A Convolutional Neural Network Fully Implemented on FPGA for Embedded Platforms

Marco Bettoni*, Gianvito Urgese*, Yuki Kobayashi†, Enrico Macii*, and Andrea Acquaviva*
* Politecnico di Torino, Torino, Italy, 0039 011 090 7042. Email: gianvito.urgese@polito.it
† NEC Corporation, Kawasaki, Japan. Email: y-kobayashi@hq.jp.nec.com

***Abstract*—Convolutional Neural Networks (CNNs) allow fast and precise image recognition. Nowadays this capability is highly requested in the embedded system domain for video processing applications such as video surveillance and homeland security. Moreover, with the increasing requirement of portable and ubiquitous processing, power consumption is a key issue to be accounted for.**

In this paper, we present an FPGA implementation of CNN designed for addressing portability and power efficiency. Performance characterization results show that the proposed implementation is as efficient as a general purpose 16-core CPU, and almost 15 times faster than a SoC GPU for mobile application. Moreover, external memory footprint is reduced by 84% with respect to a standard CNN software application.

I. Introduction

In this paper, we propose the design of a hardware architecture implementing a customizable *Convolutional Neural Network* (CNN) framework where several CNN schemas can be configured and executed. We analyzed the CNN computational flow for identifying the most critical points to be parallelized in the FPGA implementation. We described the CNN framework architecture using a High Level Synthesis (HLS) language and tested the new *HW-CNN* module on an *Altera Stratix V FPGA* embedded in a *Terasic DE-5-Net* board.

The CNN algorithm performs fast and precise image recognition, which is a highly requested feature in the context of embedded systems. The biggest involvement of this type of algorithms can be found in the Artificial Intelligence field, bringing contribution to numerous applications, such as in fire detection in forests [1], robotics [2], autonomous driving [3] and mobile applications [4]. In this latter domain, battery lifetime and memory resources are a serious concern for CNN implementations.

Several CNN models are available in the literature for general purpose applications. In 2012, Alex-Net Model [5] has been the first efficient application of the CNN and lately more accurate models have been proposed, such as GoogLeNet [6] featuring the *Inception* concept and Fast R-CNN [7] with advanced capabilities for detecting the position of the subject in the picture.

For teaching the CNN to recognize defined objects, the network needs to be *Trained*. During the *Training Phase*, a set of labeled images is used for generating the set of parameters to be applied in the neural network. By means of the *Test Phase*, the capability of the network to identify and classify the pictures is evaluated.

The chosen model defines how to perform the training and the computation involved during the test, specifying parameters and functions to be used for CNN recognition. The CNN flow is summarized as follows:

- *Convolution*: The matricial convolution operator is applied over the *feature maps* of the Input image, such as the RGB color channels. The computation is shown in Equation 1, where O is the number of Output feature maps of size $H \times W$, I is the number of Input feature maps, and $K \times K$ is the size of the *Kernel*, which is the convolution operand obtained from the training.

$$Out[O][H][W] = \Sigma_{i=0}^{I}\Sigma_{kh=0}^{K}\Sigma_{kw=0}^{K} In[i][H+kh][W+kw] \times Kernel[O][i][kh][kw] \quad (1)$$

- *Activation*: A threshold function which is applied on the convolution output. The $ReLU(x) = max(0,x)$ function is widely adopted, but others are common as well, such as *Tanh* and *Sigmoid*.
- *Pooling*: The average or maximum value over an input region is evaluated, generating a resized image representative of the pool values. Equation 2 shows the *Pooling by Average* operation, where $Ph \times Pw$ is the pooling window size.

$$Out[O][H][W] = (\Sigma_{ph=0}^{Ph}\Sigma_{pw=0}^{Pw} In[O][H+ph][W+pw])/Ph/Pw \quad (2)$$

- *Fully-Connected* (FC): Implemented at the end of a CNN, the FC layers provides the classification of the features extracted by convolution. The FC layers are implemented as in Equation 3:

$$Out[O] = \Sigma_{i=0}^{I} In[i] \times Kernel[O][i] \quad (3)$$

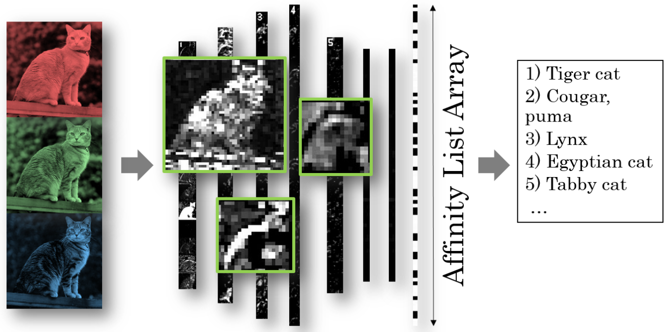


Fig. 1: Convolution process representation. The magnifications are representative of the CNN edge-detection.

The Alex-Net model is used as a reference in this work, since it performs an accurate recognition (84.7% Top-5 accuracy) and it is generally used as a benchmark for CNN implementations. This model includes 5 Convolutional Layers followed by 3 FC Layers, and makes use of both Activation and Pooling, requiring in total nearly 1.5 billions operations.

Figure 1 shows an example of the Alex-Net model used to set-up our HW-CNN architecture running on the FPGA. The RGB channels of an image are provided as inputs and processed by the following 8 layers. The intermediate images are shown, highlighting the edge-detection capability of the CNN.

A common approach for CNN acceleration exploits GPU cards, able to perform recognition over several hundreds of images per second [8]. An alternative state-of-art approach leverages on FPGA for matricial convolution acceleration while the other computation steps are performed on a general purpose CPU [9].

In this paper we proposed a CNN fully implemented in FPGA, which executes Convolution, Activation, Pooling and FC layers. More specifically, the proposed solution named HW-CNN has the following characteristics:

- Standalone Implementation, (no GPU, no CPU);
- Power efficient CNN computation;
- Low FPGA resources and memory dependency;
- Software reprogramming for CNN model compliance.

To characterize the proposed implementation, we performed comparative performance and power evaluations against a software version running on a general purpose CPU and a mobile SoC GPU. Memory utilization results are also reported. Overall, the results show that the proposed implementation is power efficient and lend itself to be adopted in mobile application with stringent power and resource requirements. The rest of the paper is organized as follows: a description of the internal implementation (Section II), the performance and obtained results (Section III) and finally the conclusion (Section IV).

II. Implementation

The developed architecture can be configured for execute all the steps of a CNN model defined by users. Thus, it can be entrusted for computing recognition of a picture or a video-stream. The HW-CNN allows great compatibility with any CNN model because the CNN parameters can be reconfigured in software. For computing a classification, the input image is passed from an external communication interface (LAN or Serial connection). Then, the HW-CNN compute all the CNN steps generating a list of recognition decision that is sent back to the host. The recognition is completely performed on the FPGA, which requires a DDR-RAM to store the raw input image and the intermediate results.

A. FPGA Implementation

We developed the HW-CNN implementation using the NEC *CyberWorkBench HLS compiler* (CWB) [10], exporting the component described in pseudo-C to an RTL format.

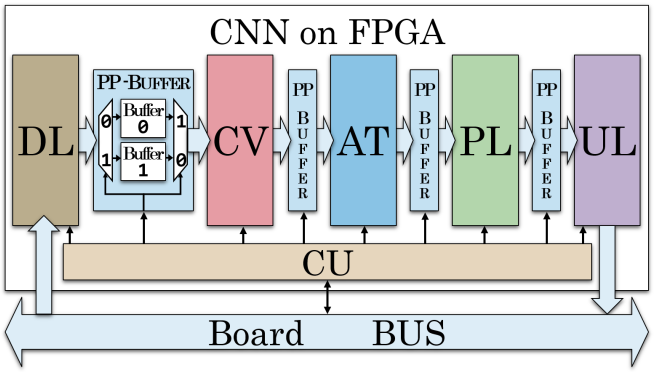


Fig. 2: CNN Pipeline. The data-flow pass through the DL, CV, AT, PL, UL Units, all controlled by the CU. A double buffering system is implemented as *PP buffers*, allowing concurrent operation on the picture tiles.

We implemented a parallel version of a general and customizable CNN architecture. For this purpose, we used two parallelization techniques: the *Tiling Technique* designed by Zang et al. [9] and the *Pipelining Technique* commonly implemented for data stream elaborations.

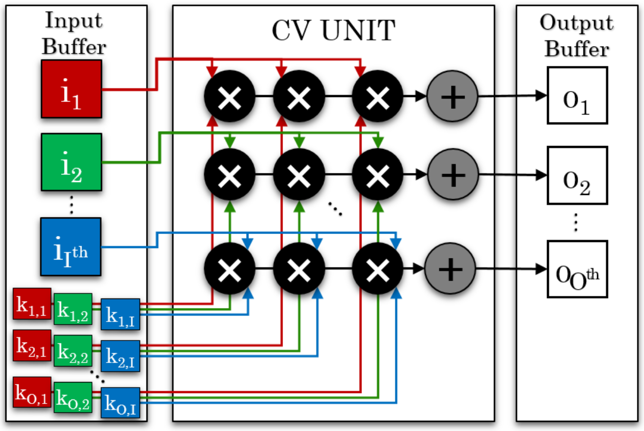
The Tiling Technique has been exploited to overcome the data dependency of the CNN calculation, which, due to the massive recurrence of the data, does not allow to fit the FPGA internal memory. The input image is therefore fragmented in tiles smaller than the original picture, and the CNN can performed by computing each tile individually. The biggest advantage of performing the tiling technique on an FPGA is the significant parallelization degree achievable by computing several tiles efficiently on the hardware logic. In our HW-CNN implementation, up to 8 tiles of size 32×32 pixels are computed in parallel.

The Pipeline is composed of 5 units: *Download* (DL), *Convolution* (CV), *Activation* (AT), *Pooling* (PL) and *Upload* (UL), which perform the homonyms functions. The scheduling of the function is managed by the *Control Unit* (CU), which generates the parameters for each unit depending on the CNN-Model size and structure.

This configuration is shown in Figure 2. The computation flow passes through all the units from DL to UL, where the extremes are dedicated to the data transfer with the on-board RAM. In order to avoid the *data hazard* existing between two consequent unit, a *double buffering* [11] system has been adopted, implementing the *Ping-Pong Technique* already exploited in [9]. The buffer duplication prevents the units to read uncompleted data, or to update data which has not been elaborated yet. In the successive pipeline stage, the CU unit swaps the data by means of control logic depicted in the *Ping Pong* (PP) Buffer of Figure 2.

B. Convolution Unit

The core of the CNN computation is the CV Unit, where both Convolution and Fully-Connected layers are computed.



Input Buffer
CV UNIT
Output Buffer
i_1
i_2
$i_{I^{th}}$
o_1
o_2
$o_{O^{th}}$
$k_{1,1}$ $k_{1,2}$ $k_{1,I}$
$k_{2,1}$ $k_{2,2}$ $k_{2,I}$
$k_{O,1}$ $k_{O,2}$ $k_{O,I}$

Fig. 3: Convolution Unit (CV) schema. The i_I matrices on the left side are the *Input tiles*, the o_O matrices in the right side are the *Output tiles* and the $K_{o,i}$ are the *Kernel matrices*.

The convolution operator requires to perform a considerable amount of MAC operations, which is proportional to the Neural Network size and the image resolution. The HW-CNN implementation optimizes this computation by parallelizing 24 times the MAC operator.

In Figure 3 is represented the CV schema of the HW-CNN implementation, where it is possible to notice the MAC parallelization of I Input tiles and O outputs.

The CV Unit has been efficiently optimized by an internal scheduling which avoids idleness among the CV components. In fact, for the MAC computation, data must be acquired by the input buffer, and after the operation, stored in the output buffer. The process has been internally pipelined for guaranteeing that the *address calculation*, *MAC execution*, *buffer reading* and *writing* were performed in a single clock cycle. This internal pipelining has been efficiently coded by means of the CWB *automatic pipelining* feature, and proved on the real hardware with the scheduling shown in Figure 4.

The CV operation is repeated for each pixel that compose the Output tile. The loop logic has been hard-coded for the $T_{width} \times T_{height}$ dimension of the Output tile. Eventually, Equation 4 computes the number of clock cycles necessary to the CV unit to complete a CNN stage, which depends on the kernel size $K \times K$, the Output tile size and the CV internal pipeline latency T_{pl}.

$$CV_{time} = T_{pl} + K^2 \times T_{width} \times T_{height} \qquad (4)$$

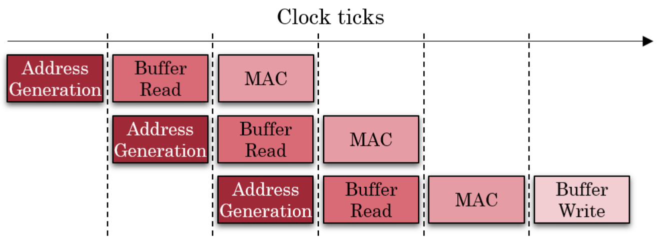


Fig. 4: Convolution Unit internal pipeline.

III. Results and Discussions

We configured the HW-CNN with the Alex-Net model structure and parameters used in the Cong work [9]. The following evaluation has been compared with a software implementation, since the FPGA-based implementation [9] found in literature reports only performances the CNN steps computed on FPGA and not considering all the other steps executed on the host. Thus making impractical a direct comparison between the two architectures. We tested the performances by comparing different CNN software implementations: an optimized C code designed by Zhang et al. [9], the Caffe Python Library working in CPU-only mode (without GPU parallelization) [12] and, for the GPU mobile comparison, the *clBLAS OpenCL* library which has been evaluated by Lokhmotov et al. [13].

A. Timing and Power Results

For the *timing* performances, we considered the time required to compute an image recognition. The time required by our HW-CNN is reported in Figure 5a where it is compared with the Caffe execution over an *Intel Xeon CPU E5-2630 v3 @ 2.40GHz 32-CPU* and with the clBLAS library tested on an *ARM Mali-T628 GPU*.

For the *power* comparison, the *Performance per Watt* unit has been used, a value obtained by the Equation 5 where the number of operations executed by the device is considered, altogether with the Thermal Device Power in Watt.

$$\text{Performance per Watt} = \frac{Operations}{Time \times Power} \qquad (5)$$

The aim of this comparison is to give an idea of the different timing performances, considering the discrepancies in terms of hardware and level of portability. We designed the HW-CNN module with the clear intent to reduce at minimum the hardware requirements, while relaxing the timing constraint to a reasonable value, still bearable for the mobile user. On the other side, the FPGA adoption brings a speed-up by almost 15 times over the SoC GPU.

Table I reports the CNN power efficiencies of the considered implementations. The general purpose CPU values have been extracted from [9], where the results has been obtained by executing an optimized CNN code on an *Intel Xeon CPU E5-2430 (@2.20GHz)*.

This comparison shows that running the Alex-Net Model on the FPGA using our HW-CNN architecture is as power efficient as the CPU implementation running on 16 threads and almost 3 times more efficient than the software execution without parallelization. The comparison with mobile

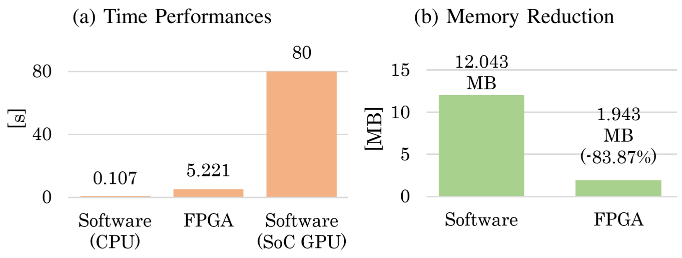


Fig. 5: In 5a are compared the recognition times over a CPU, on the FPGA implementation and a SoC GPU, highlighting the performance cost for the sake of portability. 5b compares the RAM memory requirements, where the FPGA makes efficient use of on-chip data compared with the other software implementations.

TABLE I: Power performances.

Device	OP/s [GOP/s]	Power [W]	**Perf. per Watt** [GOP/s/W]
GPU Mobile OpenCL [13]	0.02	3	0.007
CPU single thread [9]	3.54	95	0.037
CPU 16-threads [9]	12.87	95	0.135
FPGA HW-CNN [*This work*]	0.75	5.54	0.135

hardware shows that the current mobile implementation of CNN are outperformed by the FPGA by more than a 19× factor. This demonstrates the effectiveness of the low-power FPGA computation, and the possibility to adopt similar CNN implementation for mobile applications where battery life is the major constrain.

B. Resource usage

The percentage of required FPGA resources is reported in Table II. The synthesis report shows that few resources have been implemented, allowing the architecture to fit more compact FPGAs, such as the Xilinx Zynq. The most required is the on-chip memory, which has been exploited for the main purpose of caching the intermediate CNN results on FPGA.

TABLE II: FPGA Resources.

Resource	**Stratix V - FPGA Chip Usage**
Logic	65,463 ALMs (28%)
Register	3.5kB (3%)
DSP	104 blocks (41%)
Memory	4,752kB (73%)

The memory necessary on the on-board DDR RAM is greatly reduced, as it is shown in Figure 5b. The chart reports the comparison of RAM memory requirements for performing the Alex-Net model on both FPGA and software. The values are motivated by the fact that during the CNN execution, the intermediate results are stored in the PP buffers along the pipeline, rather than transferred to the RAM memory.

The reduced amount of external Memory and FPGA resources are significant figures for encourage the adoption of *HW-CNN*-like architectures in mobility applications. While this allows the implementation to be placed on smaller FPGA or silicon component, the architecture can be easily adapted to fit the smallest FPGA devices, exploiting the versatility of the HLS synthesis coupled with the modular programming technique.

IV. Conclusion

In this paper we propose a Convolutional Neural Network (CNN) fully implemented in FPGA, that enables image recognition in low-power embedded systems with limited resources. This features have been made feasible by extending state-of-the-art implementations where only the convolution step is accelerated on FPGA, with the modular addition of extra functionalities. This modularity guarantees compliance with existing CNN models, but also the possibility to easily introduce new functionalities.

The experiments show that our HW-CNN can quickly perform image recognitions, outperforming the reference SoC GPU in both terms of timing (15×) and power (16×). The proposed implementation is even 3 times more power efficient with respect to the reference CPU, and equivalently efficient to the same CPU parallelized 16 times. Moreover, the requirement of an external memory has been reduced by 83%, when compared to the software version of CNN.

Finally, this architecture has been designed to allow software reconfiguration, which allows the user to apply various CNN model and to efficiently test the same picture against different trained data and recognized classes.

Acknowledgments

The HLS compiler and the technical support was provided by NEC Corporation, Japan.

References

[1] Qingjie Zhang et al. "Deep Convolutional Neural Networks for Forest Fire Detection". In: *2016 International Forum on Management, Education and Information Technology Application*. Atlantis Press. 2016.

[2] Lei Tai and Ming Liu. "Deep-learning in Mobile Robotics-from Perception to Control Systems: A Survey on Why and Why not". In: *arXiv preprint arXiv:1612.07139* (2016).

[3] Mariusz Bojarski et al. "End to end learning for self-driving cars". In: *arXiv preprint arXiv:1604.07316* (2016).

[4] Ryosuke Tanno and Keiji Yanai. "Caffe2C: A Framework for Easy Implementation of CNN-based Mobile Applications". In: *Adjunct Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing Networking and Services*. ACM. 2016, pp. 159–164.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[6] Christian Szegedy et al. "Going deeper with convolutions". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 1–9.

[7] Shaoqing Ren et al. "Faster r-cnn: Towards real-time object detection with region proposal networks". In: *Advances in neural information processing systems*. 2015, pp. 91–99.

[8] NVIDIA. *GPU-Based Deep Learning Inference*. URL: https://www.nvidia.com/content/tegra/embedded-systems/pdf/jetson_tx1_whitepaper.pdf (visited on 01/18/2017).

[9] Chen Zhang et al. "Optimizing fpga-based accelerator design for deep convolutional neural networks". In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2015, pp. 161–170.

[10] Kazutoshi Wakabayashi. "CyberWorkBench: Integrated design environment based on C-based behavior synthesis and verification". In: *VLSI Design, Automation and Test, 2005.(VLSI-TSA-DAT). 2005 IEEE VLSI-TSA International Symposium on*. IEEE. 2005, pp. 173–176.

[11] Wikipedia. *Multiple buffering*. URL: https://en.wikipedia.org/wiki/Multiple_buffering (visited on 02/28/2017).

[12] Evan Shelhamer Yangqing Jia. *Caffe - Deep learning framework by the BVLC*. URL: http://caffe.berkeleyvision.org (visited on 01/20/2017).

[13] Anton Lokhmotov and Grigori Fursin. "Optimizing convolutional neural networks on embedded platforms with OpenCL". In: *Proceedings of the 4th International Workshop on OpenCL*. ACM. 2016, p. 10.