2016 IEEE International Parallel and Distributed Processing Symposium Workshops

On the Automation of High Level Synthesis of Convolutional Neural Networks

Emanuele Del Sozzo, Andrea Solazzo, Antonio Miele, Marco D. Santambrogio

Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Milano, Italy
andrea.solazzo@mail.polimi.it
{emanuele.delsozzo, antonio.miele, marco.santambrogio}@polimi.it

***Abstract*—Convolutional Neural Networks (CNNs) are a particular type of Artificial Neural Networks (ANNs) inspired by cells in the primary visual cortex of animals, and represent the state of the art in image recognition and classification. Nowadays, such supervised learning technique is very popular in Big Data analytics. In this context, due to the huge amount of data to be processed, it is crucial to find techniques to speed up the computation. In particular, the dataflow pattern of CNN algorithm results to be suitable for hardware acceleration. This paper proposes a framework to automatically generate a hardware implementation of CNNs on Field Programmable Gate Arrays (FPGAs), based on the High Level Synthesis (HLS) of configurable offline-trained networks.**

***Index Terms*—Field Programmable Gate Arrays, Convolutional Neural Networks, High Level Synthesis**

I. Introduction

In the deep learning context, Convolutional Neural Networks (CNNs) [1, 2] are the most used algorithm for image classification. CNNs are a Supervised Learning algorithm derived from Artificial Neural Networks (ANNs) that apply convolutional filters on images to extract features, which are then used for classification. Indeed, CNNs are based on visual mechanisms of living organisms, in particular on the complex arrangement of cells (called *receptive fields*) in animals primary visual cortex.

CNNs are the state-of-the-art solution in image recognition and classification, and they are widely used in several fields, like video surveillance, mobile robot vision, Big Data analysis, etc. [3, 4, 5, 6]. Hence, in all these scenarios, CNNs must provide considerably high performance. In fact, in many embedded and mobile applications, such video surveillance or robot vision, the system has to guarantee a minimum throughput in order to process the input frames that use to arrive with a strict rate. On the other hand, in server applications, such as Big Data analysis, the challenging factor for the system to provide a response in a reasonable time is the huge amount of data to be processed.

For this reason, in the last years, different works have proposed to accelerate CNNs on Graphic Processing Unit (GPU), Field Programmable Gate Array (FPGA) and even Application Specific Integrated Circuit (ASIC) [7, 8, 9] in order to achieve high performance in terms of throughput or turnaround time. Indeed, the acceleration of such algorithms in hardware has obtained particularly promising results due to their computational pattern, which is mainly dataflow-based and presents very few control structures. However, the process of designing and implementing a CNN on hardware can be very complex and requires a lot of effort in terms of design time. In fact, the CNN algorithm presents different levels of customization in terms of the depth and the width of the network structure as well as the integrated classification kernels, which require many parts of the hardware module to be reimplemented for every design.

For this reason, this paper proposes an automated framework based on a web-application for the generation and synthesis of a hardware implementation of a CNN on Xilinx FPGA devices given in input a high level specification of an already-trained network. More in details, the proposed framework produces a synthesizable C++ code that is transmitted to a High Level Synthesis (HLS) tool called *Vivado HLS* [10], and then employed to generate bitstream file using *Vivado Design Suite* [11]. To this end, the framework also provides the *tcl* scripts for Vivado HLS and Vivado to automatize such process. Finally, the framework allows the user to customize the design of the equivalent CNN, and choose the target platform (the currently available platforms are Zybo [12] and Zedboard [13]).

It is clear that, in order to tune and train the CNN instance, in particular to assign a value to the various weights, it is necessary to implement at least a software version of the algorithm. Hence, one may argue that since we already have the source code of a software version of the network, it is useless to have another framework automatically generating the C++ code of the same application. The motivations are the following. First, HLS tools deal only with a small set of programming languages (C, C++, etc.), while CNN and, in general, machine learning frameworks use many different languages; for instance, *Torch* [14] framework employs *LuaJIT* scripting language. Second, even though the CNN is written in C/C++ language, this does not imply that the network is synthesizable. Indeed, HLS tools only accept a subset of C/C++ structures. This fact, in addition to the need of enhancing the performance of classification phase using hardware accelerators, represents the rationale of the proposed work.

The paper is organized as follows. Section II discusses related work and Section III gives the necessary background on CNN algorithm. Then, in Section IV the proposed framework is presented and it is employed in a set of case studies in sub-

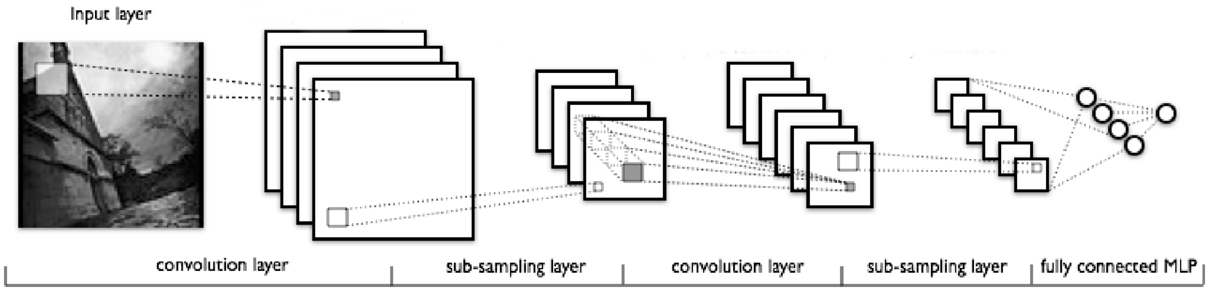


Fig. 1: Convolutional Neural Network structure

sequent Section V. Finally, Section VI draws the conclusions and sketches the future works.

II. Related Work

CNNs were originally proposed by Professor Yann LeCun in the late 1990s [1]. The original purpose of CNNs was the recognition of handwritten letters and digits. Thanks to improvements in technology and algorithms, it is now possible to build and train large neural networks able to identify different and more complex features in images. Nowadays, CNNs are employed in many fields with impressive results. For instance, in [3] a 3D CNN model for human action recognition has been developed, in [5] CNNs were used for image classification, while authors in [15] presented a natural language processing engine based on CNNs.

The working flow of CNNs is composed of a training phase and a classification phase. Although the former phase is mainly control dependent and therefore a software implementation of the CNN may be more versatile to perform the tuning of the weights, recent works proposed hardware acceleration of this phase based on both GPU [16, 17, 18] and FPGA implementations [19]. On the other hand, the nature of the classification phase well suits a hardware acceleration. As result, many works [7, 8, 9], like the one we propose, present implementations on hardware accelerators like FPGAs.

Peemen et al. worked on a memory-centric design method for CNN accelerators on FPGAs [7]. The authors showed that, thanks to an efficient data pattern access, it is possible to minimize on-chip memory size and increase data reuse. As result, the performance rises, while area and energy usage decrease. Such design flow was evaluated on a Xilinx Virtex-6 FPGA board, and overcame the performance of standard scratchpad memories accelerators, while using the same amount of reconfigurable resources.

Sankaradas et al. proposed a massively parallel coprocessor for accelerating an entire CNN [8]. Such coprocessor has two key features: it is coupled with off-chip memory with a large bandwidth, and it packs multiple data words into each memory operation (low data precision is used). The prototype of the whole coprocessor was mapped to Xilinx Virtex-5 FPGA on an off-the-shelf Peripheral Component Interconnect (PCI) card with 1GB DDR2 memory. Finally, the prototype resulted to be 31x faster than a software implementation on a 2.2 GHz AMD Opteron processor, reaching 3.4 Giga Multiply-Accumulate operations (GMACs).

Zhang et al. presented an analytical design scheme for exploration of FPGA-based CNN design space [9]. The proposed analysis is based on *roofline model* [20], a model that estimates the performance of a hardware platform by combining system performance peak and off-chip memory bandwidth. Thanks to this approach, the authors were able to efficiently explore the design space and identify the right solution, in terms of both performance and FPGA resource usage. The resulting implementation outperformed the previous approaches by reaching a peak performance of 61.62 GFLOPS under 100MHz working frequency.

To the best of our knowledge, there are no available frameworks able to generate, starting from network weights, a synthesizable equivalent CNN that can be easily accelerated on FPGA. Hence, as stated previously, this work may result useful to network designers for implementing and accelerating their networks on hardware.

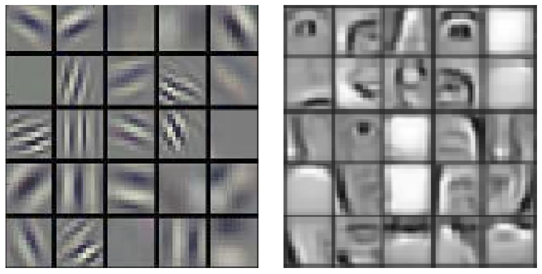
III. Background on Convolutional Neural Networks

The main structure of a CNN is the one shown in Fig. 1. Typically, a network is made of an arbitrary number of layers belonging to two main typologies: *convolutional layers* and *linear layers*. Moreover, the convolutional layers are commonly alternated with sub-sampling layers. For instance, the example in Fig. 1 is composed by two convolutional layers alternated with two sub-sampling layers and, in the end, a linear layer, implemented by *Multi-Layer Perceptron (MLP)*. The main features and the mathematical formulas implemented by each layer are discussed in the following subsections.

A. Convolutional Layer

Convolutional layers consist of K weighted filters, also called *kernels*. At a high level of abstraction, each kernel is swiped across the image and produces the corresponding output called *feature map*. The weights of the kernels are used to compute the convolution of the inputs. The values of those parameters are determined and tuned during the training phase, which will be briefly described later. Each element of the k feature maps is calculated with the following equation:

$$o_{k,i,j} = \sum_{m=0}^{M} \sum_{n=0}^{N} (w_{k,m,n} \cdot x_{i+m,j+n}) + b_k \tag{1}$$



(a) Simple filters (b) Complex filters

Fig. 2: Convolutional filters

where *M* and *N* are the width and height of the kernel, respectively. In this way, the dimension of the resulting *feature maps* will be reduced according to the following formulas:

$$width_{new} = width_{old} - width_{kernel} + 1 \tag{2}$$

$$height_{new} = height_{old} - height_{kernel} + 1 \tag{3}$$

At early stages, trained kernels are able to “extract” features like lines, circles, arcs etc. The extracted features become more sophisticated at the final layers of the convolutional part of the network, which are capable of catching “complex” objects like eyes, faces, wheels etc. (as shown in the example in Fig. 2).

Finally, in order to emphasize relevant features, the non-linearity of the output can also be increased; this operation is performed by the Rectified Linear Unit (ReLU) layers and it can be implemented with different kinds of functions like the hyperbolic tangent or the sigmoid function.

B. Sub-Sampling Layer

Sub-sampling layers, instead, are used to reduce the amount of data to be stored in memory, and to forward only features that are relevant for classification. Sub-sampling is implemented in the same way of convolution, i.e. with filters swiped over the input. Moreover, there are different types of sampling, like Mean-pooling, which computes the average value of sub-matrices to generate a point in the new image, or Max-pooling, which computes the value of the point in the new image as the maximum in the corresponding input sub-matrix. As for the convolution, the dimension of the output of the sub-sampling layers will be reduced by a quantity determined by the following formulas:

$$width_{new} = \left\lfloor \frac{width_{old} - width_{kernel}}{p_{step}} \right\rfloor + 1 \tag{4}$$

$$height_{new} = \left\lfloor \frac{height_{old} - height_{kernel}}{p_{step}} \right\rfloor + 1 \tag{5}$$

where p_{step} is the amplitude of the sliding of the pooling kernels.

C. Linear Layer

This type of layer, called *perceptron*, is composed of simple neurons. They are responsible of putting together the information collected by the convolutional part, and predicting the class of the initial input image. The output elements of these layers are calculated with the following equation:

$$o_j = \sum_{i=0}^{I} (w_i \cdot x_i) + b_j \tag{6}$$

which is basically a weighted sum of the *I* outputs of the layer $j - 1$.

The last layer of the linear part of the network contains as many neurons as the classes to be recognized. Usually, it is followed by the *LogSoftMax* operator, which normalizes results in the following way:

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad for\ j = 1, \ldots, K \tag{7}$$

LogSoftMax enforces the K values of the output vector **z** to lie in range $[0, 1]$ and to sum up to 1, such that they can be interpreted as the probability of the input to belong to a certain class (i.e. the maximum probability).

As conclusion, from this brief overview of CNNs, it is possible to note how the overall structure is highly regular and the internal implementation of the various layers is mainly based on a pure dataflow structure with a sequential and regular control flow. As result, CNNs are particularly targeted for a hardware implementation and also for a short high level model to be automatically translated into a hardware module by means of a HLS process without any peculiar design activity.

IV. The Proposed Automation Framework

The proposed work consists in a easy-to-use framework that allows to design and configure a CNN by means of a web-based Graphical User Interface (GUI), and to specify in input the file containing the trained weights; then, the framework returns the equivalent and synthesizable network and tcl scripts for the Xilinx tools as output. Fig. 3 shows the working flow of the proposed application.

The framework requires the input network to be already designed and trained so that the user can provide the related weights to generate the executable model and the hardware specification. Nevertheless, this aspect does not represent a limitation affecting the purpose of the framework. Indeed, the literature presents various state-of-the-art frameworks (such as Torch [14]), specifically targeted for machine learning applications, that allow to design and tune CNNs with limited effort. Once trained, the network configuration (i.e. the weights) can be exported. Moreover, if the user is interested only in the evaluation of the performance of the desired network, he/she can also directly use the proposed automation framework to generate the related hardware implementation by specifying random weights for the sake of simplicity and by temporarily ignoring the correctness of the prediction. Indeed,

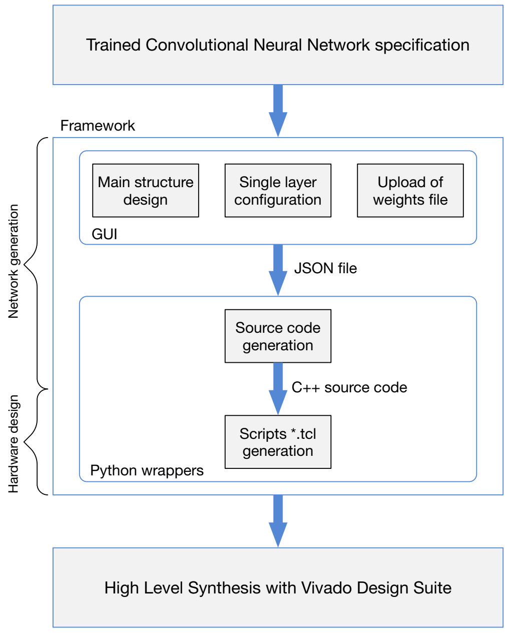
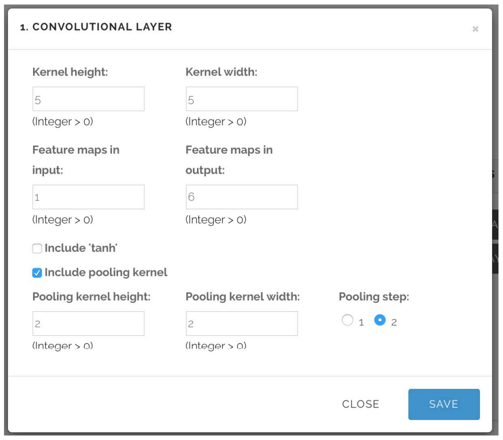


Fig. 3: Workflow of the framework


1. CONVOLUTIONAL LAYER

Kernel height: 5 (Integer > 0)

Kernel width: 5 (Integer > 0)

Feature maps in input: 1 (Integer > 0)

Feature maps in output: 6 (Integer > 0)

☐ Include 'tanh'

☑ Include pooling kernel

Pooling kernel height: 2 (Integer > 0)

Pooling kernel width: 2 (Integer > 0)

Pooling step: 1 [✓ 2]

CLOSE SAVE

Fig. 4: Convolutional layer options

the generated network is likely to provide a significantly high prediction error, but, in terms of hardware implementation and employed resources, there is no difference with a network built using trained weights (clearly assuming that both networks share the same structure).

The various steps of the presented working flow are discussed in details in the following subsections.

A. Network Generation

The framework has been developed as a web-application to be easily accessible. The client-side was implemented in HTML5 and Javascript. As shown in the above part of Fig. 3, the GUI of the framework requires first the user to specify the main structure of the CNN, by providing the number of convolutional and linear layers, and the dimension of the input data. Then, the GUI supports the user in the configuration of each layer per se with the following options:

- For each convolutional layer, the user can specify the number and the dimension of kernels in it (*Feature maps out*). Moreover, a sub-sampling layer featuring a Max-pooling algorithm can also be integrated and configured within this stage (as shown in Fig. 4).
- For each linear layer, the user can specify the number of neurons in the network and whether or not to include the hyperbolic tangent function at the end of it.

Finally, the GUI requires to select the board the CNN module will be synthesized for. At the moment, the framework supports Xilinx Zybo [12] and Zedboard [13] platforms.

At this point, the application creates a *JSON* file containing all the parameters specified by the user. This descriptor is transmitted to a back-end module, implemented into two Python wrappers, responsible of producing both the C++ source code of the network and the configuration tcl scripts to be used for the synthesis of the hardware module.

The first wrapper generates the C++ source code of the network. The output consists of a single file containing all the parameters of the network, included the hard-coded weights, and the function that will be implemented in hardware. This function is implemented in a subset of C++ compliant with the synthesis rules of Vivado HLS, and consists in a sequence of blocks of instructions corresponding to the layers.

Moreover, the wrapper adds by default at the end of the function the code block implementing a *LogSoftMax* operator to normalize the outputs, as discussed in Section III. Finally, the generated function returns an integer value representing the index of the predicted class of the image.

Once the source code has been generated, the second wrapper is called in order to build the tcl files to be used by Xilinx Tools. In particular, the wrapper parses a template of tcl files and specifies the values for the parameters specified by the user. In the end, both the source code and the tcl files are returned to the user, that can proceed with the subsequent synthesis of the hardware implementation. Currently, the execution of those scripts with Vivado tools has to be performed manually by the user due to license management issues. In the future we plan to extend the web-application to support both the synthesis and the bitstream generation automatically.

B. Hardware Design

The tcl scripts are used to synthesize an IP Core from the CNN source code, and then generate the bitstream of the whole hardware design for the selected platform. In particular, the framework produces three different tcl files:

- `cnn_vivado_hls.tcl`
- `directives.tcl`
- `cnn_vivado.tcl`

The first two scripts are used by Vivado HLS, while the last one by Vivado Design Suite.

Vivado HLS tool accelerates IP Core generation starting from C, C++ and System C specifications. The scripts mainly define how the IP Core has to be generated and the directives

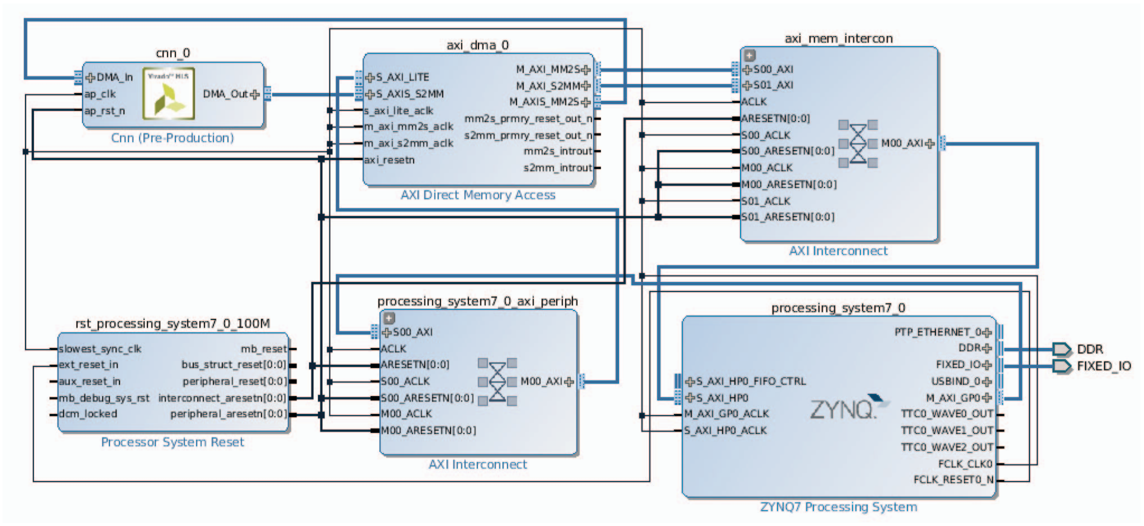


Fig. 5: Block design

related to hardware interfaces and optimization. The C++ source code follows a dataflow pattern, where data pass through intermediate buffers used as output for layer *i* and as input for layer *i+1*. The only exception to this pattern is the first layer, which takes inputs in a streaming fashion by using an *AXI4-Stream* connection. Such interconnection is employed to transfer data to the IP Core (images to be classified) and to return the classification value to the CPU. This has been realized by mean of internally developed high level wrappers to facilitate the communication with the Direct Memory Access (DMA) [21]. Once the HLS is done, the IP Core is ready to be imported into Vivado Design Suite.

The `cnn_vivado.tcl` script creates the block design reported in Fig. 5. The proposed design is composed of the following list of blocks:

- ZYNQ7 Processing System,
- AXI DMA,
- 2 AXI Interconnect,
- Processor System Reset,
- CNN IP Core.

The ZYNQ7 Processing System (a hardwired ARM dual-core processor) transfers the data to the AXI DMA through the AXI Interconnect. In particular, the ZYNQ7 Processing System exploits the AXI high performance slave interface for the data transfer. Then, the CNN IP Core receives the stream of data from the DMA, and, once the computation is completed, it returns the classification value to the DMA, which will then send it back to the processor.

The script is in charge of instantiating all the necessary blocks and properly connecting them. Once the block design is complete, the script validates it, creates the HDL wrapper, and finally starts the synthesis flow towards the bitstream generation. Thus, the produced bitstream can be directly downloaded on the target device.

V. Experimental Results

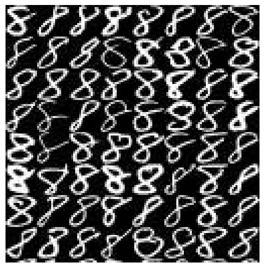
We employed the proposed framework in a set of case studies to show its effectiveness; in this section, we present the experimental results. The tests were performed by implementing different CNNs on the Zedboard Evaluation Board; the board is powered by the Xilinx Zynq-7000 All Programmable System on Chip (APSoC), which features the Dual ARM Cortex-A9 MPCore and Series-7 Programmable Logic, while the hardware synthesis and the bitstream generation have been performed with Vivado Design Suite 2015.2. We then compared the software and hardware implementations of such case studies in terms of execution time, predicted error and energy consumption. Finally, we analyzed the FPGA resource utilization of each hardware implementation.

The measures of the energy/power consumption of the various hardware and software implementations were performed by using the following means. Since the board does not provide any interface for measuring the power consumption of the Zynq chip, we sensed the overall board by means of an external device, that is *Energy Logger 4000* by Voltcraft [22]. Then, to divide the contribution of the reconfigurable logic and the one of the hardwired ARM multiprocessor, we esteemed the power consumption of the FPGA by using Vivado power analysis tools with the default setting, and we computed the other term as the difference of the previous two values. Finally, the energy consumption has been obtained by integrating the average power consumption in the time.

The training phase of each CNN has been done by using the Torch scientific framework, which provides wide support for machine learning algorithms [14]. The datasets used for training and testing are:

- the handwritten digits automatically scanned from envelopes by the U.S. Postal Service (USPS);
- the CIFAR-10 dataset [23].

Such datasets are often used for image recognition purposes, hence they result suitable for testing our framework (Fig. 6


(a) USPS dataset

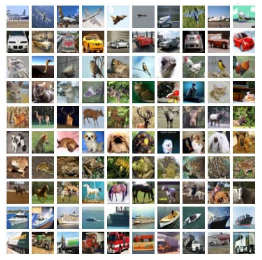

(b) CIFAR-10 dataset

Fig. 6: Examples of images from datasets

displays instances taken from those datasets).

Finally, software and hardware implementations employ 32-bit floating point weights. From the FPGA prospective, this reasonably implies a higher usage of resources, but, on the other hand, it reduces the prediction error and makes the hardware solution prediction similar to the software one.

A. *Test 1: Naive CNN Implementation*

In a first test, we employed USPS dataset and designed a network composed of two layers, one convolutional and one linear, configured as follows:

- The convolutional layer takes 16x16 grayscale images as input. It is composed of six 5x5 filters applied on the input image; the resulting feature maps are then sub-sampled using the Max-pooling operator with 2x2 kernels.
- The linear layer is made of 10 neurons corresponding to the digits from 0 to 9.

The obtained hardware implementation is a naive one; therefore we used none of the possible optimization available in Vivado HLS.

As reported in the first line of Tab. I, hardware implementation is slightly faster than software one (1.18X speedup). Moreover, both implementations produce the same prediction error (the test set is composed of 1000 images). This fact was not as immediate as it may seem; indeed, software and hardware implementations of certain mathematical functions (e.g. exponential, logarithm) could be different, and, consequently, they could condition the final output. This was not the case, which means that hardware implementation is as accurate as software one. In terms of power consumption, the software implementation (i.e. the CPU only) consumes 2.2W, while, the hardware implementation (i.e. CPU and FPGA) overall consumes 4.19W. As results, even though the hardware implementation has (slightly) better performance than the software one, this does not justify the usage of FPGA, since the hardware solution energy consumption is 1.6X the energy consumption of the software one.

Finally, the first line of Tab. II reports the resource usage. We can notice that DSP slices are the most used resources (41.82%), which is due to the mathematical functions we implemented, while the other resources usage is quite low. This fact has two consequences: first, our approach has a reasonable impact on FPGA resources, which means that it can scale with the size of the network; second, we can exploit the other resources to improve our implementation to both deliver higher performance and reduce energy consumption.

B. *Test 2: Optimized CNN Implementation*

In this second test, we considered the same CNN of the previous test, and the same dataset as well. However, in this case we optimized the hardware implementation of the network, by using some Vivado HLS directives in order to reduce the latency of the synthesized IP Core. In particular, the Vivado directives we applied are:

- `HLS DATAFLOW`
- `HLS PIPELINE`

The former enables task level pipelining, allowing functions and loops to execute concurrently, while the latter reduces the initiation interval by allowing the concurrent execution of operations within a loop or function. More in detail, the `HLS PIPELINE` directive was applied to the inner loop of convolutional layer. In this way, the IP Core latency was remarkably reduced.

We then compared again the execution time of software implementation with this improved hardware implementation (on the same test set). As result, the execution time on FPGA decreased to 0.53s, which implies a 6.23X speedup (Tab. I), while the predicted error did not change. Moreover, thanks to such speedup, hardware solution results to be more convenient than the software one also in terms of energy consumption. Indeed, the adopted optimization has a slight impact on FPGA power consumption, but guarantees enough performance to overtake software implementation energy efficiency.

Looking at the resources usage (Tab. II), as expected the utilization percentage rose in most of the resources (with the exception of flip-flops). This is the consequence of the optimization directives we enforced. However, many resources are still available, hence there is room for bigger networks.

C. *Test 3: Optimized Implementation of a Larger CNN*

The third test presents a bigger network for the USPS dataset composed of three layers, two convolutional and one linear. While the first convolutional layer is equal to the one of the previous network, the second convolutional layer takes as input the Max-pooling outputs (six 6x6 feature maps) and applies sixteen 5x5 kernels. The result are sixteen 2x2 feature maps. On the other hand, the linear layer is the previous one.

Like in the other tests, the hardware implementation on FPGA outperforms software implementation on ARM Cortex-A9 in terms of execution time (as reported in Tab. I). Indeed, FPGA takes 0.48s to classify the test set (1000 images), while CPU requires 4.3s to complete the process (9.0X speedup). In addition, while the energy consumption of the software implementation rises due to the higher execution time (9.46J), the hardware solution results to be more energy efficient than the previous test (2.04J).

It is interesting to notice that the two implementations provide a prediction error (7.1%) that results to be higher with respect to the previous networks. One might believe that the introduction of a second convolutional layer implies a lower

TABLE I: Hardware implementation vs. software one

Test	Dataset	Predicted Error		Execution Time		Speedup	Power		Energy	
		Software	Hardware	Software	Hardware		CPU	CPU + FPGA	Software	Hardware
Test 1	USPS	3.9%	3.9%	3.3s	2.8s	1.18X	2.2W	4.19W	7.26J	11.73J
Test 2	USPS	3.9%	3.9%	3.3s	0.53s	6.23X	2.2W	4.21W	7.26J	2.23J
Test 3	USPS	7.1%	7.1%	4.3s	0.48s	9.0X	2.2W	4.24W	9.46J	2.04J
Test 4	CIFAR-10	89.4%	89.4%	2565s	223s	11.5X	2.2W	4.37W	5643J	975J

TABLE II: FPGA resources usage

Test	Resources				
	Flip-Flops (106400)	LUT (53200)	Memory LUT (17400)	BRAM (140)	DSP Slices (220)
	Utilization	Utilization	Utilization	Utilization	Utilization
Test 1	15.86%	2.56%	2.56%	6.43%	41.82%
Test 2	8.86%	17.18%	3.38%	7.14%	44.09%
Test 3	9.32%	18.10%	3.06%	9.29%	46.36%
Test 4	10.39%	20.25%	3.13%	76.07%	48.64%

prediction error. However, this is not the case; indeed, as it happens for other machine learning algorithms, increasing the complexity of a CNN may have a negative impact on the prediction error, since the new network may overfit the training set and, as consequence, worsen the prediction on the test set.

Finally, even though the network has more layers than the previous ones, the bigger size has an almost negligible impact on FPGA resources, as reported in Tab. II. Indeed, the utilization percentage slightly grew for all the resources, with the exception of *Resource Memory LUT*.

D. Test 4: CNN Implementation on a More Complex Dataset

In the last test, we designed a network for the CIFAR-10 dataset. The generated network is structured as follows:

- The first convolutional layer takes 32x32 RGB images as input, and applies twelve 5x5 filters. Max-pooling operator sub-samples the resulting features maps using 2x2 kernels.
- The second convolutional layer takes the output of the first layer (twelve 14x14 feature maps) and applies thirty-six 5x5 kernels. Sub-sampling is performed again using Max-pooling operator (2x2 kernels).
- The first linear layer is composed of 36 neurons, while the second one of 10 neurons, which correspond to the 10 classes of the dataset.

For the sake of simplicity, we used random weights to build the network. This represents a reasonable choice since we were more interested in the performance of our framework rather than in the prediction error; in fact this last one is out of the focus of this work, being a consequence of the training phase. To have a fair comparison, the only important aspect we ensured was to achieve the same prediction error in all the implemented versions of the CNN.

We evaluated this network on a test set of 10000 images. Also in this case, the hardware implementation overtakes the software one; indeed, we reached a 11.5X speedup with respect to the software implementation, in terms of execution time (Tab I). In addition, the hardware implementation is more energy efficient than the software one (975J against 5643J), even though this FPGA implementation consumes more power than the previous ones (2.17W). As expected, the prediction error is remarkably high, but the important fact is that both implementations provide the same error (89.4%).

Finally, as consequence of the fact that we are dealing with 32x32 3-channel images, it is necessary to employ more resources to store the weights of the network. In particular, the BRAM utilization increased to 76.07%, while the utilization of all the other resources slightly rose.

E. Final Remarks

As final remarks, we want to highlight some important aspects. The decision of using a programming language like C++ and a HLS tool, instead of other solutions (e.g. parametric VHDL model) derives from the need of a flexible and scalable approach. Indeed, Vivado HLS, along with a high level specification, allows to explore faster the design space and analyze different solutions (in terms of optimization) in a agile way, and finally converge to the most suitable implementation for that particular task. We followed this approach in order to come up with the Vivado optimization directives we applied to the tested hardware implementations. As consequence of the achieved performance results, we decided to include such optimization directives in the C++ source code generation.

Finally, the proposed framework is available at the following link: http://cnn2fpga.hosting.necst.it.

VI. Conclusions and Future Work

This paper presented a preliminary framework for the automation of HLS of CNNs. In particular, starting from the weights of a trained network, the framework allows the user to customize the design of an equivalent CNN, and it automatically generates both a synthesizable C++ code of the network and ready-to-use scripts for Vivado Design Suite. The hardware implementation was tested on Xilinx Zedboard, and the experimental results reported higher performance, in terms of execution time and energy efficiency, with respect to the ARM Cortex-A9 multiprocessor. Besides, the restrained FPGA resource usage allows this solution to support bigger networks.

As stated in the previous sections, this work facilitates the whole procedure of implementing by hand a CNN on hardware. Indeed, even though it is necessary to implement the CNN itself for the training phase, this does not imply that the resulting network may be synthesized on hardware with no effort. Moreover, given the dataflow nature of the feed-forward computation of CNN algorithm, a hardware implementation is ideal to improve performance.

Currently, the framework supports Xilinx Zybo and Zedboard platforms, but we plan to extend it also to other boards like Xilinx Virtex-7. On the other hand, we intend to add the possibility to online sign up to the framework; in this way, provided a regular Vivado License from the user, it would be possible to execute on the server all the steps up to the bitstream generation on user's behalf. Finally, the framework will be expanded by adding more configuration options (e.g. other sub-sampling operators like Mean-pooling) and the possibility to train the designed CNN online with Torch framework, provided the dataset for training and testing.

Acknowledgment

This work was supported by the European Commission in the context of the H2020 FETHPC EXTRA project (#671653).

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] P. Y. Simard, D. Steinkraus, and J. C. Platt, "Best practices for convolutional neural networks applied to visual document analysis," *2013 12th International Conference on Document Analysis and Recognition*, vol. 2, p. 958, 2003.

[3] S. Ji, W. Xu, M. Yang, and K. Yu, "3D convolutional neural networks for human action recognition," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 35, no. 1, pp. 221–231, 2013.

[4] Y. LeCun, K. Kavukcuoglu, and C. Farabet, "Convolutional networks and applications in vision," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, May 2010, pp. 253–256.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[6] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

[7] M. Peemen, A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, Oct 2013, pp. 13–19.

[8] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, "A massively parallel coprocessor for convolutional neural networks," in *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*. IEEE, 2009, pp. 53–60.

[9] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "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] Xilinx Inc., "Vivado HLS." [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[11] ——, "Vivado Design Suite." [Online]. Available: http://www.xilinx.com/products/design-tools/vivado.html

[12] ——, "Zybo Zynq^{TM}-7000 Development Board." [Online]. Available: http://www.xilinx.com/products/boards-and-kits/1-4azfte.html

[13] "Zedboard." [Online]. Available: http://zedboard.org/product/zedboard

[14] "Torch Framework." [Online]. Available: http://torch.ch

[15] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th International Conference on Machine Learning*, ser. ICML '08. New York, NY, USA: ACM, 2008, pp. 160–167. [Online]. Available: http://doi.acm.org/10.1145/1390156.1390177

[16] D. Strigl, K. Kofler, and S. Podlipnig, "Performance and scalability of GPU-based convolutional neural networks," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, Feb 2010, pp. 317–324.

[17] D. C. Ciresan, U. Meier, J. Masci, L. Maria Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," in *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, vol. 22, no. 1, 2011, p. 1237.

[18] NVIDIA, "Accelerate Machine Learning with the cuDNN Deep Neural Network Library." [Online]. Available: http://devblogs.nvidia.com/parallelforall/accelerate-machine-learning-cudnn-deep-neural-network-library/.

[19] K. Samal, "Fpga acceleration of cnn training," 2015.

[20] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[21] "ZedBoard Linux DMA driver." [Online]. Available: https://github.com/durellinux/ZedBoard_Linux_DMA_driver

[22] "Voltcraft." [Online]. Available: http://www.voltcraft.com

[23] "CIFAR-10." [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html