedy, et al.: “Going deeper with convolutions,” Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. (2015) 1 (DOI: 10.1109/CVPR.2015.7298594).

[5] K. He, et al.: “Deep residual learning for image recognition,” Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. (2016) 770 (DOI: 10.1109/CVPR.2016.90).

[6] J. Hu, et al.: “Squeeze-and-excitation networks,” Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. (2018) 7132 (DOI: 10.1109/CVPR.2018.00745).

[7] L. Xie, et al.: “A new CNN-based method for multi-directional car license plate detection,” IEEE Trans. Intell. Transp. Syst. 19 (2018) 507 (DOI: 10.1109/TITS.2017.2784093).

[8] O. M. Parkhi, et al.: “Deep face recognition,” BMVC (2015) 3 (DOI: 10.5244/C.29.41).

[9] C. Chen, et al.: “Deep-driving: Learning affordance for direct perception in autonomous driving,” Proc. IEEE Int. Conf. Comput. Vis. (2015) 2722 (DOI: 10.1109/ICCV.2015.312).

[10] M. Anthimopoulos, et al.: “A two-stage scheme for text detection in video images,” Image Vis. Comput. 28 (2010) 1413 (DOI: 10.1016/j.imavis.2010.03.004).

[11] M. Khammari, et al.: “Robust face detection using haar features with weber local descriptor and local binary pattern,” KES 19 (2015) 117 (DOI: 10.3233/KES-150313).

[12] E. Fidalgo, et al.: “Compass radius estimation for improved image classification using edge-SIFT,” Neurocomputing 197 (2016) 119 (DOI: 10.1016/j.neucom.2016.02.045).

[13] A. Arakani, et al.: “An architecture to accelerate convolution in deep neural networks,” IEEE Trans. Circuits Syst. I, Reg. Papers 65 (2018) 1349 (DOI: 10.1109/TCSI.2017.2757036).

[14] J. Chang and J. Sha: “An efficient implementation of 2D convolution in CNN,” IEICE Electron. Express 14 (2017) 20161134 (DOI: 10.1587/elex.13.20161134).

[15] A. Dendar, et al.: “Embedded streaming deep neural networks accelerator with applications,” IEEE Trans. Neural Netw. Learn. Syst. 28 (2017) 1572 (DOI: 10.1109/NNLS.2016.2545298).

[16] F. Tu, et al.: “Deep Convolutional Neural Network architecture with reconfigurable computation patterns,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 25 (2017) 2220 (DOI: 10.1109/TVLSI.2017.2688340).

[17] S. I. Venieris and C.-S. Bouganis: “fpgaConvNet: Mapping regular and irregular convolutional neural networks on FPGAs,” IEEE Trans. Neural Netw. Learn. Syst. 30 (2018) 326 (DOI: 10.1109/TNNLS.2018.2844093).

[18] Z. Liu, et al.: “Throughput-optimized FPGA accelerator for Deep Convolutional Neural Networks,” ACM Trans. Reconfigurable Technol. Syst. 10 (2017) 17 (DOI: 10.1145/3079758).

[19] Y. Shen, et al.: “Maximizing CNN accelerator efficiency through resource partitioning,” Comput. Archit. News 45 (2017) 535 (DOI: 10.1145/3140659.3080221).

[20] C. Zhang, et al.: “Optimizing fpga-based accelerator design for deep convolutional neural networks,” ACM SIGDA-FPGA (2015) 161 (DOI: 10.1145/2684746.2689060).

[21] T. Chen, et al.: “Dianmao: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” ACM Sigplan Notices (2014) 269 (DOI: 10.1145/2541940.2541967).

[22] Y.-H. Chen, et al.: “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks,” IEEE J. Solid-State Circuits 52 (2017) 127 (DOI: 10.1109/JSSC.2016.2616357).

[23] K. Abdelouahab, et al.: “Tactics to directly map CNN graphs on embedded FPGAs,” IEEE Embed. Syst. Lett. 9 (2017) 113 (DOI: 10.1109/LES.2017.2743247).

[24] H. Nakahara and T. Sasao: “A deep convolutional neural network based on nested residue number system,” Int. Conf. Field Prog. (2015) 1 (DOI: 10.1109/FPL.2015.7293933).

[25] L. Du, et al.: “A reconfigurable streaming Deep Convolutional Neural Network accelerator for Internet of Things,” IEEE Trans. Circuits Syst. I, Reg. Papers 65 (2018) 198 (DOI: 10.1109/TCSI.2017.2754940).

[26] N. Li, et al.: “A multistage dataflow implementation of a Deep Convolutional Neural Network based on FPGA for high-speed object recognition,” SSIAI (2016) 165 (DOI: 10.1109/SSIAI.2016.7459201).

[27] H. Li, et al.: “A high performance FPGA-based accelerator for large-scale convolutional neural networks,” Int. Conf. Field Prog. (2016) 1 (DOI: 10.1109/FPL.2015.7577308).

[28] Y. Wang, et al.: “DeepBurning: Automatic generation of FPGA-based learning accelerators for the neural network family,” Design Automation Conference (DAC) (2016) 110 (DOI: 10.1145/2897937.2898003).

[29] L. Lu, et al.: “Evaluating fast algorithms for convolutional neural networks on fpgas,” FCCM (2017) 101 (DOI: 10.1109/FCCM.2017.64).

[30] Y. LeCun, et al.: “Gradient-based learning applied to document recognition,” Proc. IEEE 86 (1998) 2278 (DOI: 10.1109/5.726791).