2018 IEEE International Parallel and Distributed Processing Symposium Workshops

A Framework with Cloud Integration for CNN Acceleration on FPGA Devices

Niccolò Raspa, Giuseppe Natale, Marco Bacis, Marco D. Santambrogio
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Via Ponzio 34/5, Milano, Italy
{niccolo.raspa, marco.bacis}@mail.polimi.it
{giuseppe.natale, marco.santambrogio}@polimi.it

Abstract—**The recent years have seen a rapid diffusion of deep learning algorithms as Convolutional Neural Networks (CNNs), and as a consequence, an intensification of industrial and academic research focused on optimizing their implementation. Different computing architectures have been explored, and among all of them, FPGAs seem to be a very attractive choice, since they can deliver sustained performance with high power efficiency, as CNNs can be directly mapped onto hardware, and still offer flexibility thanks to their programmability.**

In this paper, we present an end-to-end framework to implement CNNs using a dataflow acceleration methodology. The resulting spatial accelerator can be scaled in size if enough resources are available and can exploit both intra- and inter-layers parallelism. We integrate the proposed framework with the deep learning engine Caffe, meaning that we are able to generate the accelerator starting from a Caffe model. We also provide cloud integration of such framework, enabling users to synthesize and deploy the accelerator on the Amazon F1 instances.

Index Terms—**Convolutional Neural Networks, Dataflow Architectures, Cloud-based acceleration, FPGA**

It is undeniable that we are currently assisting at the artificial intelligence revolution of Artificial Neural Network (ANN) algorithms, commonly referred to as *deep learning*. Indeed, thanks to their impressive results, they are rapidly growing in popularity and seeing massive adoption in many fields, demonstrated by the interest of industry leaders as Facebook [1], Google [2] and Baidu [3], that are integrating deep learning in their core businesses. Among the different ANN algorithms, particularly relevant are Convolutional Neural Networks (CNNs), that today are the de facto state of the art for visual recognition and classification [4], [5], [6], [7], and are successfully applied also in many other fields [8], [9]. CNNs essentially mimic the behavior of biological neurons in the *Primary Visual Cortex*, by applying consecutive convolution filters on the input data to extract features at an increasing level of abstraction, used then for classification, emulating the *receptive field* of such neurons.

With the improvements in accuracy and the refinements of the CNN models, size and complexity of the resulting networks are also increasing. As CPUs are basically no longer a viable alternative, many works have focused on exploiting GPUs to implement CNNs [4], [10], [11]. However, GPUs power consumption can be prohibitive, especially at the datacenter level [12]. For this reason, recently academia and industry have started exploring less conventional options as FPGAs [13], [14], [15], [16], [17], [18] or ASICs [19], [20] to cope with the increasing computational and energy efficiency requirements. In particular, FPGAs offer an interesting compromise between hardware and software thanks to their reconfigurability, being more flexible than ASICs but still highly energy efficient. Indeed, they have shown the potential to become the platform of choice for deep learning acceleration [21], and even to perform better than recent ASIC solutions [22].

The CNN computational pattern is mainly dataflow-based and presents very few control structures, and while implementing CNNs requires high computational power, as previously stated, it also offers great potential for massive parallelization and data reuse. The reconfigurability of FPGAs enables the possibility of exploiting these opportunities, especially if the synthesized architecture is a dataflow, spatially distributed architecture, that better suits the characteristics of both FPGAs as well as CNNs. Nevertheless, albeit FPGAs are a very promising candidate for CNN inference acceleration, the process of designing and deploying hardware accelerators is still a hard and complex task that requires expertise in FPGA programming and knowledge of hardware design tools. As far as we know, only one work [16] in literature offers integration with an industry standard deep learning library, Caffe [23], to alleviate this issue. However, this work still requires users to have physical access to FPGA devices, that given their prohibitive cost cannot always be assumed to be the case. Recently, Amazon expanded its cloud offerings including also FPGAs [24]. This presents a great opportunity since FPGA-based CNN accelerators can now be deployed in the cloud, with no need for physical access (nor ownership) of the devices. However, to the best of our knowledge, there is no automated framework in the literature that allows the user to deploy an FPGA-based CNN accelerator in the cloud.

Given the presented context, we can now summarize the contribution of this paper as follows:

- We implement an end-to-end framework that is fully integrated with the industry standard Caffe, and therefore allow to use Caffe models as input, completely avoiding the hassle of FPGA programming. This framework is also integrated with the Amazon AWS F1 instances and is, therefore, therefore, able to deploy the resulting CNN in the cloud, dramatically increasing the use case scenarios for FPGAs in this space.

- We propose an FPGA-based spatial accelerator for CNNs inference, consisting in a distributed dataflow architecture of simple and independent elements communicating over FIFOs. This accelerator can exploit different level of parallelism, and can theoretically scale in size, according to the available FPGA resources. The proposed methodology builds upon the previous work presented in [25].

1. Related Work

Several recent works in literature aim at providing a solution for FPGA-based CNN acceleration.

In [13] the authors present a CNN accelerator that is synthesized via High Level Synthesis (HLS). They focus on optimizing the communication exploiting on-chip reuse buffering, and the computation performing loop transformations as tiling, unrolling and pipelining. They also propose a performance evaluation mechanism based on the Roofline model, that allows for the selection of the best implementation in the solution space.

The work in [14] provides a Verilog-based CNN accelerator that can exploit inter-output parallelism, *i.e.* where the computation of the different feature maps within a convolutional layer is parallelized and tiling is applied to reduce the overall memory footprint as done in [13]. Data quantization is performed to reduce bandwidth requirements and resource utilization, with negligible impact on the resulting accuracy.

The authors of [15] propose an Open-CL accelerator, which implements the convolutional layers as matrix multiplications, flattening and rearranging the input feature maps. Also, they provide a methodology to find the accelerator's hyperparameters that minimize the total CNN execution time under bandwidth and resource constraints.

The approach proposed in [18] employs the three classical loop optimizations, namely unrolling, tiling, and interchange, to optimize the resulting CNN implementation. The authors provide an in-depth analysis of the impact of these transformations and elaborate an analytical model that can be used to find what loop transformations to apply and to which extent. A uniform systolic architecture of Processing Elements (PEs) is then used to implement the entire network.

In [17] it is presented an automation flow that starts from a C/C++ description of the computation. The resulting accelerator is a fine-grained 2-D systolic array designed to improve timing and exploit data reuse. The authors also provide an analytical model for resource utilization, as well as a model for performance, and propose a 2-phase design space exploration mechanism where the first phase optimize the CNN design, while the second phase perform platform-specific optimizations.

Caffeine, presented in [16], is a framework that propose a unified matrix multiplication representation of both the convolutional and fully-connected layers, implementing the resulting accelerator as an HLS-generated systolic array. The framework is fully integrated with Caffe and therefore allow for the synthesis on FPGA of Caffe models.

The systolic designs presented in [16], [17], [18] can be compared to our dataflow architecture, in the sense that they spatially distribute the computation onto smaller PEs as we do in this work. However, none of these approaches explicitly exploit the dataflow computation pattern of CNNs providing a tailored spatial architecture. Moreover, among all the presented works, only [15], [16], [17] provide an automated framework to implement CNNs from a high-level specification. However, only [16] is integrated with an industry standard deep learning library. No works in literature instead implement and evaluate cloud-based acceleration. To the best of our knowledge, this work is the first step in that direction.

2. CNN Overview

Convolutional Neural Networks (CNNs) extends from ANNs and are first inspired by neuroscience [26]. After over twenty years of evolution, started by the work of professor Yann LeCun in the late 1990s [27], CNNs begun to attract the attention of industry and academia when in 2012 a team of researchers from the University of Toronto outperformed the competition by a substantial margin in the ImageNet challenge [4]. From there, CNNs have rapidly become the state of the art in visual recognition and classification [5], [6], [7], finding applications also in other fields as text classification [8] and recommendation systems [9].

The structure of a CNN can be summarized as a configurable chain of multiple and different layers, whose purpose is first to extract the relevant features from the structured input data, and then perform classification of the input based on these features. We can, therefore, identify two separate phases within a CNN, each characterized by different kinds of layers. The first stage is the *features extraction*, and is composed of alternating *convolutional* and *sub-sampling* layers. In this phase, the chain of layers extracts more and more abstract features from the input data, relying on the processing made by the previous layers. At first, the features extracted are simple lines, circles, squares, and become increasingly more complex as data flow deeper in the network (an example being shown in Figure 2). These features are encoded in *feature maps*. The second stage is the *classification* and is implemented by a classical Multi-Layer Perceptron (MLP), a fully-connected network of simple neurons. The MLP takes as input the feature maps extracted by the previous stage, and classify the input accordingly. We will now describe more in detail the distinct layers that we can find in CNNs. Figure 1 depicts a generic CNN architecture.

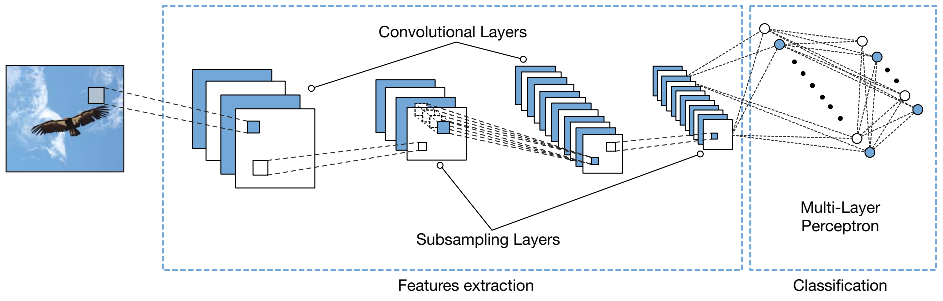

Figure 1. Convolutional Neural Network structure. We can separate the network into two parts: the *features extraction*, composed of alternating convolutional and sub-sampling layers, and the *classification*, implemented as a MLP.

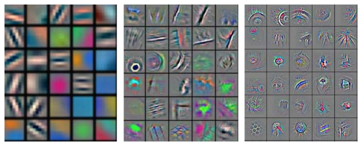

Figure 2. Example of increasingly more abstract features, from left to right, identified during the features extraction stage.

2.1. Convolutional Layer

Convolutional layers apply a series of F filters (referred to also as *kernels*) on the input grid, with the purpose of detecting features in the presented data. From a high-level perspective, each filter $f \in F$ is convolved across the input data, producing the corresponding output. Input and output features are encoded as *feature maps*. Each filter f is automatically determined during the learning phase through weight adjustments, trained to activate when a specific feature at some spatial position in the input is detected. The corresponding equation to compute each element (i, j) of the output feature map ϕ can be written as:

$$o_{i,j,\phi} = \sum_{m=0}^{M_f} \sum_{n=0}^{N_f} (w_{m,n,\phi} \cdot x_{i+m,j+n}) + b_\phi \tag{1}$$

where M_f and N_f are the width and the height of the filter f, x represent the input data, w the weights, and b_ϕ is an optional bias. The dimension of the resulting feature map will be reduced according to the following equation:

$$\begin{aligned} \omega_{\text{new}} &= \omega_{\text{old}} - \omega_f + 1 \\ \gamma_{\text{new}} &= \gamma_{\text{old}} - \gamma_f + 1 \end{aligned} \tag{2}$$

with ω representing the height, and γ the width, and with ω_f and γ_f being respectively the height and width of the given filter f. A convolutional layer allow also for the selection of other hyperparameters, like stride and zero-padding. Moreover, to emphasize relevant features, the non-linearity of the output may also be increased applying the Rectified Linear Unit (ReLU) function $f(x) = max(0,x)$, or other activation functions like sigmoid $f(x) = \frac{1}{1+e^{-x}}$ or hyperbolic tangent $f(x) = tanh(x)$. In general, ReLU is preferable to the other functions as it can train the CNN faster [4] without significant accuracy penalty.

2.2. Sub-sampling Layer

A sub-sampling layer (know also as *pooling* layer) is usually inserted in between two convolutional layers and its function is to lower the amount of data to be stored in memory (and inherently the amount of computation), reducing the spatial size of the feature maps, providing also a form translational invariance of the features, as well as helping controlling overfitting. Sub-sampling layers are implemented similarly to convolutional layers, with filters swiped over the input data. The difference is in the operations applied, that usually consists in substituting the input sub-matrix with its average or its maximum (*max-pooling*). The most common size for a sub-sampling filter is 2×2, that is also the smallest. As done for the convolutional layers, we can express the reduction in size of the output feature maps for a sub-sampling layer as:

$$\begin{aligned} \omega_{\text{new}} &= \left\lceil \frac{\omega_{\text{old}} - \omega_f}{\rho} \right\rceil + 1 \\ \gamma_{\text{new}} &= \left\lceil \frac{\gamma_{\text{old}} - \gamma_f}{\rho} \right\rceil + 1 \end{aligned} \tag{3}$$

where ρ represent the amplitude of the sliding window of the sub-sampling filters.

2.3. Fully-connected Layer

After the feature extraction stage has taken place, the extracted features are passed to a MLP for the classification, composed of multiple fully-connected layers, where each neuron of a layer is connected with all the neurons of the previous and subsequent layers. The output of a single neuron l consists of a weighted linear combination of the neurons of the previous layer:

$$o_l = \sum_{h=0}^{H} (w_{h,l} \cdot x_h) + b_l \tag{4}$$

with H the neurons of the previous layer, w representing the weight of the link between the neuron with index $h \in H$ and l, x_h the neuron of the previous layer, and b_l an optional bias. The last fully-connected layer contains as many neurons as the classes to be recognized. A *normalization* layer, usually

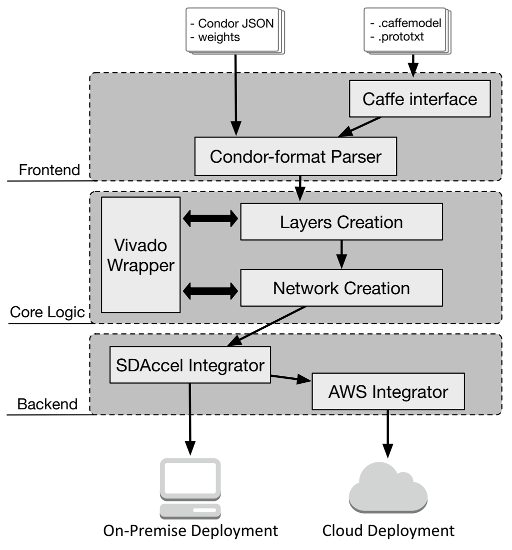

Figure 3. Condor framework multi-tier architecture.

implemented by means of a *LogSoftMax* operator σ, is sometimes appended to the final layer:

$$\sigma(\boldsymbol{o})_y = \frac{e^{\boldsymbol{o}_y}}{\sum_{y=1}^{Y} e^{\boldsymbol{o}_y}} \quad for\ y = 1,\ldots,Y \qquad (5)$$

This operation enforces the X values of the output vector $\boldsymbol{o}$ to lie in the range $[0,1]$ and to sum up to 1, such that they can be interpreted as the the probability of the input to belong to a certain class.

3. Proposed Automation Framework

We now present the architecture of the proposed framework, including the different possibilities concerning integration and deployment, in Section 3.1, an overview of the hardware accelerator design in Section 3.2, and the implemented design automation flow in Section 3.3.

3.1. Framework Architecture

Figure 3 illustrates the architecture of the framework, that is a multi-tier architecture consisting of three layers: the *frontend*, the *core logic*, and the *backend*. The framework is mainly developed using Python, although we also rely on C for a restricted number of functionalities, mainly to interact with the Xilinx toolchain. We called this framework *Condor* (CONvolutional neural networks Dataflow Optimization using Reconfigurable hardware). We now provide an overview of the functionalities of each tier.

3.1.1. Frontend. The main goal of the top tier, the frontend, is to collect all the necessary input to allow the design of the accelerator. There are currently two supported methods: the user can either specify all the input files manually, according to the Condor internal specification or use a pre-trained Caffe model, providing the *caffemodel* and *prototxt* files. In the future, more input methods will be added. We plan to support other popular deep learning libraries, such as TensorFlow or Caffe2, and we are considering adding support to the ONNX format[1].
The input consists of:

- ***Network Representation*:** the core-logic tier uses an internal JSON to describe the topology of the network. It resembles the caffe prototxt file but contains more information about the underlying hardware of the accelerator, such as the desired board, the operating frequency and desired level of parallelism of each layer. The user can either specify this file manually or provide the prototxt file of a pre-trained caffe model.
- ***Weights***: users must also provide all the weights and biases of the convolutional and fully-connected layers. This is needed as we do not perform the training step, we focus on the acceleration and optimization of the sole CNN inference. Weights and biases are kept as external files and are loaded dynamically at runtime. This enables the update of the network (for instance if better accuracy is achieved) without the need for re-synthesizing the accelerator. As per the network representation, weights and biases can be expressed manually or automatically extracted from the *caffemodel* file.
- ***Deployment Option***: users must select beforehand where to deploy the resulting accelerator, either specifying one of the supported boards or choose to deploy the accelerator on an Amazon F1 instance. Different actions will be taken by the next tiers depending on which option is selected.

3.1.2. Core Logic. This tier contains the main logic of the framework. It uses the input provided by the frontend to create a hardware accelerator that exploits the dataflow computational pattern and is tailored to the selected deployment option. The intermediate result of this tier is a fully packaged Vivado IP. The layer consists of the following logical modules:

- ***Vivado Wrapper***: this module interacts with Vivado and Vivado HLS in order to provide a high-level python API to the above modules.
- ***Layer Creation***: this module is responsible for creating each layer that characterizes the input CNN. The Condor-specific JSON file (the output of the frontend), containing the network representation, is analyzed and used to map the different layers into hardware according to the methodology presented in Section 3.2. Each layer is packaged as a Vivado IP.
- ***Network Creation***: this module connects all the created layers to derive the hardware accelerator for the CNN under analysis. Using Vivado IP Integrator, the IPs packaged by the previous module are connected to form the final accelerator IP.

3.1.3. Backend. To allow developers to use the final accelerator to optimize their applications and take advantage of FPGA platforms without any prior knowledge required,

1. https://onnx.ai

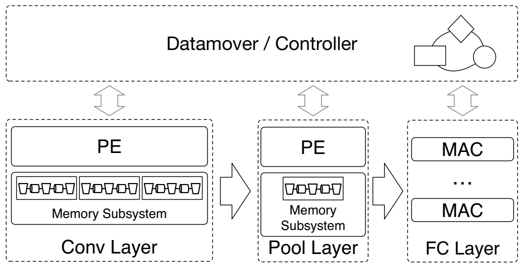

Figure 4. A scheme representing the proposed hardware accelerator structure.

we have decided to integrate the Condor framework with SDAccel. SDAccel is a software development environment targeting FPGA platforms that enables a CPU/GPU-like development experience. With this integration, it is possible to deploy the resulting accelerator on-premise (on a locally accessible board) or in the cloud using the F1 instances. In the first case, the framework uses the Xilinx OpenCL Compiler (XOCC) to produce the Xilinx OpenCL Compute Unit Binary (xclbin) file needed to configure the target board directly. In the second case, it is not possible to load a bitstream directly onto the FPGAs of an F1 instance; it is instead necessary to create an Amazon FPGA Image (AFI) first. The AFI creation process can be done on-premise with Xilinx tools, but it requires special licenses and additional setup[2] which may not be accessible to machine learning practitioners. Therefore, for usability and accessibility reasons we have decided to require users to run the Condor framework inside an FPGA Developer Amazon Machine Image, which provides the aforementioned licenses at no additional cost. In this way, we can hide all the complexity and cost of FPGA development to the end-user and provide an end-to-end process that starts from a high-level description of a CNN and produces an AFI containing the hardware accelerator. We do not entirely exclude however the possibility to create the AFI on-premise, we just do not investigate this option. The end-user might still be able to follow this approach with some tweaking of the Condor framework.

3.2. Hardware Design

The focus of the work proposed in this paper is mainly on the automation and integration with industrial tools of what we have previously done by hand in [25]. In that work, we propose an architectural template that exploits the dataflow computational pattern distributing data movement and actual computation to a set of independent modules that communicate through FIFO channels using blocking reads and writes. As the work evolved though, we had to implement a set of structural changes to extend the applicability of our acceleration methodology. Albeit the automation framework is fully functional and these changes are mostly already implemented, the extension process is still in development, and we plan to discuss and evaluate the refined architecture in greater detail in a future publication. We here provide an overview of the current version.

The accelerator is a composition of a set of building blocks with different functionalities: *PEs*, that implement the actual computation performed by the various CNN layers, *filters*, that feed the PEs and implement on-chip buffering for the features extraction layers using non-uniform memory partitioning [28], and *FIFOs*, that are used to implement the communication channels between PEs and filters, as well as to perform the non-uniform memory partitioning together with the filters, as we will discuss shortly. We interface our accelerator with the on-board memory using a custom *datamover* that exchanges data with the accelerator using streaming connections. We rely on the on-board memory to transfer input, output, weights and store partial results when they do not fit on the on-chip storage. Figure 4 provides a visualization of the proposed hardware accelerator template.

A PE can be used to implement multiple subsequent layers of the CNN and is concurrently active together with all the PEs synthesized. More specifically, the PEs are arranged as a high-level pipeline where the output of a PE is the input to the next one. In practice though, each PE also communicates with our custom datamover to receive the weights and exchange partial results in case needed. In principle, we could have a separated PE for every layer of the CNN. However, for large CNNs, this might not be possible given the available resources. For this reason, our methodology includes the possibility to map multiple logical layers onto a single PE, so long as they implement a similar computation (that is, we cluster together in a single PE either layers from the features extraction part or fully-connected layers). This functionality is achieved with an additional outer loop that iterates through the implemented layers, and a set of conditionals to infer which input ports must be read (in case the window size changes). We can therefore spatially unfold the accelerator, if enough resources are available, to implement different layers as separated concurrent PEs, up to the point where there is a 1:1 mapping of layers onto PEs, thus exploiting full intra-layer parallelism. Moreover, we can exploit inter-layer parallelism reading multiple input feature maps concurrently and computing multiple output feature maps in parallel, as already done in [25]. Again, according to the amount of available resources, we can choose to implement a layer (or a set of layers) as a single-input/single-output port PE, where input feature maps are read sequentially and output feature map are equally serially computed, or increase the level of parallelism reading and processing multiple feature maps at once.

The memory access pattern for a single input feature map of a features extraction layer – either convolutional or sub-sampling – consists of a sliding window that spans through the entire input, meaning that to compute an output point a collection of input points must be read together. This data access pattern presents high data locality, that can be exploited to reduce the on-chip memory footprint. We do so using the non-uniform memory partitioning presented

2. https://github.com/aws/aws-fpga/blob/master/hdk/docs/on_premise_licensing_help.md

in [28]. In particular, for each feature map read in parallel we create a pipeline of filters interleaved by FIFOs. This collection of pipelines constitutes the *memory subsystem* for a features extraction PE. Within a pipeline, each filter represents an access to the input feature map (a point of the sliding window) and extract the elements from the input stream that belong to its data domain, sending them to the PE. It also sends each element read to the subsequent filter writing to the FIFO in between them. The FIFOs between filters realize the on-chip buffering and their size is equal to the spatial distance between the two accesses that the filters at each end of the FIFO represent. Such a structure allows for concurrent reads of all the elements of the sliding window, without any possibility of on-chip memory port contention. Moreover, it reduces the on-chip storage requirements, as only the elements that are spatially located in between the first and the last access are buffered on-chip, at any point in time. For this pipeline to work correctly without stalls, its filters are ordered in lexicographically inverse order according to the polyhedral model [28]. When multiple layers are fused together, the memory pipeline is created considering the layer with the biggest window size (that translates to a higher number of filters). The FIFOs size is instead determined considering the layer with the greatest input feature maps size. A set of conditionals within the filters then ensures that the pipeline works properly (correct number of active filters, correct filtering conditions) according to the currently "active" layer (the one that the PE is currently computing).

At current times, we implement fully-connected layers very similarly to what we have done previously in [25]. We will make some further considerations on how we implement these layers when we discuss the design automation flow in the following section.

3.3. Design Automation Flow

To have a better understanding of how the final accelerator is created, we here provide a description of the implemented automation flow. As stated previously, the user should specify the necessary input either as a pre-trained caffe model, providing the prototxt and caffemodel files, or use the Condor-specific format for network and weights. The output of the framework will either be an FPGA binary file that can be loaded on local hardware, or an AFI, that enables users to deploy the accelerator in the cloud, using the AWS F1 instances.
The proposed design automation flow consists of the following steps:

1) **Input Analysis**: the input files provided by the user are inspected in order to extract weights and network representation. The files from an external deep learning library, only Caffe as of now, are translated in the Condor format.
2) **Design Space Exploration**: as specified in Section 3.2 the accelerator has the ability to exploit different level of parallelism. In this phase, given the available FPGA resources, different configurations are explored to find the optimal tradeoff between resource consumption and performance. This phase is still not automated and therefore requires human intervention, but in the future, it will be performed automatically relying on resource consumption and performance models that are still under development.
3) **Creation of the features extraction stage**: Once we have the desired topology and all the weights, it is possible to begin the creation of the different layers, starting with the convolutional and sub-sampling layers. As specified before, each layer is composed of filters, FIFOs, and a PE. The following steps are performed for every layer in the features extraction stage:
   a **Characterization of the PE**: the C code performing the computation of the layer is automatically generated, and the PE is synthesized via Vivado HLS.
   b **Characterization of the filters**: the filters are designed in order to exploit the dataflow computational pattern of convolutions. Each filter is associated to a set of inequalities that are used to select which of the elements present in the input stream of the filter have to be sent to the PE to perform the current computation (convolution or pooling). Given the size of the sliding window and the size of the input image, the code for the filters is automatically generated and the filters are synthesized with Vivado HLS.
   c **Creation of the layer**: an empty Vivado IP Integrator project is created, the filters are first linked together to form the memory subsystem and then connected to the PE to form the final structure of the layer. Finally, the layer is packaged as a Vivado IP.
4) **Creation of the classification stage**: the structure of a fully-connected layer is similar to the structure of a convolutional layer, due to the similar output dependence on all the input values. In fact, a fully-connected layer performs Multiply-and-Accumulation (MAC) operations on all the input with the given weights, and this kind of execution can be described as a 1×1 convolution. The unitary window size of this layer makes the memory subsystem of the classification layers not needed. Moreover, due to the high number of input and output channel, if the layer is completely parallelized, it would increase too much the DSPs utilization. In order to face this two issues, we decided to implement a fully-connected layer as a single-input/single-output convolutional PE. Therefore, the only steps performed to create the fully-connected layers are the generation of the C code of the related PE and its synthesis using Vivado HLS.
5) **Connection of the layers**: all the IPs of the layers packaged in the previous steps are linked together following the specified topology to create the final CNN accelerator.
6) **Integration with SDAccel**: in order to allow developers to use the accelerator in SDAccel we need to express it as an OpenCL Kernel. An OpenCL kernel is the computational foundation of every OpenCL application since it defines the code that will be accelerated in hardware.

Normally, kernels in SDAccel are express directly in C/C++ using the OpenCL API. However, the Xilinx tool allows also for the possibility of using external RTL specifications as kernels. We use this opportunity in our approach, requiring the following additional steps:

a **Creation of the kernel description XML file**: in order for our IP to be used as an SDAccel kernel we need to create a kernel description XML file. This file contains basic information of the kernel such as the name and vendor as well as the communication interface with the host via an AXI4 master port and an AXI4-Lite slave port.

b **Package as Xilinx Object file**: the final step is to package the IP of the accelerator and the kernel XML together into a Xilinx Object file (.xo), so it can be processed by the SDAccel compiler.

7) **Deployment on board**: the final step of the framework is to generate the xclbin file. The xclbin file is a binary library of kernel compute units, in our case only of the single accelerator, that will be loaded together into an OpenCL context for a specific device. The kernel binaries are generated using XOCC, that creates custom logic based on the characteristics of the selected target device. We also generate and provide the user with a default host code to run and test the performance of the resulting accelerator. The user can use this code as is or edit and adapt it according to her needs.
8) **Creation of the AFI (for cloud deployment only)**: running the accelerator on F1 instances requires some additional steps to be performed. First, as in the previous step the xclbin is synthesized targeting the Xilinx UltraScale+ FPGA devices used on the F1 instances. Then, using the AWS command line interface the AFI generation process is started. The framework automatically generates the AFI inside a user-specified Amazon S3 Bucket and returns the AFI global ID, which is used to refer to an AFI from within an F1 instance. Once the AFI generation completes, it can be loaded on an FPGA slot of an F1 instance and executed.

4. Framework Evaluation

We evaluate the proposed automated framework deploying two CNNs on the F1 instances, with two different objectives. The first test case is the CNN used in [25] trained on the USPS dataset, and we refer to it as *TC1*. The achieved frequency is 100MHz, and the generated network processes each feature map sequentially but can exploit full intra-layers parallelism. With this test, our purpose was to validate the hardware generation process of Condor with respect to what we have previously done by hand. The second test case is LeNet, generated starting from a Caffe model[3] with an achieved frequency of 180MHz and again without parallel processing of the feature maps but full intra-layers parallelism. For LeNet, the objective was to validate the integration with Caffe.

3. https://github.com/BVLC/caffe/blob/master/examples/mnist/lenet.prototxt

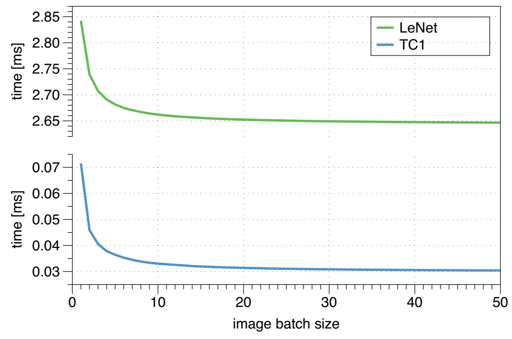

Figure 5. Mean time to process an image in relation to the images batch size.

The results regarding resource occupation, performance and power efficiency are reported in Table 1. The reader may find the tested AFIs and a guide on how to deploy them at https://bitbucket.org/necst/condor-aws-raw2018. Figure 5 shows instead how, in line to what we have experimented in our previous work, using the proposed architecture the mean time to process an image decreases as we increase the batch size, until convergence is reached, due to the high-level pipeline created exploiting the intra-layer parallelism. For both cases convergence is reached approximately when the batch size is bigger than the total number of layers of the network. As stated in Section 3.2, the refinements of the hardware design are still in process of development. We here report preliminary results of the sole features extraction part for TC1, LeNet and VGG-16, in Table 2. We are still investigating the optimization of the classification part. For instance, the fully-connected layers of VGG-16 would not be synthesizable with the current methodology. We intend to perform a more rigorous experimentation in the future, once the design methodology improvements are completed.

5. Conclusions and Future Work

In this paper, we have presented an end-to-end framework for the automatic dataflow implementation of CNNs on FPGAs, which implements the acceleration methodology first presented in [25]. This framework is integrated with Caffe and can be either used to deploy the implemented network on-premise with SDAccel or in the cloud using the Amazon F1 instances. We also provided an overview of the design improvements done to [25], with the caveat that the work is still developing and is not complete yet. Future work will address the finalization of the hardware design improvements, the integration of the proposed framework with other deep learning libraries and the automation of the design space exploration.

References

[1] O. Yadan, K. Adams *et al.*, "Multi-gpu training of convnets," *arXiv preprint arXiv:1312.5853*, 2013.

Table 1. AWS F1 DEPLOYMENT RESULTS

	LUT %	FF %	DPS %	BRAM %	GFLOPS	GFLOPS/W
TC1	10.47	9.02	5.63	0.97	8.36	1.56
LeNet	9.48	8.6	2.53	24.38	3.35	0.78

Table 2. PRELIMINARY RESULTS OF THE IMPROVED METHODOLOGY FOR THE FEATURES EXTRACTION PART

	TC1	LeNet	VGG-16
GFLOPS	16.56	53.51	113.30

[2] C. Szegedy, W. Liu *et al.*, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[3] 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.

[4] 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.

[5] Q. Li, W. Cai *et al.*, "Medical image classification with convolutional neural network," in *Control Automation Robotics & Vision (ICARCV), 2014 13th International Conference on*. IEEE, 2014, pp. 844–848.

[6] A. Sharif Razavian, H. Azizpour *et al.*, "Cnn features off-the-shelf: an astounding baseline for recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2014, pp. 806–813.

[7] A. Karpathy, G. Toderici *et al.*, "Large-scale video classification with convolutional neural networks," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2014, pp. 1725–1732.

[8] S. Lai, L. Xu *et al.*, "Recurrent convolutional neural networks for text classification." in *AAAI*, vol. 333, 2015, pp. 2267–2273.

[9] A. van den Oord, S. Dieleman, and B. Schrauwen, "Deep content-based music recommendation," in *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou *et al.*, Eds. Curran Associates, Inc., 2013, pp. 2643–2651. [Online]. Available: http://papers.nips.cc/paper/5004-deep-content-based-music-recommendation.pdf

[10] 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*. IEEE, 2010, pp. 317–324.

[11] D. C. Ciresan, U. Meier *et al.*, "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.

[12] A. Putnam, A. M. Caulfield *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 13–24, 2014.

[13] C. Zhang, P. Li *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.

[14] J. Qiu, J. Wang *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.

[15] N. Suda, V. Chandra *et al.*, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 16–25.

[16] C. Zhang, Z. Fang *et al.*, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference on*. IEEE, 2016, pp. 1–8.

[17] X. Wei, C. H. Yu *et al.*, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE, 2017, pp. 1–6.

[18] Y. Ma, Y. Cao *et al.*, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 45–54.

[19] Y. Chen, T. Luo *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.

[20] N. P. Jouppi, C. Young *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 1–12.

[21] E. Nurvitadhi, G. Venkatesh *et al.*, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 5–14.

[22] Forbes, "Microsoft: FPGA Wins Versus Google TPUs For AI," https://www.forbes.com/sites/moorinsights/2017/08/28/microsoft-fpga-wins-versus-google-tpus-for-ai/, 2017, [Online; accessed 07-December-2017].

[23] Y. Jia, E. Shelhamer *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[24] "Amazon EC2 F1 Instances," https://aws.amazon.com/it/ec2/instance-types/f1 (accessed: 5th of December 2017).

[25] M. Bacis, G. Natale *et al.*, "A pipelined and scalable dataflow implementation of convolutional neural networks on fpga," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE, 2017, pp. 90–97.

[26] D. H. Hubel and T. N. Wiesel, "Receptive fields and functional architecture of monkey striate cortex," *The Journal of physiology*, vol. 195, no. 1, pp. 215–243, 1968.

[27] Y. LeCun, L. Bottou *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[28] J. Cong, P. Li *et al.*, "An Optimal Microarchitecture for Stencil Computation Acceleration Based on Non-Uniform Partitioning of Data Reuse Buffer," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 77:1–77:6.