A FPGA Parameterizable Multi-Layer Architecture for CNNs

Guilherme Korol
PUCRS - Porto Alegre, Brazil
guilherme.korol@acad.pucrs.br

Fernando Gehm Moraes
PUCRS - Porto Alegre, Brazil
fernando.moraes@pucrs.br

ABSTRACT

Advances in hardware platforms boosted the use of Convolutional Neural Networks (CNNs) to solve problems in several fields such as Computer Vision and Natural Language Processing. With the improvements of algorithms involved in learning and inferencing for CNNs, dedicated hardware architectures have been proposed with the goal to speed up the CNNs' performance. However, the CNNs' requirements in bandwidth and processing power challenge designers to create architectures fitted for ASICs and FPGAs. Embedded applications targeting IoT (including sensors and actuators), health devices, smartphones, and any other battery-powered device may benefit from CNNs. For that, the CNN design must follow a different path, where the cost function is a small area footprint and reduced power consumption. This paper is a step towards this goal, by proposing an architecture for the main modules of modern CNNs. The proposal uses as case-study the Alexnet CNN, targeting Xilinx FPGA devices. Compared to the literature, results show a reduction up to 9 times in the amount of required DSP modules.

CCS CONCEPTS

• **Computer systems organization** → **Neural networks**; **System on a chip**;

KEYWORDS

FPGA, Convolutional Neural Networks, hardware accelerators, Alexnet.

ACM Reference format:
Guilherme Korol and Fernando Gehm Moraes. 2019. A FPGA Parameterizable Multi-Layer Architecture for CNNs. In *Proceedings of 32nd Symposium on Integrated Circuits and Systems Design, Sao Paulo, Brazil, August 26–30, 2019 (SBCCI '19),* 6 pages.
https://doi.org/10.1145/3338852.3339840

1 INTRODUCTION

The number of applications using CNNs is countless. The two most common fields of application are Natural Language Processing and Computer Vision. Convolutional Neural Networks have shown a great deal of improvement on audio, video, image classification, and segmentation to name a few. There are even networks that achieve better than human-level performance in some tasks, such as the ResNet that in 2015 achieved error rates lower than human experts in image classification [4]. Unfortunately, this performance comes at a cost. The algorithms employed by the most modern artificial neural networks require high memory bandwidth and extensive use of computational resources.

Memory is the first challenge to address when developing hardware for deep learning. A neural network consists of a set of neurons organized in layers. Each layer can be viewed as a series of multiply and accumulate operations between the input data and some parameters. It turns out that the number of layers is getting larger with each new network made public. From a hardware perspective, the growth in the number of layers means that the amount of data to be transferred from memory to the processing unit and then saved back to memory is also increasing. For instance, in 2012 Alexnet [7] won the Imagenet competition with a network of 60 million parameters to process a single image. Two years later in 2014, VGGNet [11] won the same contest, but instead, this network has seven times more parameters to load. Hence, it is expected that the architectures make good use (and reuse) of data to minimize the time consumed by memory transfers.

The extensive use of computational power comes from convolutions and fully connected layers. For instance, a convolution slides a window over the inputs, computing a sum of products several times. The fully connected layer can be viewed as a sequence of matrix multiplications, which can also be computationally challenging for any hardware implementation. Taking the networks previously cited, Alexnet has five convolutional layers and three fully connected layers while VGGNet consists of 16 convolutional layers and three fully connected layers.

There are three design paths to follow when developing hardware architectures for CNNs. The first one is adopting the use of GPUs [9], where developers implement a network by programming these and taking advantage of the high level of parallelism. The other two paths include FPGAs or ASICs. The former is the most flexible choice due to its reconfigurability, and the latter achieves the highest performance [8].

The *goal* of this paper is to present a parameterizable design for the main modules of modern CNNs. The cost function of the design is a small area footprint, by optimizing the usage of the arithmetic operations required in the convolutional layers. The motivation for proposing a lightweight design is to enable battery-powered embedded devices to take benefit from CNNs.

The *original contribution* of our work is in the detailing of the architecture, with the modules to access the memory and the way to execute the convolution through a pipeline that allows the reuse of arithmetic operators for different layers.

This paper is organized as follows. Section 2 presents the Alexnet CNN. Section 3 describes the hardware modules used in the implementation. Section 4 is the paper core, detailing the architecture of the generic convolutional layer. Section 5 reviews related works. Section 6 presents results and compare our proposal to the reviewed works. Finally, Section 7 concludes the paper.

SBCCI '19, August 26–30, 2019, Sao Paulo, Brazil

ACM ISBN 978-1-4503-6844-5/19/08…$15.00
https://doi.org/10.1145/3338852.3339840

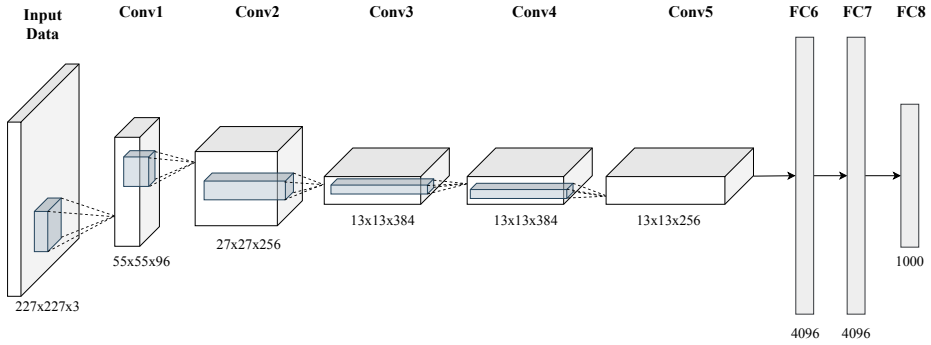

Figure 1: The Alexnet CNN [3].

2 THE ALEXNET CNN

The architecture chosen to be implemented in this work is the Alexnet [7]. There are two main reasons for this choice. First, the network is a state-of-the-art large-scale CNN [10]. Alexnet enables to exercise all the main operations of any modern CNN such as multiple layers of cascaded convolutions, fully connected layers, activation, and max-pooling layers. Second, the Alexnet is a popular choice of network to many authors who are working on dedicated hardware architectures, which makes it possible to compare our work to other implementations.

Alexnet classifies images using an eight-layer deep architecture, 5 of which are convolutional, and 3 of which are fully-connected. The input is a 227x227 pixels RGB image. Further, the output is a column vector containing the probability of each of the 1000 classes from the ImageNet dataset [7]. The network architecture cascades five convolutional (CONV) layers ending with three fully-connected (FC) layers. Table 1 details the shapes of each layer.

Table 1: Shapes for the Alexnet CNN.

Layer	Operation	Input Size (padded)	Weight/ Filter Size	Output Size
1	CONV	227x227x3	11x11x3 (x96)	55x55x96
1	MAX-POOL	55x55x96	3x3	27x27x96
2	CONV	31x31x96	5x5x96 (x256)	27x27x256
2	MAX-POOL	27x27x256	3x3	13x13x256
3	CONV	15x15x256	3x3x256 (x384)	13x13x384
4	CONV	15x15x384	3x3x384 (x384)	13x13x384
5	CONV	15x15x384	3x3x384 (x256)	13x13x256
5	MAX-POOL	13x13x256	3x3	6x6x256
6	FC	6x6x256	6x6x256x4096	4096
7	FC	4096	4096x4096	4096
8	FC	4096	4096x1000	1000

3 HARDWARE INFRASTRUCTURE

This section details the hardware modules that will be used in the various CNN layers. The hardware modules in this project should be as generic as possible. Through offline configuration, the designer can implement a CNN with any set of parameters and size.

3.1 Generic Memory Module

Memory modules are instantiated for both input and output feature maps, and weight values. The memory configuration uses two parameters: *depth*, corresponding to the amount of 1-bit wide memory blocks – B; *width*, corresponding to the word width – W. The width parameter enables to tweak the numeric precision with no need to rewrite memory modules.

The generic module, named **GenMem**, is configurable in depth ($B = \lceil nb\_words/32Kb \rceil$) to store the set of weights and feature maps that vary in size across layers. An important feature of the memory modules is the dual-port read access, enabling to read simultaneously two different data, feature explored in the proposed architecture. For instance, the Alexnet layer one input is a 227x227x3 tensor (Table 1), corresponding to 154,587 29-bit floating-point words, or 145 BRAM to store Alexnet layer one.

3.2 Floating-Point Format

Despite existing studies pointing out that the use of fixed-point has a little impact on the performance of deep learning accelerators [6], this work adopted floating-point representation to minimize the accumulated error between layers and to simplify the verification against a software implementation (Results, Section 6.1). Our work adopted the Flopoco framework for floating-point arithmetic [2]. Flopoco provides a list of configurable arithmetic operators tailored for FPGAs, along with its proper floating-point format.

The format used across all modules of this work is the Flopoco floating-point configured to an 8-bit exponent and 18-bit mantissa, as shown in Figure 2. The exponent width was chosen based on the IEEE-754 standard that uses an exponent of same size. However, the mantissa was set to the DSP block input width of the target FPGA [14]. It was necessary to fit the mantissa to the DSP block input, so that integer multiplications require a single block to be performed.

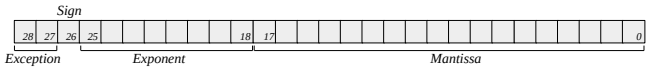

Figure 2: The adopted floating-point format.

3.3 Arithmetic Operators

The Flopoco framework offers an extensive list of operators [1]. This section only presents the ones used in this design.

3.3.1 Multiplier. Multiplication is at the core of convolutional and fully-connected layers. Once generated, the Flopoco floating-point multiplier was modified by instantiating a DSP Block. Afterwards, a control logic around the multiplier was implemented to

[1] http://flopoco.gforge.inria.fr

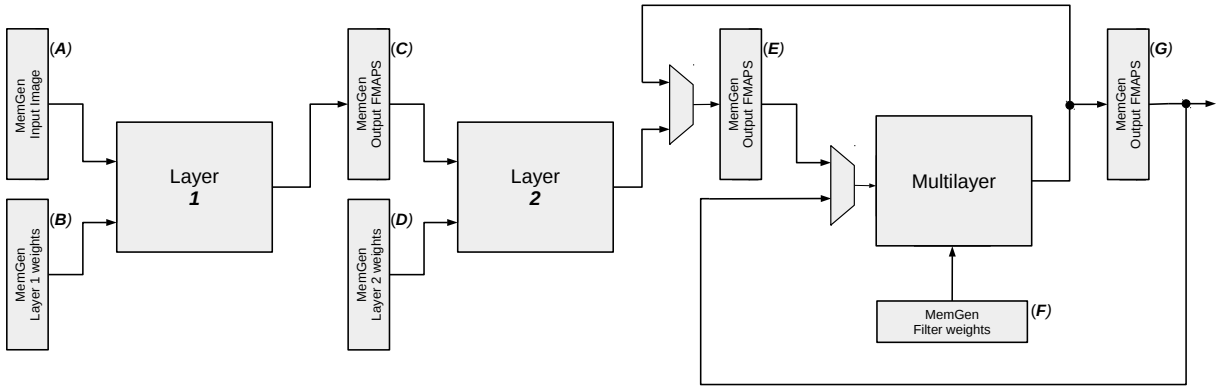

Figure 3: Architecture top view related to the convolutional layers.

control and synchronize the operation with other modules. The resulting module multiplies two floating-point numbers in three clock cycles. It requires 205 LUTs, 166 flip-flops, and a single FGPA DSP block.

3.3.2 Adder. Another important operation in CNNs is addition. Most modern accelerators and hardware implementations adopt multiply and accumulate (MAC) modules [12]. The advantage on using MACs is precision, because the accumulator comes right after the multiplication module, and the result of the first operation is not rounded. We adopted a different architecture in the convolutional layers, described in Section 4.1.2, based on a pipelined adder tree. The Flopoco dual path adder customized to this work performs a floating-point addition in six clock cycles, requiring 324 LUTs, 18 shift-registers (SRL) and 208 flip-flops.

3.3.3 Fixed-Point Converter. The third Flopoco operator used is a converter to fixed-point format. Conversion is necessary to perform comparisons in the max-pool layer. This layer first converts all the input values to fixed-point, then finds the maximum value among them by comparing their fixed-point representation. The conversion consists of a shifter, by the amount in the exponent, and a possible truncation. The fixed-point width was set to 32 bits to avoid overflow and underflow. Where the lower 15 bits are the fractional part, the next 16 bits represent the integer part, and the most significant bit gives the sign.

4 HARDWARE ARCHITECTURE

This Section first presents a top view of the architecture. Figure 3 presents the proposed architecture to implement each layer of the CNN. The left memory (*A)* contains the pre-processed input image and memory *B* contains the set of weights for the first layer. The first layer includes the convolutional and max-pool modules that fill the output feature map memory (*C*). Next, the second layer reads from memory *C*, executing the functions related to the second layer. Once layer two is completed, the *Multilayer* starts. The *multilayer* executes sequentially layers three, four, and five. Note that this *multilayer* structure enables to implement an arbitrary number of layers sequentially. The result of the fifth layer, its output feature map, is written into memory *G. This architecture is, in the Authors opinion, a relevant contribution, since it allows to implement CNNs with an arbitrary number of layers.*

The "Layer" encapsulates the convolutional and max-pool modules, interconnected in a ping-pong arrangement. The first advantage of using ping-pong buffers is memory saving, once it requires less memory space since only two channels need to be stored instead of the entire feature map. For example, the first layer channel size is 55x55 while the entire feature map contains 96 channels. The second advantage is the latency reduction since the max-pool layer starts its computations after each channel is computed, not waiting for the complete feature map be ready for consumption.

Next sections detail the modules responsible for performing the convolutional and max-pool layers, access to weight and feature map memories, as well as the *multilayer* module, which can execute an arbitrary number of convolutional layers sequentially, reducing the area footprint.

4.1 Convolution Implementation

The module responsible for executing the convolutions contains six functional blocks:

- **Address generator** block, generates addresses and control signals to read from or write into memory blocks: Input Feature Map (IFMAP) memory, weights memory, output channels.
- **Input buffers**, pair of shift registers that store one window of the feature map and data from the next stride positions (*stride buffer*), being their size equal to the convolution filter length. There is also an extra pair of buffers, responsible for storing the weights of two current filters.
- **FSMs**, control the convolutional tree and manage the write and read synchronization between the input buffers and the address generator. The FSMs need to be tightly synchronized to produce the right set of output values. This synchronization is necessary because the number of clock cycles taken by the convolutional tree, loading the current and stride buffers, and memory latency can vary.
- **Convolutional tree**, computes a convolution window, being also parameterizable according to the filter size. The design processes two windows in parallel, with the goal to speed up the inference. At the leaves, it multiplies the IFMAP with the weight array. Then, it adds the multiplication results using an adder tree. The output of the convolutional tree produces a value used by the **neuron adder**.

– **Neuron adder**, is in charge of summing up the results from the input channels belonging to the same output neuron, and to add the bias. The results from the first input channel are added with the bias, and the result is written to the OFMAP memory. The subsequent channels accumulate the result of the convolutional tree with the value previously stored in the OFMAP memory. This process avoids the extra step of the bias addition after the OFMAP completion.
– **ReLU**, only when the next layer reads from the output feature map (OFMAP), the **activation** function is executed. The address generated by the next layer is read from the memory, then, it passes through the activation module that performs the **ReLU** operation.

All internal modules are configurable and are instantiated using a set of user-defined VHDL *generics*, enabling the same VHDL entity to be used across all layers.

4.1.1 Buffers Operation. The convolution slides a window (the filter) through the IFMAP. This window reads a certain number of positions to execute the computation of the next neuron. The next window is read according to the stride value. Figure 4 shows the *current* and *stride* buffers structure configured to a window size of 3x3 and stride of 2. At the beginning of an operation (a new window or a new IFMAP), both buffers are empty. The *current* buffer is loaded with a window according to the addresses generated by the address generator block. After filling the *current* buffer, the FSM controlling the convolutional tree starts the computation. In parallel, the "Stride FSM" switches the IFMAP memory port to the *stride* buffer and the address generator starts to request the addresses for the new window. When the stride buffer is filled *and* the multiplications in the convolutional tree executed, the contents of the stride buffer are shifted to the current buffer, and the next window may start the execution. This process optimizes the loading from IFMAP by inserting shortcuts in the shift registers. This is possible by using a memory port that is idle during the arithmetical operations.

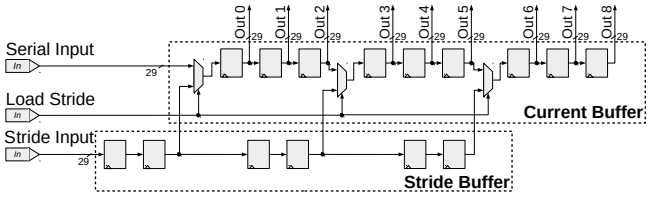

Figure 4: Current and stride buffers.

For instance, in the Alexnet first layer, the window size is 11x11 with a stride of 4. It means that 121 values need to be loaded from the IFMAP memory for the first window. For the next window, the addresses increment four positions to the right. Thus, instead of reading all 121 values, only 44 new values are read (4 * 11). This reduction represents 63.6% less memory reads in subsequent windows.

4.1.2 Convolutional Tree. Most of the literature in Deep Learning accelerators adopts systolic arrays to implement convolutions. The approach has the advantage of a high level of parallelism. However, it requires a considerable amount of resources.

An architectural option to systolic arrays is the multiply-add trees (Figure 5). This approach uses a line of multipliers and an adder tree [16]. The inputs of the multiply-add tree are the *current buffer* and the weight array. Next, the result produced by each pair of multipliers are added in the next level. The result of the adders in the first level travels in the tree, up to the root, producing the final result. The size of the adder tree is configured through VHDL generic parameters.

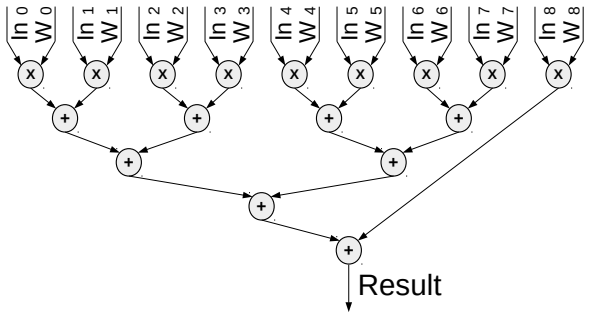

Figure 5: Example of a 3x3 pipelined multiply-add tree. Each layer has an input register.

The multiply-add tree is pipelined. The multiplication takes 3 clock cycles and each addition 6 clock cycles. Thus, for layer one there are 121 multipliers in parallel (plus 7 dummy modules to obtain a full binary tree), and 8 adder levels ($log_2 n + 1$). The time to compute one 11x11 convolution in the current implementation is 51 clock cycles. The convolutional tree behaves as a standard pipeline, with each stage of the tree computing the convolution of a given window.

4.2 Max-Pool Layer

The Max-pool Layer makes use of a single FSM that controls a shift register input buffer and the max-pool tree. A process generates the address of each window loading the input buffer. When the window is ready (buffer filled), the FSM starts the max-pool tree. After the maximum value in the window is found, it is written to the OFMAP memory.

The input values of the max-pool tree are converted to a fixed-point format (as presented in Section 3.3.3). Then, the tree of comparators evaluates all values. It is important to note that the fixed-point numbers are not re-converted to floating-point. Instead, they are only used to execute the comparisons that will select the output value. Hence, no precision is lost, and the comparator can be simplified to work with integer operands.

4.3 Multilayer

Following the first two Alexnet layers, the Multilayer module implements layers three, four, and five. As shown in the top view (Figure 3), the Multilayer feedbacks itself creating a loop that could implement any number of subsequent convolutional layers using the same filter size. Even that the Multilayer could be used to implement all five convolutional layers on Alexnet, the architecture dedicates two Layer modules for the first two convolutional layers, which have filter sizes of 11x11 and 5x5 respectively, while the remaining layers, executed by the multilayer, have 3x3 filters. The hardware implementation as it is depends on the off-line configuration to instantiate the convolutional tree. An improvement could be made to re-configure at run-time the convolutional tree, so it would be possible to execute convolutions of different filter sizes.

Another important aspect of the Multilayer module is that it saves FPGA resources since only one module is capable of performing multiple CNN layers.

The Multilayer encapsulates a convolutional module, the ping-pong buffer that interfaces with a max-pool or a bypass module, and writes the result in the OFMAP memory. Additionally, there is a control FSM that coordinates the two multiplexers seen in top view.

In Figure 3, the second layer writes its results into memory *E*. Next, when the Multilayer starts the computation of the third layer the FSM sets the multiplexers in such a way that the Multilayer reads the IFMAP from memory *E* and writes the OFMAP to memory *G*. As soon as the third layer finishes, the FSM re-configure both multiplexers, so the Multilayer starts to perform the fourth convolutional layer, reading the IFMAP from memory *G* and writing the resulting OFMAP to memory *E*. Finally, the fifth layer can start its execution. The FSM returns to the initial configuration, where the IFMAP comes from memory *E*, and the OFMAP goes to memory *G*. Additionally, the FSM manage the load of the filters' weights for each layer (memory *F*).

5 RELATED WORK

Zhang et al. proposed a method that chooses the best optimization parameters (such as loop tiling and unrolling) using the roofline model [16]. The architecture, configured by an FPGA embedded processor, executes the CNN layers serially performing ping-pong buffer operations between the external and internal memories to load weights and feature maps. A set of parallel multipliers followed by a group of adder trees are the core of the authors' computation engine. The accelerator achieved a 61.62 GFLOPS on a Virtex7 VX485T at 100 MHz.

In contrast, authors in [10] proposed a pipelined architecture capable of running all layers in the device. First, the weights are read from the external memory; next, the computation engine is made of a set of Processing Elements (PEs). A number of input feature maps are processed in parallel by a group of PEs, and then an adder tree adds the result from all PEs concerning the same receptive region. The architecture, which uses a 16-bit fixed precision, executes the Alexnet at a performance of 565.4 GOPS (156 MHz).

Caffeine was proposed in 2016 [15], providing full integration with the software framework Caffe [5]. In addition, Zhang et al. focused on optimizing memory bandwidth requirements. The proposed method involves a memory reorganization, implicating in fewer memory accesses during convolutions. Also, the architecture is fully tunable, including numeric precision and number of PEs in the systolic array. Running the Alexnet in a Virtex7 690t, the authors were able to achieve 636 GOPS at 150 MHz (16-bit fixed precision).

Aydonat et al. presented an architecture that, not only, made used of Discrete Fourier Transform to reduce the number of multiplications and additions in convolutional layers, but also, the 16-bit format encompasses shared exponent techniques. The architecture was synthesized from a software description in OpenCL. Implemented on an Arria 10 device, the overall system performance achieved 1382 GFLOPS (303 MHz) running the Alexnet CNN [1]. A similar approach was taken by Wang et al. [13], where a quantization method was applied to weights and feature maps. Additionally, the convolutional layers were implemented following a Finite Impulse Response filter implementation, which reduced the number of the necessary multiplications and additions. This implementation was able to run the VGG-16 CNN with 1250.21 GFLOPS performance at 170 MHz.

The primary goal of these works is performance, measured in GOPS or GFLOPS. Next section evaluates the performance of our proposal, putting it into perspective the area used regarding DSP modules, showing that our goal, lightweight implementation, is achieved but with performance sacrifice.

6 RESULTS

This section first describes the method to assess the accuracy of the proposed hardware. Next, the main result, which is the area evaluation with its comparison w.r.t. related works. Finally, the throughput of convolutional layers is presented.

6.1 Accuracy of the Hardware Implementation

A software implementation, the Tiny-Dnn Framework [2], was used for verification. Output feature maps produced during the hardware simulation were compared against the software-generated ones. The verification is performed in three steps. First, a dump file from the hardware simulation is processed, converting the 29-bit floating-point numbers into their decimal representations. Second, it is selected in the C++ source code to write its values to a file, from the equivalent point in the CNN to where the hardware simulation values were taken. Last, both files are compared for errors.

Consequently, a small error between the software and the hardware implementation is observed. It comes from the numeric representation (C++: float single-precision, hardware: 29-bit floating-point format). Such validations were performed across all layers during development. For instance, the relative error observed up to the fifth convolutional layer is below 0.01%, demonstrating the *accuracy* of the proposed hardware architecture.

6.2 Resource Utilization

The design was synthesized using the Xilinx Vivado framework. Table 2 gives the resource utilization for the convolutional layers. The number of BRAMs is high because the weights were statically stored. In the final design, a fetching mechanism will load the weights from an external memory, reducing this usage. The LUTs and register usage come from the design complexity and the fact that the floating-point additions do not use DSP modules, which is an optimization to consider in future implementations. Also, if the target is an ASIC implementation, the adder-tree may be optimized with multiple input carry-save adders, resulting in a small silicon area. The most relevant result is related to the number of used DSPs: 310 (2x11x11 in the first layer, 2x5x5 in the second layer and 2x3x3 in the Multilayer). This result is relevant since it corresponds to the number of required multipliers by the CNN, and thus the final silicon area.

Table 3 presents the performance (second column) and resources utilization (third and fourth column). Besides the similar number of LUTs in most designs, the proposed design requires 7 to 10 fewer DSP blocks than the related works. These results corroborate for the goal of a CNN architecture with a *small area footprint.*

[2] https://github.com/tiny-dnn/

Table 2: Resource utilization for the 5 convolutional layers, XC7VX690T device.

Resource	Used	Available	Utilization (%)
Clock	1	32	3.13
BRAM	1,247	1,470	84.83
Slice LUTs	199,174	433,200	45.98
Slice Registers	144,256	866,400	16.65
DSPs	**310**	3,600	21.18

Table 3: Comparison of the proposal w.r.t the state-of-the-art.

Work	GOPS/GFLOPS	LUTs	DSPs
Wang et al., 2018 [13]	1,250.21	N.A.	2296
Aydonat et al., 2017 [1]	1,382.00	246,000	1476
Zhang et al., 2016 [15]	636.00	300,000	2833
Zhang et al., 2015 [16]	64.62	186,251	2240
Li et al., 2016 [10]	565.94	233,805	2144
This proposal	2.46	199,174	**310**

6.3 Performance Results

The hardware description was simulated to assess the performance and synthesized targeting a frequency of 250 MHz. Table 4 presents the number of clock cycles to execute each layer (3^{rd} col.), and the number of floating-point operations (4^{th} col.). As expected, the execution time is dominated by the convolutions. Considering the layers working as a pipeline, the throughput is a function of the slower stage, in this case, the Multilayer (layers 3 to 5), corresponding to 2.46 GFLOPS@250MHz, or 1 image every 0.87 sec.

Table 4: Number of clock cycles and floating point operations for a forward execution (convolutional layers only).

Layer	Operation	Clock Cycles	FP oper. 10^6
1:	Loading Input image	154,587	
2:	CONV Layer 1	23,134,272	210.83
3:	Max-Pool Layer 1	8,028	0.92
4:	CONV Layer 2	89,235,456	895.79
5:	Max-Pool Layer 2	1,868	0.58
6:	CONV Layer 3	62,668,800	299.04
7:	By-pass Layer 3	24,597	0.65
8:	CONV Layer 4	94,003,200	448.56
9:	By-pass Layer 4	676	0.65
10:	CONV Layer 5	62,668,800	299.04
11:	Max-Pool Layer 5	404	0,13
	Total:	**331,900,688**	**2,155.02**

The obtained throughput, compared to the state-of-the-art, may be considered low. Nonetheless, note that: (*i*) the convolution throughput is a function of two convolutional trees, which in the best case generate a new data every 3 clock cycles (6 cycles for each tree). On the other side, related works use arrays of PEs at each layer to increase the throughput; (*ii*) the design choice to reduce area, through the Multilayer, serialized layers 3 to 5.

Thus, these results comply with our goals, demonstrating the feasibility of implementing a CNN with low area consumption and accuracy equivalent to a software implementation of reference.

7 CONCLUSIONS

This work proposed a parametrizable architecture capable of executing multiple CNN convolutional layers. The work presented a detailed hardware architecture, being reproducible by other designers. Despite the adoption of Alexnet as a case-study, the hardware can be configured to other CNNs by modifying the set of parameters. The implementation of the first two layers showed that a set of layers could be pipelined in the presence of available resources on the target FPGA. On the other hand, the Multilayer module enables the designer to implement an arbitrary number of convolutional layers with a limited amount of resources. Results show that the hardware design is suitable for battery-powered embedded devices, due to the smallest number of DSPs.

As ongoing work, we enumerate: (*i*) finish the interface with the external SRAM to feed the weights to the memories; (*ii*) implement the fully-connected layers with a similar architecture to the convolutional layers; (*iii*) evaluate the power.

ACKNOWLEDGMENTS

Author Fernando Gehm Moraes is supported by FAPERGS (17/2551-196-1) and CNPq (302531/2016-5), Brazilian funding agencies.

REFERENCES

[1] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. 2017. An OpenCL™ Deep Learning Accelerator on Arria 10. In *International Symposium on Field-Programmable Gate Arrays, FPGA*. 55–64.
[2] Florent de Dinechin and Bogdan Pasca. 2011. Designing Custom Arithmetic Data Paths With FloPoCo. *IEEE Design & Test of Computers* 28 (2011), 18–27.
[3] Xiaobing Han, Yanfei Zhong, Liqin Cao, and Liangpei Zhang. 2017. Pre-Trained AlexNet Architecture with Pyramid Pooling and Supervision for High Spatial Resolution Remote Sensing Image Scene Classification. *Remote Sensing* 9, 8 (2017), 848.
[4] Kaiming He, Xiangyu Zhang, et al. 2016. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. 770–778.
[5] Yangqing Jia et al. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *International Conference on Machine Learning, ICML*. 675–678.
[6] Li Jiao, Cheng Luo, Wei Cao, Xuegong Zhou, and Lingli Wang. 2017. Accelerating Low Bit-width Convolutional Neural Networks With Embedded FPGA. In *FPL*. 1–4.
[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification With Deep Convolutional Neural Networks. In *Conference on Neural Information Processing Systems, NIPS*. 1106–1114.
[8] Ian Kuon and Jonathan Rose. 2007. Measuring the Gap Between FPGAs and ASICs. *IEEE Trans. on CAD of Integrated Circuits and Systems* 26, 2 (2007), 203–215.
[9] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.
[10] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. 2016. A High Performance FPGA-based Accelerator for Large-scale Convolutional Neural Networks. In *FPL*. 1–9.
[11] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014), 1–14.
[12] Vivienne Sze, Yu-Hsin Chen, et al. 2018. Hardware for Machine Learning: Challenges and Opportunities. In *CICC*. 1–8.
[13] Jichen Wang, Jun Lin, and Zhongfeng Wang. 2018. Efficient Hardware Architectures for Deep Convolutional Neural Network. *IEEE Trans. on Circuits and Systems* 65-I, 6 (2018), 1941–1953.
[14] Xilinx Inc. 2018. 7 Series DSP48E1 Slice. (March 2018). https://www.xilinx.com/support/documentation/\user_guides/ug479_7Series_DSP48E1.pdf.
[15] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2016. Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. In *ICCAD*. 12.
[16] Chen Zhang, Peng Li, et al. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *FPGA*. 161–170.