Lite-CNN: A High-Performance Architecture to Execute CNNs in Low Density FPGAs

Mário Véstias
INESC-ID, ISEL,
Instituto Politécnico de Lisboa
mvestias@deetc.isel.pt

Rui Policarpo Duarte, José T. de Sousa, Horácio Neto
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa, Portugal
rui.duarte@tecnico.ulisboa.pt, jose.desousa@inesc-id.pt, hcn@inesc-id.pt

***Abstract*—Due to the computational complexity of Convolutional Neural Networks (CNNs), high performance platforms are generally considered for their execution. However, CNNs are very useful in embedded systems and its execution right next to the source of data has many advantages, like avoiding the need for data communication. In this paper, we propose an architecture for CNN inference (Lite-CNN) that can achieve high performance in low density FPGAs. Lite-CNN adopts a fixed-point representation for both neurons and weights, which was already shown to be sufficient for most CNNs. Also, with a simple and known dot product reorganization, the number of multiplications is reduced to half. We show implementation results for 8 bit fixed-point in a ZYNQ7020 and extrapolate for other larger FPGAs. Lite-CNN achieves 410 GOPs in a ZYNQ7020.**

***Index Terms*—Embedded computing, Deep learning, Convolutional Neural Network, Field-Programmable Gate Array.**

I. Introduction

Deep Neural Networks (DNN) are at the core of many artificial intelligence applications, like car driving assistance [1], image classification [2], A Convolutional Neural Network (CNN) is a DNN consisting of multiple convolutional layers. A stack of 2D input feature maps (IFM) is convolved with a 3D filter to generate an output feature map (OFM). Many filters are applied to the same IFMs generating a stack of OFM.

In this paper, the focus is on a hardware architecture (Lite-CNN) for running inference of large CNNs in low density FPGAs (Field-Programmable Gate Arrays).

The results demonstrate that Lite-CNN can achieve high performance in low density FPGAs. The architecture is scalable and with larger FPGAs several TOPs (Tera Operations per second) of performance can be achieved.

The paper is organized as follows. In section II we describe the state of art on FPGA implementations of CNNs. Section III describes the proposed Lite-CNN architecture. Section IV shows the results and how they compare to previous works. Section V concludes the paper.

II. Related Work

FPGA is a good platform to run CNNs since it offers good performance at low energy and can be reconfigured to adapt to each CNN model.

Recently, complete accelerators were implemented for large CNN models [3], [4]. Both works consider a general hardware core for convolution that can execute different convolutional layers with different shapes.

Suda et al. [4] and Qiao [5] both adopt an accelerator for matrix multiplication converting the IFM to a matrix.

Liu et al. [6] proposed a pipeline architecture with multiple parallel processing elements (PEs). This solution requires large amounts of on-chip memory and high memory bandwidth to load weights.

Aydonat et al. [7] used an Intel's Arria 10 FPGA to design a CNN accelerator. They use a Winograd transformation to reduce the number of multiplications in the convolutional layers.

A few authors also considered the ZYNQ XC7Z020 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 implemented bigger CNN models, like VGG16, with data represented with 8 bits achieving performances of 84 GOPs.

The most recent works have performances with several hundred GOPs but using high performance FPGAs. Only a few consider low-cost FPGAs, but performances of only a few dozen GOPs. The Lite-CNN proposed in this work is able to achieve several hundred GOPs in a low cost FPGA, like the ZYNQ7020, with 8 bit data representations.

III. Lite-CNN Architecture

In this section we describe the methods and architecture used to design Lite-CNN. We start with the complexity reduction of the dot-product and then follow with the description of the architecture.

A. Dot Product for Convolutions

Convolutional and FC layers consist of dot products between neurons and weights. To do a convolution between a kernel of weights and an IFM, the kernel slides over the whole feature map to produce a partial output neuron that consists of the dot product between the kernel and a block of input neurons. An output neuron is the accumulation of 2D convolutions between a slice of the kernel and neurons of an IFM, or a 3D convolution between the complete kernel and a block of neurons of the stack of IFMs. The 3D convolution is a dot product, DP_{WP}, between n weights of a kernel, W_i and n neurons P_i, that is

978-1-5386-8517-4/18/$31.00 ©2018 IEEE
DOI 10.1109/FPL.2018.00075

$$DP_{WP} = \sum_{i=0}^{i=n-1} W_i \times P_i \quad (1)$$

In a dot product the number of multiplications can be reduced to half. Lets consider two elements vector, weights $W = (W_0, W_1)$ and pixels $P = (P_0, P_1)$. The dot product can be calculated as:

$$DP_{WP} = (W_0 + P_1)(W_1 + P_0) - W_0W_1 - P_0P_1 \quad (2)$$

Equation 2 has more multiplications but W_0W_1 can be precomputed and P_0P_1 can be computed only once and reused many times for different kernels. One problem with the multiplication in equation 2 is that it could require a multiplication with twice the size of the original multiplications, case one of the operands has zero fractional bits and the other has all bits in the fraction part. Since weights and activations in a CNN in a specific layer have all the same fixed-point representation, the operations can be done with numbers as integers and correct the final value with a shift. Considering $W_0 = W_0^{'}2^m$, $W_1 = W_1^{'}2^m$, $P_0 = P_0^{'}2^k$, $P_1 = P_1^{'}2^k$, where $W_0^{'}$, $W_1^{'}$, $P_0^{'}$, $P_1^{'}$ are the integer parts of the fixed-point representation, we have equation 4.

$$DP_{WP} = 2^{m+k}(W_0^{'}P_0^{'} + W_1^{'}P_1^{'}) \quad (3)$$
$$= 2^{m+k}[(W_0^{'} + P_1^{'})(W_1^{'} + P_0^{'}) - W_0^{'}W_1^{'} - P_0^{'}P_1^{'}] \quad (4)$$

In equation 4 the multiplication is only one bit larger than those in 1. Equation 1 can now be calculated as

$$DP_{WP} = 2^{m+k}[\sum_{i=0}^{i=\frac{n}{2}-1} (W_{2i}^{'} + P_{2i+1}^{'})(W_{2i+1}^{'} + P_{2i}^{'}) - \quad (5)$$
$$- \sum_{i=0}^{i=\frac{n}{2}-1} W_{2i}^{'}W_{2i+1}^{'} - \sum_{i=0}^{i=\frac{n}{2}-1} P_{2i}^{'}P_{2i+1}^{'}] \quad (6)$$

The second sum depends only on the weights and therefore can be calculated a priori for each kernel. The last sum can be calculated one for each block of neurons and then reused for each different kernel.

B. Execution of a CNN Model in Lite-CNN

Lite-CNN considers a configurable structure that implements one layer (convolutional or fully connected) at a time. 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. We have taken a different approach. Lite-CNN transforms 3D convolutions into a long dot product to become independent of the window size. Pixels of the initial image, neurons of feature maps and weights of kernels are stored in order (z, x, y) (see Figure 1).

Input images are stored in this order. IFMs can be stored in this order during the execution of the model and weights in kernels can be also pre-stored in this order.

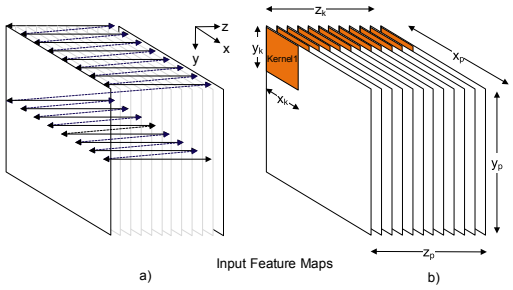

Fig. 1. Reading mode of images, feature maps and weights

Algorithm 1 Convolution with a 3D kernel

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

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 figure 1b), 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 $x_p \times z_p$.

Let's consider the 3D input feature map of size $x_p \times y_p \times z_p$, a kernel of size $x_k \times y_k \times z_k$, a pooling window of size $x_{pool} \times y_{pool}$ and a stride of size m, the convolution of a kernel with the 3D input feature map is given in Algorithm 1.

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

C. Lite-CNN architecture

The Lite-CNN architecture consists of a cluster of processing elements (PE) to calculate dot-products as explained before, a memory buffer to store on-chip the initial image and the OFMs and two modules to send neurons and to receive weights to/from the PEs (see Figure 2).

The architecture works as follows:

- The blocks are configured for a specific layer: convolutional or fully connected. The configuration also specifies

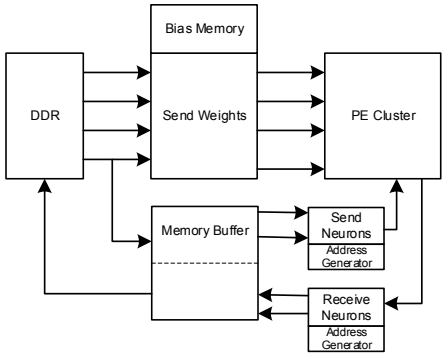

Fig. 2. Block diagram of the Lite-CNN architecture

if there is a pooling and/or a normalization layer at the output of the feature maps being calculated;

- The input image is sent to the memory buffer. At the same time, kernels are read from external memory and sent to the PEs. Besides the weights, the kernel includes a word with the bias value and the sum $\sum_{i=0}^{i=\frac{n}{2}-1} W_{2i}W_{2i+1}$. These values are stored in the bias memory to be used later. Each PE receives one kernel. So, each PE calculates the pixels associated with an OFM;
- Neurons in the memory buffer are broadcasted to all PEs, following the method described in section III-B. While reading neurons, the sum $\sum_{i=0}^{i=\frac{n}{2}-1} P_{2i}P_{2i+1}$ is calculated and stored in a register to be used latter by the receive neurons module, according to equation 6;
- After each calculation of a complete dot product associated with a kernel, all PEs send the output neurons back to the receive neurons module that subtracts the bias and the sum of weights and store the result in the memory buffer to be used by the next layer. If the layer has pooling, this module saves the neurons in a local memory and wait for the other members of the pooling window. Normalization is also implemented in this block;
- 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.

The PE cluster contains a set of PEs (see Figure 3). All PEs receive kernels weights from one or more ports from external memory. Pixels and neurons are broadcasted to all PEs from the on-chip feature memory.

Each PE has a local memory to store kernels and arithmetic units to calculate the dot product

$$DP_{WP} = \sum_{i=0}^{i=\frac{n}{2}-1} (W_{2i} + P_{2i+1})(W_{2i+1} + P_{2i}) \quad (7)$$

Each PE stores a different kernel and so is responsible for calculating the neurons of the output feature map associated with the kernel. This way multiple output feature maps are calculated in parallel. Also, the same kernel is applied to different blocks of input neurons and produce different neurons

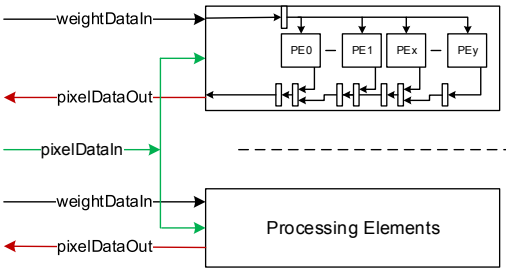

Fig. 3. PE cluster of the Lite-CNN architecture

of its OFM. The number of output neurons to be processed in parallel in each PE is configurable. For example, to calculate two neurons in parallel it receives two input neurons from the feature memory in parallel. This mechanism permits to explore the intra-output parallelism. Finally, weights and neurons are stored in groups, that is, multiple weights and neurons are 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 and the correction factor $\sum_{i=0}^{i=\frac{n}{2}-1} W_{2i}W_{2i+1}$ in equation 6.

The *sendNeurons* and *receiveNeurons* blocks are responsible for broadcasting neurons from the feature memory to the PEs and receive dot products from the PEs, respectively. The send neurons module includes a configurable address generator that implements the *startAddr* function mentioned in algorithm 1. Also, while reading the neurons, it determines the factor $\sum_{i=0}^{i=\frac{n}{2}-1} P_{2i}P_{2i+1}$ in equation 6 and saves it in a register to be used by the *receiveNeurons* block to correct the dot-products coming from the PEs. The receive neurons module implements the pooling, the normalization and activation functions (ReLU).

The memory buffer is a dual port memory with one write port and one read port. It permits to read neurons while saving the previous output pixels.

IV. Results

Lite-CNN was implemented with Vivado 2017.3 in the ZedBoard with a ZYNQ XC7Z020 at 200 MHz.

From among the many possible configurations of Lite-CNN, we implemented an architecture with with 64 PEs, two pNeurons, four nMACC and a batch of 2 with a total peak performance of 410 GOPs.

We have mapped AlexNet in Lite-CNN for 8 fixed-point and compared the performance and area with other implementations on the same FPGA and on higher density FPGAs. The overall results are shown in table I.

Compared to previous works implemented in the ZYNQ xc7z020, Lite-CNN8 has about $5\times$ the peak performance of [9]

TABLE I
PERFORMANCE COMPARISON OF LITE-CNN WITH OTHER WORKS

Ref.	FPGA	Format	LUTs	DSPs	BRAMs	MHz	CNN	GOP/s	BW GB/s	image/s
[4]	Stratix-V GSD8	fixed-16	120000	720	1500	120	AlexNet	118	—	50
[3]	ZYNQ XC7Z045	fixed-16	182616	780	486	150	VGG16	137	4.2	45
[6]	Virtex-7 VX690T	fixed 8,16	206821	2872	1021	100	AlexNet	446	14.9	153
[7]	Arria10 GX1150	float-16	246000	1476	2487	303	AlexNet	1382	17	1020
[8]	ZYNQ xc7z020	fixed-16	35644	208	9	100	LeNet	12.7	3.3	—
[9]	ZYNQ xc7z020	fixed-8	29867	190	86	214	VGG16	84	4.2	2.7
Lite-CNN8	ZYNQ xc7z020	fixed-8	45781	220	132	200	AlexNet	410	4.2	92

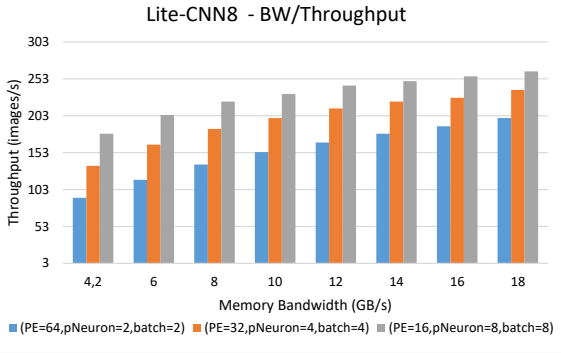

Fig. 4. Bandwidth vs. throughput for different configurations of Lite-CNN8

From the execution of AlexNet, we observed that half of the time is spent loading the weights of the fully connected layers. It means that even if we increase the number of PEs, the memory bandwidth will be the limiting factor. With the memory bandwidth of our system (4.2 GB/s) and increasing the number of PEs, the best throughput it can achieve is around 10 ms per image. The only way to reduce this value is to increase the memory bandwidth or increase the batch size. We have generated different configurations of Lite-CNN8 and estimated the throughput with different memory bandwidths and batch sizes (see figure 4).

As we can see, the throughput converges to a specific value with the increase of the bandwidth. These are the optimal points where the memory bandwidth produces a communication delay equal to the processing delay in which case increasing the bandwidth is useless. Reducing the number of PEs and increasing the intra-output parallelism leaves more memory available.

Finding the best architecture for a specific delay or power consumption is the objective in the design of a CNN architecture for embedded computing. Lite-CNN have shown that it is possible to process complex CNN models with low density FPGAs with high performance.

V. CONCLUSIONS

Lite-CNN is an architecture to run CNN models in low cost FPGAs. Lite-CNN was used to implement AlexNet in a low cost FPGA (ZYNQ xc7z020) and the results show that it is possible to run the inference of the model with small FPGAs in about 11 ms.

We are now planing to improve the configurability of the architecture to support other types of layers and other types of data representations, like 16-bit floating point and 16-bit dynamic fixed-point.

ACKNOWLEDGMENT

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013.

REFERENCES

[1] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ser. ICCV '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 2722–2730.

[2] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, Dec 2015.

[3] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 26–35.

[4] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 16–25.

[5] Y. Qiao, J. Shen, T. Xiao, Q. Yang, M. Wen, and C. Zhang, "Fpga-accelerated deep convolutional neural networks for high throughput and energy efficiency," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 20, pp. e3850–n/a, 2017, e3850 cpe.3850.

[6] Z. Liu, Y. Dou, J. Jiang, J. Xu, S. Li, Y. Zhou, and Y. Xu, "Throughput-optimized fpga accelerator for deep convolutional neural networks," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 10, no. 3, pp. 17:1–17:23, Jul. 2017.

[7] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An opencl™deep learning accelerator on arria 10," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 55–64.

[8] S. I. Venieris and C. S. Bouganis, "fpgaconvnet: A framework for mapping convolutional neural networks on fpgas," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2016, pp. 40–47.

[9] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping cnn onto embedded fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 35–47, Jan 2018.