2017 IEEE International Parallel and Distributed Processing Symposium Workshops

A Pipelined and Scalable Dataflow Implementation of Convolutional Neural Networks on FPGA

Marco Bacis, Giuseppe Natale, Emanuele Del Sozzo, Marco Domenico Santambrogio
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milano, Italy
marco.bacis@mail.polimi.it
{giuseppe.natale, emanuele.delsozzo, marco.santambrogio}@polimi.it

***Abstract*—Convolutional Neural Network (CNN) is a deep learning algorithm extended from Artificial Neural Network (ANN) and widely used for image classification and recognition, thanks to its invariance to distortions. The recent rapid growth of applications based on deep learning algorithms, especially in the context of Big Data analytics, has dramatically improved both industrial and academic research and exploration of optimized implementations of CNNs on accelerators such as GPUs, FPGAs and ASICs, as general purpose processors can hardly meet the ever increasing performance and energy-efficiency requirements. FPGAs in particular are one of the most attractive alternative, as they allow the exploitation of the implicit parallelism of the algorithm and the acceleration of the different layers of a CNN with custom optimizations, while retaining extreme flexibility thanks to their reconfigurability.**

In this work, we propose a methodology to implement CNNs on FPGAs in a modular, scalable way. This is done by exploiting the dataflow pattern of convolutions, using an approach derived from previous work on the acceleration of Iterative Stencil Loops (ISLs), a computational pattern that shares some characteristics with convolutions. Furthermore, this approach allows the implementation of a high-level pipeline between the different network layers, resulting in an increase of the overall performance when the CNN is employed to process batches of multiple images, as it would happen in real-life scenarios.

***Index Terms*—Field Programmable Gate Arrays, Convolutional Neural Networks, Dataflow Architectures**

I. Introduction

Inspired by biological vision systems, Convolutional Neural Network (CNN) is a well-known deep learning algorithm extended from Artificial Neural Network (ANN) that has demonstrated significant success in various applications, such as image search engines in data centers, video surveillance and mobile robot vision [1, 2, 3, 4], achieving far higher accuracy than traditional algorithms for computer vision. Indeed, CNNs are currently considered the state-of-the-art solution for visual recognition and classification, and are attracting a growing interest from both academia and industry [5, 6]. In a nutshell, CNN applies consecutive convolution filters on the input image in order to extract features at increasing levels of abstraction. Then, such features will be used for the classification.

As the CNN models improve and their accuracy increases, so network size and computational complexity are rapidly growing. Thus, the current requirements in terms of both performance and energy-efficiency can be hardly met by general purpose processors, effectively shifting the research interest towards more specialized hardware accelerators, such as GPUs, FPGAs, and even ASICs, which have been indeed widely explored in the last years to optimize CNN designs [7, 8, 9, 10]. In particular, FPGA-based accelerators recently gained substantial attention due to their advantages of performance and high energy-efficiency, as well as a lower power consumption compared to GPUs, and a higher flexibility than ASICs, thanks to their reconfigurability [11, 12, 13, 14].

Previous work on FPGA-based implementation of CNNs mostly focus on the acceleration of the convolutional layer [10, 14, 12], being the most compute-intensive layer, or propose solutions that do not fully exploit the suitability of CNNs for a dataflow implementation [15, 16]. In the first case, the proposed approaches introduce a non-negligible communication overhead due to the need for data exchange between accelerated and unaccelerated layers, effectively diminishing the overall performance gains. In the second, the absence of a pure dataflow implementation results in an underutilization of the inherent parallelism of FPGAs, a suboptimal exploitation of the on-chip memory, and the inability to process batches of multiple images efficiently, which is instead the common use case in real-life scenarios.

In this work, we propose a methodology to accelerate CNN designs on FPGAs with a dataflow implementation, in a modular, scalable way. The approach builds upon previous work on the acceleration of Iterative Stencil Loops (ISLs) [17, 18], a computational pattern that shares some characteristics with convolutions, and adapts both the ISL-centric proposed hardware accelerator and its concepts to suit the implementation of CNNs. In particular, the paper provides the following contributions:

- An acceleration methodology of CNN designs on FPGAs that exploits the dataflow computation pattern, scales up the size of each layer from single-input-port/single-output-port to fully parallel if enough resources are available, and is able to implement a high-level pipeline between the different network layers;
- An experimental evaluation of the proposed methodology with the implementation of 2 CNNs of different sizes on a Xilinx Virtex-7 FPGA, trained and tested respectively with the USPS and CIFAR-10 [19] datasets.

978-1-5386-3408-0/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPSW.2017.44

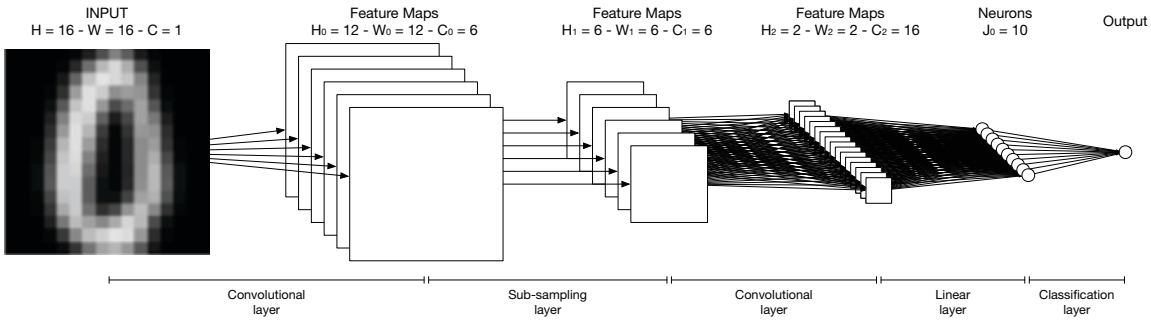

Fig. 1. Convolutional Neural Network structure.

The remainder of the paper is organized as follows. Section II provides the necessary background, while Section III discusses the related work. Then, Section IV describes the proposed acceleration methodology, and Section V presents the experimental evaluation, before ending the paper in Section VI with the concluding remarks and an outline of future work.

II. Background

This section presents the necessary background, first explaining what CNNs are and how they work, then focusing on the description of the structure of a Streaming Stencil Timestep (SST), which lays the basis of the proposed work.

A. Convolutional Neural Network

Convolutional Neural Networks were introduced in the '90s by Professor Yann LeCun [20]. This machine learning algorithm is a variant of ANN specifically tailored for image analysis and other similar 2D-structured data. Indeed, the very first application of CNNs consisted in recognizing handwritten digits. As the time went by, CNN usage resulted successful in many applications, like artificial vision, big data analysis and so on [1, 2, 3, 4]. As a consequence, CNNs now represent the state of the art in image recognition and classification.

CNN structure is a configurable chain of multiple layers, whose purpose is to first extract features from the image and then classify it. For this reason, the CNN structure may be divided into two main stages: the features extraction stage and the classification one, respectively. Figure 1 shows the overall architecture of a CNN.

The purpose of the **features extraction** stage, whose implementation relies on a particular type of ANN, is to identify and extract more and more complex features within the image. Such features are then collected into the so-called *feature maps*. In particular, the features extractor stage is defined as a chain of two different layers: the convolutional layer, and the sub-sampling (or pooling) one.

In order to extract relevant features from the image, the **convolutional layer** applies a series of K filters (or kernels) on the image. The output of each filter is a feature map. More generally, the input of a convolutional layer is a 3D volume defined by the following parameters: H, W, and C. H represents the height of the volume, W its width, C its depth. In particular, C refers to the number of the color channels (when we consider the input image) or feature maps produced by the previous layer. For each filter k, the convolutional layer applies a convolution defined as follows:

$$o_{i,j,k} = \sum_{h=0}^{H_k} \sum_{m=0}^{W_k} \sum_{c=0}^{C_k} (w_{h,m,c} \cdot x_{i+h,j+m,c}) + b_k \qquad (1)$$

where the filter is represented by a $H_k \times W_k \times C_k$ matrix of weights ($H > H_k \wedge W > W_k \wedge C \geq C_k$), b_k is a bias, i and j are the coordinates of the current pixel x in the input volume belonging to channel c. Additionally, the convolutional layer may apply a nonlinear function, e.g. $tanh()$ or $max(0, x)$, on each value in the output volume. Moreover, further hyperparameters may be configured as well, like stride S and zero-padding P.

The goal of **sub-sampling layer** is to reduce the size of the data produced by the previous convolutional layer, while maintaining the most relevant features. Indeed, usually the sub-sampling layer is inserted between two convolutional layers. This layer swipes a filter on the volume in order to cluster locally connected data. More technically, such filter is applied on each channel separately and leverages on either a *max-pooling* or *mean-pooling* function, which substitutes an input submatrix with its maximum value or its mean, respectively.

The **classification** stage is implemented by means of a classical Fully-connected Network (FCN), called Multi-Layer Perceptron (MLP). This stages receives the input from the last layer (either convolutional or sub-sampling) in the features extraction stage, and processes it in order to compute the affinity of the input image with respect to the classification classes. This stage is structured as a chain of linear layers, which may be followed by an optional final normalization operator.

The **linear layer** is composed by J simple neurons (called *perceptrons*). These neurons are in charge of aggregate the information deriving from the previous layer. The output values are therefore computed as a weighted linear combination of such neurons:

$$o_j = \sum_{i=0}^{I} (w_{i,j} \cdot x_i) + b_j \qquad (2)$$

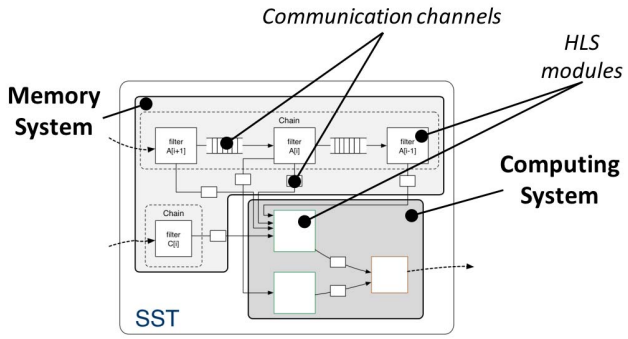

Fig. 2. A generic SST with the conceptual separation between *memory system* and *computing system*.

where $w_{i,j}$ represent the weights, x_i the neurons from the previous layer, and b_j a bias. The last linear layer will produce a number of neurons equal to the number of classification classes.

Finally, the **normalization operator** receives the output of the last linear layer and computes the affinity of the input to the classification classes as a percentage value. More technically, this operator is implemented by means of a $LogSoftMax$ operator σ:

$$\sigma_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}} \quad for\ j = 1, \ldots, K \tag{3}$$

where x_j is the output of the last linear layer. This operator enforces the K values of the output to lie in range [0, 1] and to sum up to 1. In this way, the normalization operator output can be interpreted as the probability of the input to belong to a certain class.

B. Streaming Stencil Time-step

As described in [17], an SST is an accelerator of a single ISL time-step, replicated multiple times and then arranged in a long queue of replicas to obtain the ISL-centric hardware accelerator proposed in [17]. Since convolutions are somehow amenable to single ISL time-step, in this work we base the acceleration of the CNN layers on the structure of an SST, and exploit the ideas of the work in [17] to derive the proposed dataflow accelerator of CNNs. Therefore, in order to provide the necessary background to easily read through the rest of this paper, we present an overview on the structure of an SST.

Within an SST, we can identify a *computing system* and a *memory system*, as depicted in Figure 2. The *computing system* is composed by a collection of modules that perform the actual computation taking data from the *memory system*. The *memory system* consists instead of a series of chains of *filters* interconnected via FIFO channels, one chain for every distinct input array. Each chain receives as input a single data stream, the array, and each filter represents the different array accesses needed from that array to update a single point of the ISL computation. Each filter reads any existing data element from its preceding FIFO, sending it always to the subsequent one in the chain, if any, and, when the correct data element arrives, it sends it to the computing system to allow it to produce output. The way in which this filters are interconnected ensures that the data is read only once from the off-chip memory and, at the same time, allows more concurrent accesses from the same array, also optimizing the memory resource consumption on the FPGA, that is indeed the minimum possible to achieve *full buffering*, in which data is stored on the on-chip memory until all the computations depending on it have completed.

III. Related Work

The computation pattern of CNNs is well suited for hardware acceleration, and indeed various accelerators design have been proposed to optimize the implementation of CNNs. Several works presents the acceleration of CNNs using GPUs ([2], [21], [22]), however recent works are also focusing on FPGA-based implementations. The various approaches found in literature have a common aim, which is to exploit data reuse/locality and parallelism to accelerate the CNN computation.

Farabet et al. propose a FPGA based processor for CNN implementation [14]. The proposed processor uses a dedicated memory structure and a custom Vectorial Arithmetic and Logic Unit (VALU) to take advantage of the high number of DSPs offered by the FPGA board. An efficient convolution design is also used to speed up the most critical part of the CNN.

Sankaradas et al. propose a "massively parallel, programmable coprocessor for CNNs" in [9]. The main features of the proposed coprocessor are its coupling with a high bandwidth off-chip memory and the packing of multiple data words into each memory operation. In this way, the authors implemented a stateful coprocessor able to retain large intermediate states on the off-chip memory. The proposed design obtained a performance of 3.4 Giga Multiply-Accumulate operations per second (GMACs), 31x faster than the software implementation on a 2.2GHz AMD Opteron processor.

Various work focused in particular on the memory bandwidth optimization with polyhedral model-based techniques and optimized buffer accesses to reduce the number of off-chip memory operations.

An efficient memory-centric design is shown in [7]. The proposed template for CNNs improves the performance without increasing the needed memory bandwidth, thanks to an optimized scheduling for data locality and flexible data reuse buffers. Moreover, this approach shows that it is possible to minimize the needed on-chip memory, obtaining a 13x resources reduction without decreasing the performance.

Zhang et al. propose an analytical scheme for the Design Space Exploration (DSE) to accelerate CNNs. The analysis process quantifies the computing throughput and the required memory bandwidth of the CNN design, with respect to various optimization techniques such as loop tiling. The Roofline Model [23] is then used to identify the solution that provides the best performance. The proposed case study reached 61.62 GFLOPS on a 100MHz frequency, outperforming the previous approaches.

The work in [24] presents a CNN implementation on FPGA for Image-Net large-scale classification. After the analysis of

state-of-the-art CNN models, the authors show that convolutional layers are computational centric, while Fully-Connected layers are memory centric. In order to improve the bandwidth and resource utilization, this work uses data quantization and an efficient convolver design. The results show just a 0.4% accuracy loss, and performance for the convolutional layer and the entire network of respectively 187.8 GOP/s and 137.0 GOP/s, for the implementation of VGG16-SVD on a single FPGA board.

The previously described works present the accelaration of CNNs on FPGAs, but the implementation is still manual and performed by expert hardware designers. Two works, namely [15] and [16], present instead frameworks for the automated implementation of CNNs.

The work in [16] proposes a framework to automatically generate a hardware implementation of CNNs based on the High Level Synthesis of configurable offline-trained networks. The framework is developed as a web application that takes the network parameters and weights, and converts this representation in a complete hardware implementation of the network.

The work presented in [15] presents Caffeine, an hardware/software codesign library to accelerate an entire CNN on FPGA. The proposed work focuses on the network computing and bandwidth optimization by the memory access reorganization. Moreover, the implemented library provides various hardware/software definable parameters and is integrated into the Caffe [25] deep learning framework.

To the best of our knowledge, there are no FPGA-based acceleration methodologies that fully exploit the dataflow computation pattern, provide a scalable implementation of each layer (from single-input-port/single-output-port to fully parallel that can be adapted to the available resources), and are able to implement a high-level pipeline within the CNN layers.

IV. Proposed methodology

In this section, we present the proposed architecture for the dataflow acceleration of CNNs, by describing how each layer is implemented and detailing how a complete network is constructed. The design is composed of several independent *modules*, in order to allow the implementation of different networks without redesigning the whole system. This feature will be exploited to automate the implementation of CNNs on FPGAs with this methodology in future work. Moreover, a modular architecture allows precise optimizations related to each particular layer of the network. Figure 4 and Figure 5 show two CNN designs obtained by applying the proposed approach.

A. Convolutional layer architecture

The convolutional and sub-sampling layer implementations are here described together, as they share similar memory access patterns and therefore will mostly use the same memory and design optimizations.

In order to enable the scalability of the layer and provide the ability to tune the trade-off between parallelism and resource consumption, our proposed design for the implementation of a convolutional layer is based on a *modular* and *parametric* approach. The convolutional layer accepts IN_PORTS input channels at a time, and exposes OUT_PORTS output channels ports. These interfaces can transmit multiple Feature Maps (FMs) by interleaving the different values over the same port. The number of IN_PORTS and OUT_PORTS is then used as a parameter to instantiate the *memory structure* and the *computation core*, which performs the convolutions and combine them. Figure 3 shows a schematisation of the convolutional layer.

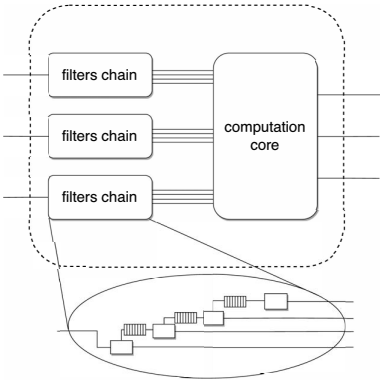

Fig. 3. Schematisation of the design of a convolutional layer. Notice how the structure resembles the one of an SST, shown in Figure 2.

The *memory structure* of the layer is designed in order to exploit the *dataflow* computational pattern of convolutions. The architecture used to transmit and use the input values over each input channel of the layer is based on the *memory system* of an SST, as described before in Section II-B. The values are moved onto IN_PORTS pipelines composed by a series of *filters* and *FIFO* queues, one separate pipeline per input channel. Within each pipeline, the different filters receive data from one input port and redirect their values to the next FIFO or also to a register slice. Scalability with respect to the data size can be achieved via a memory/bandwidth trade-off, as described in [18]. The filters and FIFOs are connected in a way which resembles a *sliding window* architecture, such that the pipeline is gradually filled with the values, while the register slices are used to store and transmit the window values.

These registers will be then read by the *computation core*. In this way, after the memory pipeline is filled, the *computation core* receives an entire window for each clock cycle. There are three cases that lead to three different configurations of the *memory structure* of a layer i, being $i-1$ the previous layer:

- $OUT\_PORTS_{i-1} = IN\_PORTS_i$,
- $OUT\_PORTS_{i-1} < IN\_PORTS_i$,
- $OUT\_PORTS_{i-1} > IN\_PORTS_i$.

The case in which $OUT\_PORTS_{i-1} = IN\_PORTS_i$ is the trivial one as the output channels of the layer $i-1$ and the input channels of the layer i can be connected directly with no adjustments. If instead $OUT\_PORTS_{i-1} < IN\_PORTS_i$, the values can be correctly routed from the output port of $i-1$ to the correct input port of i thanks to a *demux* core that redirects data to the proper input port of i according to how the

different FMs are interleaved on the output port of $i-1$. On the other hand, if $OUT\_PORTS_{i-1} > IN\_PORTS_i$, the filters of each input channel of i need to be modified adding an additional innermost loop to cycle the reads from the different output channels of $i-1$ and enlarging the FIFO size to fit the data of all this channels.

The convolutional layer may also present a *stride* parameter, meaning that the convolution window does not slide one pixel at a time, but skips of some x and y coordinates each time. This can be implemented also on the filters, by changing the condition on which the values are redirected to the window registers. The filters and demux core of the *memory structure* have been implemented by means of Vivado HLS [26]. The *memory structure* just described fills the registers that contain the current window, on which the *computation core* performs the various convolutions needed. This core is created starting from a set of parameters, related to the layer structure and the desired characteristics of the execution. In particular, the core accepts IN_PORTS used to receive IN_FM channels, and exposes OUT_PORTS output ports used to send OUT_FM channels to the next layer. Furthermore, the *computation core* expects a $KW \times KH$ window for each input port, where KW and KH are respectively the width and height of the convolution window. If the number of ports is different than the number of feature maps, the input windows will be related to a different channel each time, as described earlier.

Algorithm 1 Convolution Layer Pseudocode

```
foreach (x, y) ∈ Coordinates do
    outputs ← biases
    for i = 0 to IN_FM step IN_PORTS do
        buf ← IN_PORTS windows
        buf ← buf · weights
        outputs ← outputs + reduce(buf)
    end for
    send outputs on OUT_PORTS ports
end for
```

As for the filters and demux of the *memory structure*, the *computation core* has been implemented using Vivado HLS, and its pseudocode is shown in Algorithm 1. The kernel has $IN\_PORTS \times (KH \times KW)$ input ports and OUT_PORTS output ports, all implemented using the *Axi4Stream* protocol. As shown in Algorithm 1, the input is taken just for IN_PORTS FMs at a time, and is copied on a completely partitioned buffer. The convolution is done by multiplying the input buffer values with the respective weights, whose values are currently defined at design time and therefore hardcoded in on-chip memory, taking into account the current input and output FMs to compute. The multiplications results are then fed into a *tree adder* (indicated by the *reduce* function) and added as a partial result to the relative output value register. The tree adder is used in order to improve the initial latency of the core, as it executes the additions on parallel levels which decrease the pipeline depth. After all the needed convolutions has been performed, the core sends the results on the output ports. The *PIPELINE* directive is applied to all the internal loops, including also the input/output operations.

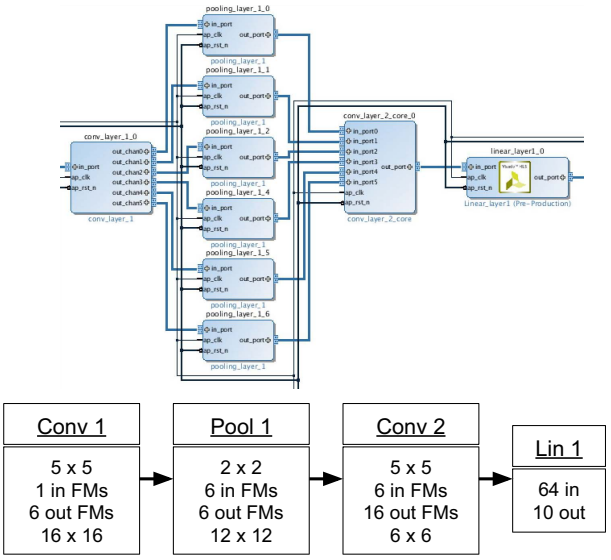

Fig. 4. Block design of the CNN for the USPS dataset. Each block in the picture shows the window size, the number of input and output channels and the number of windows taken as input. For this CNN, the low resource consumption allowed us to completely parallelize both the first convolutional and the first sub-sampling layers, with the approach described in Section IV-A.

An additional parameter is added to the directive, that is the pipeline initiation interval. The interval is computed as

$$Pipeline\ II = max(\frac{OUT\_FM}{OUT\_PORTS}, \frac{IN\_FM}{IN\_PORTS}) \quad (4)$$

This additional parameter is then used by the HLS tool to infer the level of parallelism to apply to the different instructions which belong to the loop nest. This affects both the total latency and throughput of the *computation core*, as well as the area utilization (mainly in terms of DSPs and LUTs).

B. Fully-connected layer architecture

In order to describe the FCN implementation in the proposed methodology, we have to take into account some initial considerations. First, the stucture of a FCN is similar to the structure of a convolutional layer, due to the similar output dependence on all the input values. In fact, a FCN performs Multiply-Accumulate operation (MAC) operations on all the input with the given weights, and this kind of execution can be described as a 1×1 convolution. Each input value is therefore seen as a different input channel, and each output value as a different output channel, all in a 1×1 FM and on a 1×1 window. Given this assumptions, it seems clear that the FCN layer implementation can be done by using the same code and considerations made in Section IV-A. However, given the unitary window size and the high number of operations involved, the layer implementation can be optimized with respect to the one of convolutional layers in order to become more simple and with a lower resource utilization (with respect to the number of input and output FMs). In particular, the unitary window size of this layer makes the filters sequences of the *memory structure* not needed. Moreover, due to the

high number of input and output channel, if the layer was completely parallelized, it would increase too much the DSP utilization. In order to face this two issues, we decided to implement a FCN layer as a single-input-port/single-output-port convolutional layer. In this way, the number of parallel multiplications is reduced, while the execution time remains linearly related to the number of input and output values. The core accepts a single *Axi4Stream* interface to receive the input values of the network, and offers one output interface to send the output values (which represent the classes or hidden neurons output values). For each input value, all the 1×1 convolutions related to all the output FMs are performed in the same clock cycle, while the output values are then sent to the output port sequentially after all the inputs have been processed.

An issue related to the used data types, in particular with floating point numbers, has been the high latency of each accumulation (e.g. 11 clock cycles for floats). In fact, this latency leads to an infeasible pipelining, as there is no way to have a unitary initiation interval. In order to overcome this issue, we added more accumulators and interleaved their use by exploiting a partial unrolling of the main loop. By using a higher number of accumulators than the single addition latency, we reached a lower total latency of the layer, but with a higher resource utilization. This is an additional reason to justify the implementation of a single-input-port/single-output-port version of the layer. The issue does not arise when using integer values, and will be subject to further study.

C. Network Design

The design of an entire network starts from the choice of the parameters to set for each module. This choice should take into consideration the resource utilization of each core and its total latency, as it will affect the total execution time of the network like the stage of a pipeline (i.e. the pipeline interval is its slowest stage time). Indeed, the resulting network will exactly act like a high-level pipeline. At steady state, all the different layers of the network will be concurrently active and computing. This effect becomes especially beneficial when batches of multiple images feed the network, as will be shown in the experimental evaluation of Section V.

Many options are offered when instantiating the different layers, mainly in terms of I/O ports and parallelization. In this work, we did not perform any DSE and instead we just determined empirically the levels of parallelization and therefore the number of I/O ports, essentially to find a layout that fits on the FPGA chip. Future work will address the automation of the DSE, also taking into account the possibility of splitting the layers into many parallel versions, and map such enlarged network design onto a multi-FPGA system. This approach should allow large performance improvements, as the layers can be totally parallelized given that there are enough available resources, effectively maxing out the achievable performance For the sub-sampling layer connection, as there is no combination between FM and rather just a sub-sampling of each FM, it is possible to insert parallel sub-sampling layer cores, one for each previous layer output port (with the proper

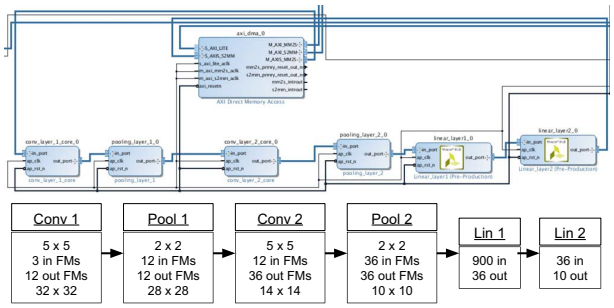

Fig. 5. Block design of the CNN for the CIFAR-10 dataset. As for Figure 4, each block in the picture show the window size, the number of input and output channels and the number of windows taken as input. To fit the entire CNN design, this time we could not perform any parallelization optimization within the layers.

memory structure as detailed in Section IV-A). In this way, the sub-sampling cores act as a standard filter inserted between the convolutional layers without occupying too much area (perfect pipelining and no multiple windows/convolutions). The FCN is implemented in the network design by splitting it in its distinct layers. For each layer, a single FCN layer core is implemented, as described in Section IV-B. The distinct layer cores are then connected sequentially through their I/O ports.

V. Experimental Evaluation

A. Eperimental setup

To validate the proposed approach, we implemented the different cores using Vivado HLS and Vivado IPI 2016.3. The tests have been performed by implementing the designs on a Xilinx VC707 board [27], mounting a Virtex-7 (xc7vx485t) FPGA, and running them at a 100MHz frequency.

For each implemented network, a base design has been used as a support for the testing phase. The design includes a soft-core processor (Microblaze) coupled with an Axi-Timer, an Axi4-Interconnect and a DMA to handle the communication from/to the CNN cores.

B. Test Cases

To evaluate the proposed approach, two different networks have been designed as test cases.

1) Test Case 1: The first network is composed of 4 different layers, and is trained and tested with images from the USPS dataset, composed of handwritten digits (16x16 grayscale images) from the U.S. Postal Service. In particular, the network includes a 5x5 convolution layer (1 input - 6 output channels) a max-pooling layer (2x2 window - 2 pixels stride), a second convolution layer (5x5 - 6 input - 16 output FMs) and a single FCN layer with 10 output classes. Given the amount of available resources and the CNN resource occupation, we have been able to parallelize completely the first convolutional and sub-sampling layers, while the second convolutional layer has been left with a single output port. The network structure and block design are shown in Figure 4.

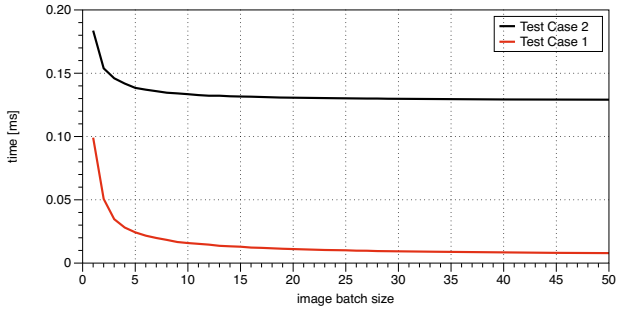

Fig. 6. Mean time to process an image related to the size of the batch of images. Notice how, in both cases, the high-level pipeline realized within the network improves the mean time per image when a batch of images is processed by the CNN. In both tests, the time converges approximatively when the size of the batch of images becomes greater than the total number of layers of the CNN. For the sake of readability, we show the results up to a batch of 50 images, as at that point convergence is already reached.

TABLE I
FPGA RESOURCES USAGE

	Resources			
	Flip-Flops	**LUT**	**BRAM**	**DSP Slices**
	Utilization	**Utilization**	**Utilization**	**Utilization**
Test Case 1	41.10%	50.86%	3.50%	55.04%
Test Case 2	61.77%	71.24%	22.82%	74.32%

2) Test Case 2: The second implemented network is quite larger, and it was trained and tested against images from the CIFAR-10[19] database (32x32 RGB images). The network is composed of 6 layers, including a 5x5 convolution layer (3 channels in input and 12 in output), a second convolutional layer (5x5, 12 input and 36 output FMs), two sub-sampling layers (2x2 window with 2 pixels stride) and two linear layers, with 10 classes as the final output. In this case, the convolutional layers require too much area to allow parallelization, so they have been implemented as the single-input-port/single-output-port version. The network design and parameters can be seen in Figure 5.

Both the networks are implemented with single floating point precision.

C. Results

Table I shows the resources usage of both test cases. Notice how, as previously stated, the CNN of test case 1 is small and consumes approximatively less than 50% of the available resources on the FPGA even with the first convolutional and sub-sampling layers fully parallelized. The CNN of test case 2 is instead bigger and consumes a higher number of resources, making impossible to improve performance by parallelizing the layers.

To validate the high-level pipeline realized by the proposed approach, we tested both CNNs against an increasingly high batch of images, from 1 up to 1000. Notice that the datapath from the DMA towards the CNN is 32 bits wide and the available bandwidth, for all the performed tests, is 400MB/s. Figure 6 shows how the mean time per image diminishes with an increasing number of images per batch, until it converges to approximatively 5.8 μs for test case 1 and 128.1 μs for test case 2. In both cases, convergence is reached approximatively when the size of the batch of images becomes greater than the total number of layers of the CNN.

TABLE II
PERFORMANCE AND POWER EFFICIENCY RESULTS

	Dataset	**GFLOPS**	**Power Efficiency GFLOPS/W**	**Image Latency (ms)**	**Images/s**
Test Case 1	USPS	5.2	0.25	0.0058	172414
Test Case 2	CIFAR-10	28.4	1.19	0.128	7809
[28]	CIFAR-10	-	-	-	2318

Finally, Table II reports performance and power efficiency. Performance measurements are done taking into account also data transfers, as they are interleaved with computation. We perform a comparison between our test case 2, built for the CIFAR-10 dataset, and the work by Microsoft Research in [28], the only work, that to the best of our knowledge, is accelerating a CNN for the same dataset on FPGA, an Altera Stratix V D5. We are able to yield 3.36x the performance of [28] with the proposed approach.

With the presented experimental evaluation, we aimed at validating the proposed methodology and verify the effectiveness of the high-level pipeline. We tested the approach on relatively small networks and with a sub-optimal usage of the available off-chip memory bandwidth. Future work will address this issue and implement larger CNNs to perform a proper comparison with the work present in literature.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a modular and scalable approach to accelerate CNN designs on FPGAs with a dataflow implementation. This approach is based on previous work on the acceleration of ISLs [17, 18], as ISLs indeed share some similar characteristics with convolutions. The resulting accelerator realizes a high-level pipeline between the different network layers, which guarantees an increase in performance when the CNN is employed to process batches of multiple images. It also realizes an efficient utilization of the on-chip memory, along with the possibility of scaling each layer from single-input-port/single-output-port to completely parallel, depending on the amount of available resources.

We evaluated the proposed approach on two test cases, a network for the USPS dataset, and a bigger network for the CIFAR-10 dataset, showing that the average time to process an image can be improved when the CNN is set to process batch of images, and tends to converge to a fixed value when the size of the batch is bigger than the number of layers of the network.

Future work will focus on different aspects. First, we will optimize the design itself, to better exploit the available off-chip memory bandwidth. Second, we will investigate scalability by implementing bigger networks on a multi-FPGA system, with an automated DSE mechanism. We will then also test the proposed approach on bigger and more popular CNN models like AlexNet [2] of VGG [29] to properly compare

this approach with the ones available in the state of the art. As last piece of future work, we envision the development of an automated design flow and its integration into industry-standard frameworks.

ACKNOWLEDGMENT

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

REFERENCES

[1] 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.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[3] 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.

[4] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio, "An empirical evaluation of deep architectures on problems with many factors of variation," in *Proceedings of the 24th international conference on Machine learning*. ACM, 2007, pp. 473–480.

[5] K. Yu, "Large-scale deep learning at baidu," in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. ACM, 2013, pp. 2211–2212.

[6] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, S. Dasgupta and D. Mcallester, Eds., vol. 28, no. 3. JMLR Workshop and Conference Proceedings, May 2013, pp. 1337–1345. [Online]. Available: http://jmlr.org/proceedings/papers/v28/coates13.pdf

[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] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[9] 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.

[10] 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.

[11] A. Dundar, J. Jin, V. Gokhale, B. Krishnamurthy, A. Canziani, B. Martini, and E. Culurciello, "Accelerating deep neural networks on mobile processor with embedded programmable logic," in *Neural information processing systems conference (NIPS)*, 2013.

[12] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 247–257.

[13] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf, "A programmable parallel accelerator for learning and classification," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 273–284.

[14] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE, 2009, pp. 32–37.

[15] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *Proceedings of the 35th International Conference on Computer-Aided Design*, ser. ICCAD '16. New York, NY, USA: ACM, 2016, pp. 12:1–12:8. [Online]. Available: http://doi.acm.org/10.1145/2966986.2967011

[16] E. Del Sozzo, A. Solazzo, A. Miele, and M. D. Santambrogio, "On the automation of high level synthesis of convolutional neural networks," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 217–224.

[17] G. Natale, G. Stramondo, P. Bressana, R. Cattaneo, D. Sciuto, and M. D. Santambrogio, "A polyhedral model-based framework for dataflow implementation on fpga devices of iterative stencil loops," in *Proceedings of the 35th International Conference on Computer-Aided Design*. ACM, 2016, p. 77.

[18] R. Cattaneo, G. Natale, C. Sicignano, D. Sciuto, and M. D. Santambrogio, "On how to accelerate iterative stencil loops: a scalable streaming-based approach," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, p. 53, 2016.

[19] "CIFAR-10," accessed: 15th of January 2017. [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 26–35.

[25] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," 2014. [Online]. Available: http://caffe.berkeleyvision.org

[26] Xilinx Inc., "Vivado HLS," accessed: 15th of January 2017. [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[27] "VC707 Evaluation Kit," accessed: 15th of January 2017. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html

[28] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, no. 11, 2015.

[29] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.