



Faster Convolutional Neural Networks in Low Density FPGAs Using Block Pruning

Tiago Peres¹ , Ana Gonçalves¹ , and Mário Véstias²  

¹ ISEL, Instituto Politécnico de Lisboa, Lisbon, Portugal

² INESC-ID, ISEL, Instituto Politécnico de Lisboa, Lisbon, Portugal
mvestias@deetc.isel.ipl.pt

Abstract. Convolutional Neural Networks (CNNs) are achieving promising results in several computer vision applications. Running these models is computationally very intensive and needs a large amount of memory to store weights and activations. Therefore, CNN typically run on high performance platforms. However, the classification capabilities of CNNs are very useful in many applications running in embedded platforms close to data production since it avoids data communication for cloud processing and permits real-time decisions turning these systems into smart embedded systems. In this paper, we improve the inference of large CNN in low density FPGAs using pruning. We propose block pruning and apply it to LiteCNN, an architecture for CNN inference that achieves high performance in low density FPGAs. With the proposed LiteCNN optimizations, we have an architecture for CNN inference with an average performance of 275 GOPs for 8-bit data in a XC7Z020 FPGA. With our proposal, it is possible to infer an image in AlexNet in 5.1 ms in a ZYNQ7020 and in 13.2 ms in a ZYNQ7010 with only 2.4% accuracy degradation.

Keywords: Convolutional Neural Network · FPGA · Block pruning

1 Introduction

A CNN consists of a series of convolutional layers where the output of a layer is the input of the next. Each layer generates an output feature map (OFM) with specific characteristics of the input image or of the previous input feature map (IFM). Each feature map is obtained from the convolution of a filter and the IFM. The last layers of the CNN are usually the fully connected (FC) layers that associate a matching probability of the image with one of the classes. Besides convolutional and fully connected layers there may be other layers, like the pooling layer and a non-linear layer (e.g. ReLU).

AlexNet [1], a large CNN, won the ImageNet Challenge. It consists of five convolutional layers plus three FC layers. Different number of kernels with different sizes are applied at each layer with a total of 61M weights requiring a 724 MACC (Multiply-accumulate) operations. Other CNN models have followed, like VGG-16 [2], GoogleNet [3] and ResNet [4].

Executing a CNN model (inference) can be done on the same platform used to train it or in an embedded computing platform with strict performance, memory and energy constraints. In a vast set of embedded applications, it is advantageous or necessary to have the inference process near the data input sensor so that important information can be extracted at the image sensor instead of sending the information to the cloud and wait for the answer. Also, in systems where the communication latency and data violations are undesirable, like autonomous vehicles, local processing at the sensor is also desirable.

A common feature of these CNN models is the high number of weights and operations. Due to the limited performance and memory of many embedded platforms it is very important to find architectural solutions to run large CNN inferences in low hardware density embedded platforms. Recently, a high performance architecture for CNN inference - LiteCNN - was proposed [5]. With a peak performance of 410 GOPs in a ZYNQ7020 FPGA (Field-Programmable Gate Array) it does an inference of AlexNet in about 17 ms.

To improve the processing delay of the inference, pruning (weight cut) can be applied, usually to the FC layers followed by data quantization. The method permits to reduce the number of operations to be performed as well as the memory size to store the weights. The problem of the method is that the sparsity introduced challenges the regular structures of computing datapaths. To reduce the sparsity problem caused by pruning, we propose a block pruning technique in which weights are pruned in blocks. We have studied the impact of pruning and the block size over the performance and area of LiteCNN.

The following has been considered for the optimization of LiteCNN:

- We have implemented a flow based on Caffe [6] and Ristretto [7] to optimize networks using block pruning followed by quantization;
- LiteCNN was upgraded to support the implementation of block pruned CNN;
- A performance model for pruned LiteCNN was developed to allow design space exploration;
- Tradeoffs among performance, area and accuracy were obtained allowing the designer to choose the most appropriate LiteCNN configuration for a particular CNN model.

The paper is organized as follows. Section 2 describes the state of art on FPGA implementations of CNNs and optimization methods based on pruning. Section 3 describes the flow used to explore block pruning and quantization. Section 4 describes the LiteCNN architecture, the modifications necessary to support pruning and the performance model. Section 5 describes the results on inference accuracy and area/performance of LiteCNN running well-known CNNs and compare them to previous works. Section 6 concludes the paper.

2 Related Work

Common general processing units achieve only a few hundred GFLOPs with low power efficiency. This performance is scarce for cloud computing and the energy

consumption is too high for embedded computing. GPUs (Graphics Processing Units) and dedicated processors (e.g. Tensor Processing Unit - TPU) offer dozens of TOPs and therefore appropriate for cloud computing.

FPGAs are increasingly being used for CNN inference for its high performance/energy efficiency, because it permits to implement a dedicated hardware architecture for each CNN model. This is an important feature if we want to apply it to embedded computing.

A few authors considered low density FPGAs as the target device. In [8] small CNNs are implemented in a ZYNQ XC7Z020 with a performance of 13 GOPs with 16 bit fixed-point data. In [9] the same FPGA is used to implement big CNN models, like VGG16, with data represented with 8 bits achieving performances of 84 GOPs. In [10] the authors implemented a pipelined architecture in a ZYNQ XC7Z020 with data represented with 16-bit fixed point. The architecture achieves 76 GOPs with high energy efficiency.

Previous works [11] show that dynamic fixed-point with 8 bits guarantee similar accuracies compared to those obtained with 32-bit floating point representations. This reduction is essential to implement CNN in target platforms with low on-chip memory and low resources. LiteCNN is a configurable architecture that can be implemented in small density FPGAs. The architecture has a peak performance of 410 GOPs in a ZYNQ XC7Z020 with 8-bit dynamic fixed-point data representation for activations and weights. This was a great performance improvement over previous implementations in the same FPGA.

In [12] deep neural networks are compressed using pruning, trained quantization and huffman coding. The techniques are applied on CPU and GPU implementations. Results show that pruning on, e.g., AlexNet results in 91% weight cut without sacrificing accuracy. In [13] pruning is considered to improve CNN execution implemented in FPGA, similar to what is done in [14]. The architecture dynamically skips computations with zeros. The problem is that they keep a dense format to store the matrix requiring to be all loaded from memory. Also, they target high density FPGAs. In [15] the authors use a large FPGA with enough capacity to store all weights on-chip after pruning. This is not possible in low FPGAs with scarce internal memory.

In [16] the pruning is adapted to the underlying hardware matching the pruning structure to the data-parallel hardware arithmetic unit. The method is applied to CPU and GPU. In this paper we propose a similar approach with block pruning. The best block pruning is found and then the hardware architecture is adapted to its size.

We have improved CNN inference in LiteCNN by exploring block pruning in the fully connected layers followed by dynamic fixed-point quantization of all layers. The new LiteCNN architecture keeps the peak performance since we do not skip zero values, but the inference delay was reduced by more than 70% since we have reduced the number of fully connected weights to be transmitted from external memory to LiteCNN, the major performance bottleneck at FC layers.

3 Framework for Data Reduction

A framework to explore the block pruning of weights in fully connected weights followed by data quantization was developed based on Caffe [6] as the main framework and Ristretto for data quantization. The Framework trains the network and generates a file of trained weights.

Pruning can be implemented with different metrics and methods to reduce the number of weights. In this work we have considered the weights magnitude. A percentage of weights whose magnitude is closer to zero is iteratively removed according to the flow in Fig. 1.

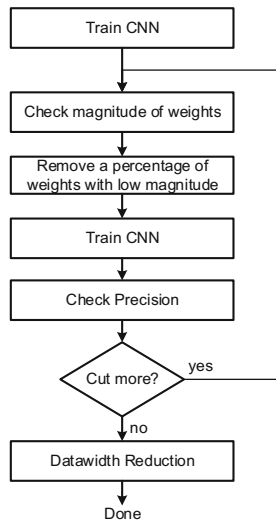


Fig. 1. Network pruning flow

In the first step we train the network or start with a pre-trained network. Then, a percentage of weights with low magnitude (below a predefined threshold) is pruned. The network is trained again with single precision floating-point. We check if the precision allows more pruning. When no more pruning is allowed, we apply Ristretto to reduce the data size. From the results, we extract the fixed-point quantifications for each layer.

Pruning introduces sparsity in the kernels of weights which degrades the performance. Also, introduces an overhead associated with the index information of the sparse vector of weights. To improve the hardware implementation and the performance of pruned networks we introduce the block pruning which performs a coarse pruning with blocks of weights. The method reduces the index overhead data and permits to efficiently use the parallel MACs of the processing units.

The technique permits to prune blocks of weights (similar to what is done in [16]) instead of single weights (see example in Fig. 2).

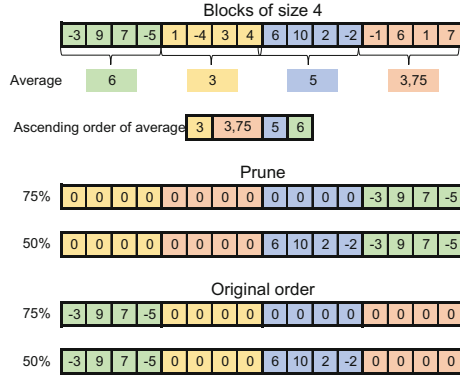


Fig. 2. Pruning method for blocks of four weights

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

4 LiteCNN Architecture

4.1 LiteCNN Architecture

The Lite-CNN architecture consists of a cluster of processing elements (PE) to calculate dot-products, a memory buffer to store on-chip the initial image and the OFMs, one module to send activations and two modules to send and to receive weights to/from the PEs (see Fig. 3).

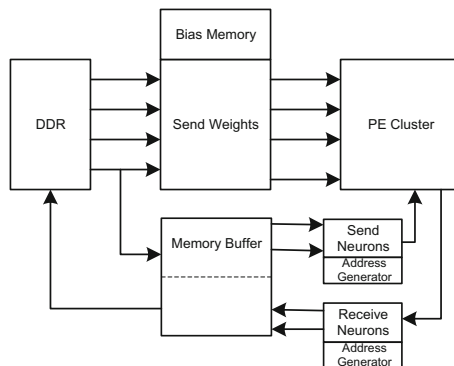


Fig. 3. Block diagram of the Lite-CNN architecture

The architecture executes layers one at a time. The execution of layers work as follows:

- Before starting the execution of a layer, the architecture is configured for the specific characteristics of the layer. It also specifies if there is a pooling layer at the output of the feature maps being calculated;
- The input image and the intermediate feature maps are stored on-chip. Since the layers are executed one at a time, the on-chip only has to be enough to store the IFM and OFM of any layer;
- For the first convolutional layer, the image is loaded from external memory. For the others, the IFM is already in on-chip memory. At the same time, kernels are read from external memory and sent to the PEs. Besides the weights, the kernel includes the bias value which is stored in the bias memory. Each PE receives one kernel. So, each PE calculates the activations associated with one OFM;
- The initial image or intermediate feature maps in the on-chip memory are broadcasted to all PEs;
- After each calculation of a complete dot product associated with a kernel, all PEs send the output activations back to the receive neurons module that adds the bias and stores the result in the on-chip memory to be used by the next layer. If the layer is followed by pooling, this module saves the activations in a local memory and wait for the other members of the pooling window;
- The process repeats until finishing the convolution between the image and the kernels. After that, the next kernels are loaded from memory and the process repeats until running all kernels of a layer.

The process allows overlapping of kernel transfer and kernel processing. While the PEs process their kernels, in case the local memory is enough to store two different kernels, the next kernels are loaded at the same time. This is fundamental in the fully connected layers where the number of computations is the same as the number of weights.

Also, in case the on-chip memory is not enough to store the whole image and the OFM (usually the first layer is the one that requires more on-chip memory), the image is cut into pieces which are convolved separately.

The PE cluster contains a set of PEs. Each PE (see Fig. 4) has a local memory to store kernels and arithmetic units to calculate the dot product in parallel.

Each PE stores a different kernel and so it is responsible for calculating the activations of the output feature map associated with the kernel. This way multiple output feature maps are calculated in parallel. Also, in convolutional layers, the same kernel is applied to different blocks of the IFM and produce different neurons of its OFM. The number of output neurons to be processed in parallel in each PE is configurable. For example, to calculate two activations in parallel it receives two input activations from the feature memory in parallel. This mechanism permits to explore the intra-output parallelism. Finally, weights and activations are stored in groups, that is, multiple weights and activations are

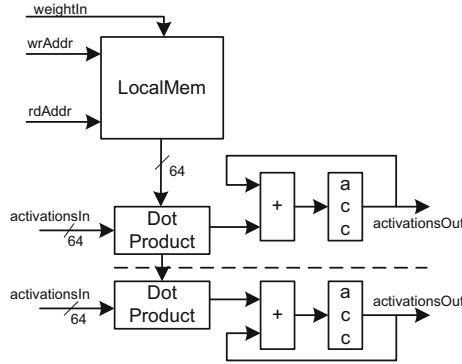


Fig. 4. Architecture of the processing elements

read in parallel in a single memory access (e.g., with 8-bit data, a 64 memory word contains eight neurons or weights) permitting to explore dot-product parallelism.

The block *sendWeights* is configured to send kernels to the PE cluster. The block receives data from direct memory access (DMA) units that retrieve data from external memory and send it to the PEs in order. It includes a bias memory to store the bias associated with each kernel.

The *sendNeurons* and *receiveNeurons* blocks are responsible for broadcasting activations from the feature memory to the PEs and receive dot products from the PEs, respectively. The send neurons module includes a configurable address generator. The receive neurons module implements the pooling layer in a centralized manner.

Most of the previous approaches use dedicated units to calculate 2D convolutions. The problem is that the method becomes inefficient when the same units have to run different window sizes. Lite-CNN transforms 3D convolutions into a long dot product to become independent of the window size. Pixels of the initial image, activations of feature maps and weights of kernels are stored in order (z, x, y) (see Fig. 5).

Each neuron of an OFM is calculated as a dot product between the 3D kernel of size $x_k \times y_k \times z_k$ and the correspondent neurons of the IFM of size $x_p \times y_p \times z_p$ (see Fig. 5b), where z_p is the number of IFMs. The weights of kernel are all read sequentially from memory since they are already ordered. The neurons are also read in sequence from memory but after $x_k \times z_k$ neurons it has to jump to the next y_k adding an offset to the address of the input feature memory being read. For a layer without stride nor followed by pooling, the offset is $x_p \times z_p$.

LiteCNN also implements a method to reduce the number of multiplications by half [5] leading to a considerable reduction in the hardware resources required to implement a convolutional or fully connected layers.

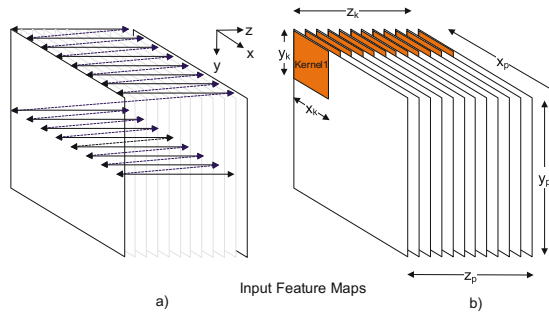


Fig. 5. Reading mode of images, feature maps and weights

We have extended LiteCNN to support pruned FC layers as follows:

- The sparse vectors of weights are sent to the local memory of PEs. The next address index is stored in the parity bits of the BRAMs which were not used in the original LiteCNN. When the size of the index is not enough, we consider extra zero blocks in the middle;
- Activations are sent to the processing elements keeping its dense format and multiply-accumulated by the respective weights. If the activation index corresponds to a zero weight block then its is multiplied by zero keeping the pipeline full.

This solution has no computational advantage, since the number of operations is the same as the case without pruning, but the weight data to be read from memory is considerably reduced. Since the data reduction method is applied in the fully connected layers where the data access is the bottleneck and not the computations, the method permits to achieve high performance improvements, as will be seen in the results. Also, it simplifies the implementation of the PEs permitting to keep the operating frequency and only a small increase in the required hardware resources.

The main modification of the LiteCNN datapath was in the arithmetic core of the PE (see Fig. 6).

Two different datapath modifications are considered. One in which the block size times the quantized datawidth (8 bits) equals 64 (Fig. 6a). In this case, each block has 8 weights the same number of activations received in parallel by the core. The second datapath is when the block size times the quantized datawidth (8 bits) equals 32. In this case, the blocks have only 4 weights and so are read in words of 32 bits. Since the core receives 8 activations in parallel, we read two independent groups of weights from two independent local memories (Fig. 6b).

4.2 Performance Model of LiteCNN

The performance model provides an estimate of the inference execution time of a CNN network on the LiteCNN architecture with block pruning. The model determines the time to process each layer.

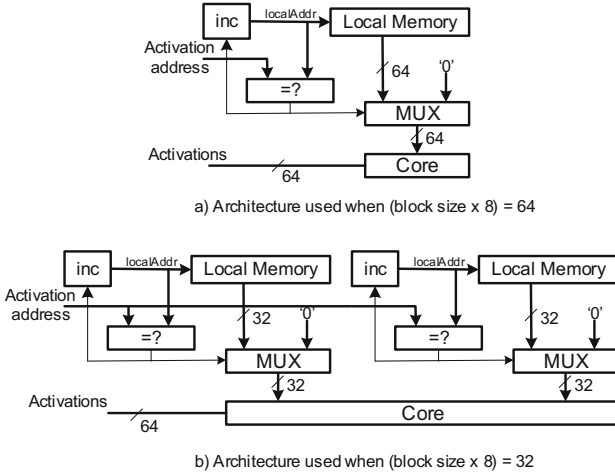


Fig. 6. Modified datapath of the PE to support weight pruning

Considering convolutional layers, the time to transfer all kernels depends on the number of kernels, $nKernel$, the size of kernels, $kernelSize$, the number of bits used to represent weights, $nBit$ and the memory bandwidth, BW . The total number of bytes, $tConvByte$, transferred in each convolutional layer is given by Eq. 1.

$$tConvByte = nKernel \times kernelSize \times \frac{nBit}{8} \tag{1}$$

The number of cycles to execute a convolutional layer, $convCycle$, is

$$convCycle = \left\lceil \frac{nKernel}{nCore} \right\rceil \times \frac{nConv \times kernelSize}{nMAC} \tag{2}$$

where $nCore$ is the number of processing elements, $nConv$ is the number of 3D convolutions and $nMAC$ is the number of parallel multiply-accumulations of each PE (intra-output parallelism). From these two equations, the total execution time, $convExec$ depends on the local memory capacities. If local memories of PEs have enough space to store two kernels, than communication and processing of kernels can overlap, otherwise, they must be serialized. Considering an operating frequency, $freq$ de execution time is given by Eq. 4.

$$convExec = \frac{tByte}{BW} + \frac{convCycle}{freq} \quad \text{without overlap} \tag{3}$$

$$convExec = \max\left(\frac{tByte}{BW}, \frac{convCycle}{freq}\right) \quad \text{with overlap} \tag{4}$$

For the totally connected layers, the equation to determine the number of bytes to transfer all kernels, $tFCByte$, must consider the size of the pruning blocks, $bSize$, and the pruning percentage, $prune$, (see Eq. 5).

$$tFCByte = nKernel \times kernelSize \times \frac{nBit}{8} \times \frac{100 - prune}{100} \times \frac{1 + bSize}{bSize} \quad (5)$$

The equation to determine the number of cycles to process the FC layer is given by:

$$fcCycle = \left\lceil \frac{nKernel}{nCore} \right\rceil \times \frac{kernelSize}{nMAC} \times nParallel \quad (6)$$

Since in the fully connected layers there is no intra-output parallelism, only one line of parallel MACs of the PE is used. Given the number of intra-output parallel processing lines, $nParallel$, the number of processing cycles is multiplied by this value.

The total execution time of FC layers is similar to 4.

$$fcExec = \frac{tFCByte}{BW} + \frac{fcCycle}{freq} \quad \text{without overlap} \quad (7)$$

$$fcExec = \max\left(\frac{tFCByte}{BW}, \frac{fcCycle}{freq}\right) \quad \text{with overlap} \quad (8)$$

The total execution of a CNN inference in LiteCNN is the sum of the time to transfer the image to FPGA and the result from FPGA ($\frac{imageSize+result(bytes)}{BW}$) plus the time to process each layer. Between layers there is negligible configuration time of the architecture to adapt to the layer done by the ARM processor of ZYNQ.

We have checked the accuracy of the model from the results of LiteCNN 8×8 running AlexNet. The delay obtained with the model without pruning is about 1% lower (16.94 ms) against (17.1 ms) of the implementation.

5 Results

We describe the results of the pruning methodology with LeNet, Cifar10-full and AlexNet. All LiteCNN architectures were implemented with Vivado 2017.3 in the ZedBoard with a ZYNQ XC7Z020 and in a ZYBO board with a ZYNQ7010 and run at 200 MHz.

For each CNN we found the relation between block pruning and accuracy. For AlexNet (the larger and most demanding CNN) we have determined the relation between pruning and delay. All results of accuracy are for top-1 classification, since the state-of-the-art works we are comparing also use this metric. Similar tradeoffs were obtained when the top-5 accuracy is used as the metric.

LiteCNN was configured and implemented with 4, 8 and 16 bit dynamic fixed-point (fixed-point numbers in different layers may have different scaling factors), with different block pruning sizes and for each configuration the number of cores was adjusted to obtain a similar area (see the area results in Table 1).

Table 1. Area occupation for different block size configurations of LiteCNN

<i>Activation</i> × <i>Weight</i>	4 × 4		8 × 8		16 × 16	
	8	16	4	8	2	4
PEs	60	64	64	64	38	38
MACC/PE	32	32	16	16	16	16
LUT	47661	47477	47830	43378	45232	43614
DSP	220	220	220	220	220	220
BRAM (36 Kbits)	130	130	130	130	132	132
Peak GOPs	768	819	410	410	243	243

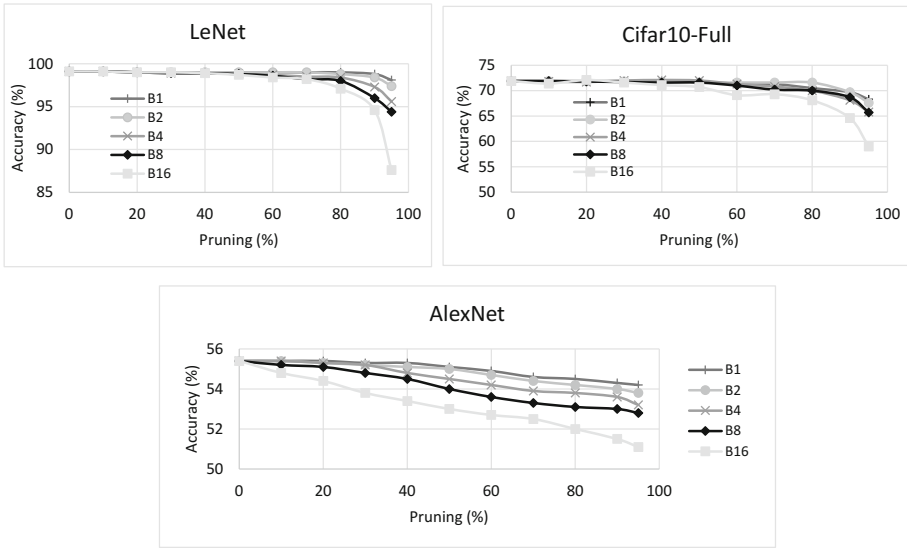


Fig. 7. Variation of accuracy with pruning percentage and block size

The table gives the number of processing elements and the number of MACC in each PE. A line with the peak performance was also included (The peak performance takes into consideration that the architecture reduces the number of multiplications to half).

For each CNN, we have determined the accuracy of the network for different pruning percentages with different block sizes and 8 bit dynamic fixed-point quantization (see Fig. 7, where Bx is the configuration with block size x).

From the results, we observe that the size of the pruning block has a small influence over the accuracy, except for a block of 16. In this worst case, the lost in accuracy is about 4%. Similar results were obtained with 16 bit quantization since the accuracy difference between 8 bit and 16 bit quantizations is small (around 1.5%).

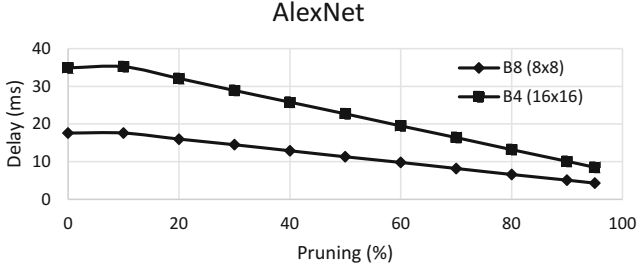


Fig. 8. Variation of delay with pruning percentage for configuration B8 and B4

In order to keep a fair comparison with previous works, we have determined the delay of configurations B8 ($activation \times weight = 8 \times 8$) and B4 ($activation \times weight = 16 \times 16$) for AlexNet for different pruning percentages (see Fig. 8).

Pruning has a big impact in the inference delay of AlexNet in LiteCNN since the execution bottleneck of AlexNet is in the fully connected layers because of the huge number of weights to be transferred from external memory. Pruning FC layers reduces the communication time and consequently the whole inference process.

We have also tested with LeNet and Cifar10-Full. With LeNet the delay reduces from 0.1 ms to 0.01 ms when we increase pruning from 10% to 90%. In the case of Cifar10-Full the impact is negligible since the only FC layer of the network has only 2.2% of the total number of weights of the CNN.

We have compared configuration B4 with 16 bit quantization and 8 bit quantization, both with 90% of pruning (1% accuracy loss) with previous works running AlexNet. The overall results are shown in Table 2.

Compared to previous works implemented in the ZYNQ xc7z020, in particular the best implementation from [19], the peak performance and the ratios GOPs/kLUT and GOPs/DSP of LiteCNN are about $2 \times$ better and the latency is about $5 \times$ better. LiteCNN (8×8 configuration) reduces the latency of the original implementation of LiteCNN without pruning (17 ms) to only 5.1 ms with only 1% accuracy loss. This delay allows an inference performance of 196 images/s in a ZYNQ xc7z020.

With LiteCNN we could map AlexNet in the smallest SoC FPGA from Xilinx - ZYNQ7010 - in a ZYBO board. As expected, inference delays are higher because it has less resources (less PEs) and since the available on-chip RAM is not enough to hold the image and the first OFM, the image has to be halved and processed separately. However, notably, it can run AlexNet in real-time (30 fps).

To better understand the impact of pruning of FC weights on the inference delay, we have determined the time to execute convolutional layers and the time to execute FC layers (see Fig. 9). The graph indicates the observed GOPs (and the percentage of peak performance).

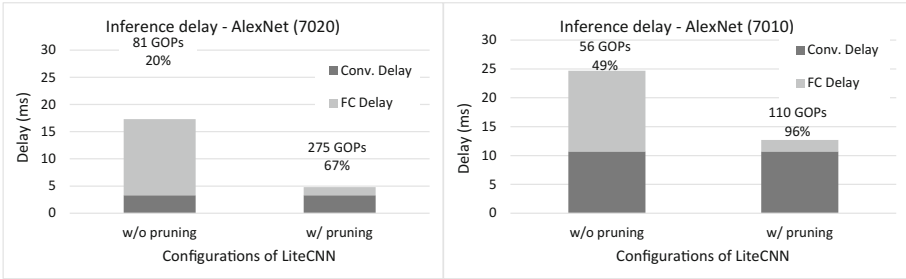
Without pruning, the execution time of FC layers is higher than that of convolutional layers. The execution time of FC layers is dominated by the com-

Table 2. Performance comparison of Lite-CNN with other works in low density ZYNQ7020 and ZYNQ7010 SoC FPGAs

Work	Format	Freq (MHz)	GOPs	GOPs/LUT	GOPs/DSP	Latency (ms)	Acc.
ZYNQ 7020							
[17]	16 × 16	100	19	0.35	0.08	71.75	(a)
[18]	16 × 16	150	20	0.38	0.09	—	(a)
[19]	16 × 16	125	38	0.73	0.17	52.4	(a)
[10]	16 × 16	200	80	1.5	0.36	16.7 ^(b)	(a)
[9]	8 × 8	214	84	1.6	0.38	—	53.9
LiteCNN	16 × 16	200	139	3.2	0.63	10.1	53.7
LiteCNN	8 × 8	200	275	6.3	1.25	5.1	53.5
ZYNQ 7010							
LiteCNN	8 × 8	200	275	6.3	1.25	13.2	53.5

(a) Authors assume accuracy close to that obtained with floating-point - 55.9%

(b) With pruning and image batch

**Fig. 9.** Execution time of convolutional and FC layers for LiteCNN with and without pruning running AlexNet

munication of weights from external memory. This fact degrades the average GOPs. Pruning FC weights improves the real GOPs of the architecture. The real GOPs improves when LiteCNN is mapped on ZYNQ7010. In this case, the execution time of FC layers is about the same (the memory bandwidth is the same in both FPGAs) and the execution time of convolutional layers increase. So, the implementation in ZYNQ7010 is more efficient.

6 Conclusions

In this work we have proposed block pruning and modified the LiteCNN architecture to support pruned regular networks. The extended LiteCNN with configurable pruning datapath proposed in this work permits to improve the performance/area efficiency while keeping the inference accuracy of the CNN. This is fundamental for embedded systems with low resources.

The results show that block pruning achieves very good accuracies and at the same time simplifies the hardware implementation for regular CNN.

We are now studying the relation between pruning and data size reduction.

Acknowledgment. This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2019 and was also supported by project IPL/IDI&CA/2018/LiteCNN/ISEL through Instituto Politécnico de Lisboa.

References

1. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1, NIPS 2012, pp. 1097–1105. Curran Associates Inc., USA (2012)
2. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In: Proceedings of the 3rd International Conference on Learning Representations (2015)
3. Szegedy, C., et al.: Going deeper with convolutions. In: 2015 IEEE Conference on Computer Vision and Pattern Recognition, CVPR, pp. 1–9, June 2015
4. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR, pp. 770–778, June 2016
5. Véstias, M.P., Duarte, R.P., de Sousa, J.T., Neto, H.: Lite-CNN: a high-performance architecture to execute CNNs in low density FPGAs. In: Proceedings of the 28th International Conference on Field Programmable Logic and Applications (2018)
6. Jia, Y., et al.: Caffe: convolutional architecture for fast feature embedding. arXiv preprint [arXiv:1408.5093](https://arxiv.org/abs/1408.5093) (2014)
7. Gysel, P., Pimentel, J., Motamedi, M., Ghiasi, S.: Ristretto: a framework for empirical study of resource-efficient inference in convolutional neural networks. *IEEE Trans. Neural Netw. Learn. Syst.* **29**, 5784–5789 (2018)
8. Venieris, S.I., Bouganis, C.S.: fpgaConvNet: a framework for mapping convolutional neural networks on FPGAs. In: 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM, pp. 40–47, May 2016
9. Guo, K., et al.: Angel-Eye: a complete design flow for mapping CNN onto embedded FPGA. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* **37**(1), 35–47 (2018)
10. Gong, L., Wang, C., Li, X., Chen, H., Zhou, X.: MALOC: a fully pipelined FPGA accelerator for convolutional neural networks with all layers mapped on chip. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* **37**(11), 2601–2612 (2018)
11. Gysel, P., Motamedi, M., Ghiasi, S.: Hardware-oriented approximation of convolutional neural networks. In: Proceedings of the 4th International Conference on Learning Representations (2016)
12. Han, S., Mao, H., Dally, W.J.: Deep compression: compressing deep neural network with pruning, trained quantization and Huffman coding. *CoRR*, abs/1510.00149 (2015)

13. Nurvitadhi, E., et al.: Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, pp. 5–14. ACM, New York (2017). <https://doi.org/10.1145/3020078.3021740>
14. Albericio, J., Judd, P., Hetherington, T., Aamodt, T., Jerger, N.E., Moshovos, A.: Cnvlutin: ineffectual-neuron-free deep neural network computing. In: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture, ISCA, pp. 1–13, June 2016
15. Fujii, T., Sato, S., Nakahara, H., Motomura, M.: An FPGA realization of a deep convolutional neural network using a threshold neuron pruning. In: Wong, S., Beck, A.C., Bertels, K., Carro, L. (eds.) ARC 2017. LNCS, vol. 10216, pp. 268–280. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56258-2_23
16. Yu, J., Lukefahr, A., Palframan, D., Dasika, G., Das, R., Mahlke, S.: Scalpel: customizing DNN pruning to the underlying hardware parallelism. SIGARCH Comput. Archit. News **45**(2), 548–560 (2017). <https://doi.org/10.1145/3140659.3080215>
17. Wang, Y., Xu, J., Han, Y., Li, H., Li, X.: DeepBurning: automatic generation of FPGA-based learning accelerators for the neural network family. In: 2016 53rd ACM/EDAC/IEEE Design Automation Conference, DAC, pp. 1–6, June 2016
18. Sharma, H., et al.: From high-level deep neural models to FPGAs. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, pp. 1–12, October 2016
19. Venieris, S.I., Bouganis, C.: fpgaConvNet: mapping regular and irregular convolutional neural networks on FPGAs. IEEE Trans. Neural Netw. Learn. Syst. **30**(2), 326–342 (2019)