IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received October 12, 2020, accepted October 25, 2020, date of publication November 9, 2020, date of current version November 23, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3036975

Raising the Abstraction Level of a Deep Learning Design on FPGAs

DARÍO BAPTISTA[1,2,3], LEONEL SOUSA[1,3], (Senior Member, IEEE), AND FERNANDO MORGADO-DIAS[2,4], (Member, IEEE)

[1]Instituto Superior Tecnico, Universidade de Lisboa, 1649-001 Lisbon, Portugal
[2]ITI/Larsys and Madeira Interactive Technologies Institute (M-ITI), 9020-105 Funchal, Portugal
[3]INESC-ID Instituto de Engenharia de Sistemas e Computadores—Investigação e Desenvolvimento, DEEC, 1000-029 Lisbon, Portugal
[4]Ciências Exatas e Engenharia, University of Madeira (UMA), 9020-105 Funchal, Portugal

Corresponding author: Darío Baptista (dario.baptista@tecnico.ulisboa.pt)

This work was supported by the Portuguese Foundation for Science and Technology through LARSyS Projeto UIDB/50009/2020 and Project UIDB/50021/2020; ARDITI - Agência Regional para o Desenvolvimento da Investigação, Tecnologia e Inovação under the scope of the Project M1420-09-5369-FSE-000001-PhD Studentship; and also supported by the Project MITIExcell co-financed by Regional Development European Funds, for the Operational Programme "Madeira 14-20" - EIXO PRIORITÁRIO 1, of Região Autónoma da Madeira, with no. M1420-01-0145-FEDER-000002.

ABSTRACT Autonomous and intelligent systems based on deep learning, continuously attract the attention of researchers and engineers. With the progress on the application of deep learning for modern applications arises the challenge of reaching real-time processing. To face this challenge, Field Programmable Gate Arrays (FPGAs) can be used; however, deep learning generic implementations on an FPGA are still a topic of research. Advances in FPGA technology allow for designs based on High-Level Synthesis (HLS) for accelerating and facilitating implementations of complex problems on hardware. A platform based on HLS for emulating a generic parameterizable deep learning system on an FPGA is proposed in this paper, allowing for the implementation of any structure based on the following layers: convolution, max-pooling, batch-normalization, and fully connected networks. Through this platform, it is possible to implement a deep learning system on FPGAs using an N-Fold or a Flow architecture without the assistance of central processing units. Whereas the N-Fold architecture requires fewer hardware resources, as it re-uses resources, the Flow architecture presents a higher throughput. The developed platform improves the deep learning design productivity by automating the generation of the system, achieving efficiency and raising the level of abstraction, as was experimentally verified and evaluated.

INDEX TERMS Neural Network, machine learning, deep learning, systems architecture, flow architecture, N-Fold architecture.

I. INTRODUCTION

A large quantity of computation is required to analyze data based on deep learning [37]. Thus, the hardware required to implement a deep learning efficiently has been the subject of intensive investigation and investment, mostly targeting Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs) [7], [25], [26]. Even though GPU providers have positioned GPUs as the most performant devices for this new era, FPGAs are becoming widely adopted since they provide a better trade-off between performance and power consumption [25], making them efficient in real-time embedded systems.

The associate editor coordinating the review of this manuscript and approving it for publication was Szidónia Lefkovits.

The advance of FPGA technology makes their design a challenge to fully utilize their capacities to address complex problems [12]. The design methodology on FPGAs can be structured hierarchically over several levels of abstraction. To better understand the levels of abstraction, Gajski-Kuhn proposed in 1983 a model called the Y-chart, which is represented in Figure 1 [8], [14]. The three domains of the Y-chart are represented on its radial axes. Each of them can be divided into levels of abstraction, using concentric rings. The Y-chart has five concentric circles representing the following levels of abstraction [22]: Circuit level, Logical level, Register Transfer Level (RTL), Algorithmic level, and System level. Each level represents the information in different ways, with the amount of detail increasing from higher to lower levels [8].

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

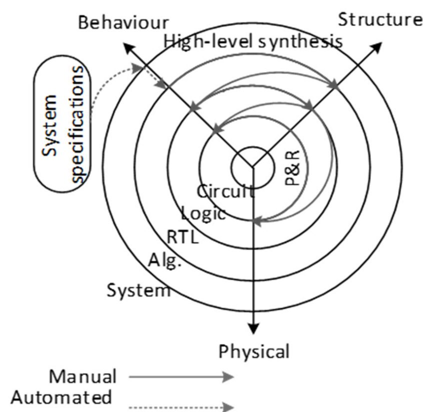


FIGURE 1. High-level synthesis in the Gasjki-Kuhn Y-Chart.

Previously, designers would manually refine the behavioral system specifications down to the RTL. From that point on, RTL synthesis would complete the design. Recently,however, High-level Synthesis (HLS) has improved design productivity by automating the refinement from the Algorithmic Level to the RTL [22]. Even though HLS can produce the design from the Algorithmic Level, a significant amount of time and a certain level of hardware design expertise are required to deploy deep learning on an FPGA. HLS tools require the algorithm to be specifically expressed in order to enable the synthesis tools to identify and exploit parallelism. Dealing with hardware description languages at a level of abstraction, as pure software, has been explored but often leads to inefficient use of the hardware. The key factor to bear in mind is that the FPGA has hardware characteristics that are not taken into consideration in a software-based design. Thus, each statement describes hardware that must be built rather than providing a set of instructions to be executed.

A novel way of solving this problem, proposed in this paper, is to increase the abstraction to the system level by providing a platform to deploy a generic parameterizable-based deep learning on an FPGA. Such a platform provides designers with a choice in choosing or developing their own deep learning framework while providing efficient implementations. To achieve this, designers have the freedom to provide the network topology description and the parameterization of each layer.

The proposed platform allows for the designing of Convolutional Neural Networks (CNNs), Deep Neural Networks (DNNs), and Artificial Neural Networks (ANNs). Two different architectures are provided to make the platform more efficient and useful, allowing the designer to easily adjust the achieved throughput and the required resources. In this way, the designer has the freedom to perform design space exploration, which allows for finer customization. Thus, the platform offers the advantage of optimization for different objectives to find the best solution for each application and according to the desired specifications. The main advantage of using such a platform is that the designer can easily implement deep learning on FPGAs without the necessity of an in-depth understanding of the underlying hardware.

II. BACKGROUND

CNNs are divided into two main components: feature extraction and classification components. The feature extraction component makes use of convolution, max-pooling and batch-normalization functions, while the classification component consists of fully-connected layers, common in ANNs. All layers of these networks can be organized into two sub-layers. The first one, called the main operation sub-layer, applies the main function, i.e. the convolution, the max-pooling, the batch-normalization, or the fully-connected operation. The second sub-layer, called the non-linear sub-layer, is supported on the hyperbolic tangent or by ReLU functions. ReLU has become the most successful and widely used non-linearity, given its simplicity and effectiveness [29]. Additionally, for classification models, a softmax non-linear function is commonly used in the last layer.

A convolutional layer contains a set of kernels whose parameters need to be learned for extracting multiple features, typically from an image [3]. The dimension of the kernel is relatively smaller than that of the input data. The kernel is slid across the width and height of the matrix input and the scalar products between the input and kernel are computed in every spatial position. Assuming the matrix F as an input ($F \in \mathbb{R}^{W\times W\times L}$, where $W \times W \times L$ corresponds to height × width × #input matrices), the matrix K as a kernel ($K \in \mathbb{R}^{H\times H\times L\times M}$, where $H \times H \times L \times M$ corresponds to height × width × #input matrices × #output matrices), and b as the bias, the equation for computing the calculation of each element in the output matrix ($C \in \mathbb{R}^{R\times R\times M}$, where $R\times R\times M$ corresponds to height × width × #outputs matrices) is given by equation (1).

$$C_{i,j,m} = \sum_{l=0}^{L-1}\sum_{p=0}^{H-1}\sum_{q=0}^{H-1} F_{i+p,j+q,l}K_{p,q,l,m} + b_m \tag{1}$$

The max-pooling layer "smoothes" and progressively down-samples the spatial size of the representation, to reduce the number of parameters and to control overfitting. The max-pooling layer operates independently on every matrix input and resizes it spatially without performing any learning. Typically, the maximum value within an $H \times H$ pooling window is selected, reducing the number of parameters to be learned in the next layer [24]. Thus, the expression to compute the max-pooling response of each element in the output matrix is given by equation (2).

$$P_{i,j,l} = \max_{p\in[0;H[\wedge q\in[0;H[} \left(F_{i+p,j+q,l}\right) \tag{2}$$

The most common form is a max-pooling layer with a 2×2 pooling window size and a stride of 2 along the width and height of each input matrix, discarding 75% of the activations. Each operation would, in this case, take a maximum value of four numbers. The batch-normalization layer normalizes each input matrix processed by the convolution layer. Using batch-normalization, the internal covariate shift is reduced and,

in consequence, the training of deep learning systems accelerates. Batch-normalization standardizes values, x_i, by calculating the mean μ_B and variance σ_B^2 over a mini-batch as shown in equation (3).

$$\widehat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \tag{3}$$

where ε is the numerical stability when the mini-batch variance is too small. To accommodate inputs with a mean equal to 0 and a variance equal to 1, the value of $\widehat{x_i}$ should be scaled and shifted as shown in equation (4). The offset, β, and scale factor, γ, are parameters that are updated during the training.

$$y_i = \gamma \widehat{x_i} + \beta \tag{4}$$

In a Fully Connected layer all the input elements from the previous layer are connected to all the neurons. Typically, equation (5) is used to compute the response of each of the neurons present in the layer. The activation function σ is a non-linear function applied to the weighted input sum to produce the response y.

$$y = \sigma(\sum_{i=1}^{n} x_i w_i + b) \tag{5}$$

III. HARDWARE ACCELERATORS FOR DEEP LEARNING

Ordinary processors are not efficient for deep learning; they can hardly meet performance requisites [9]. Since CNN processing exhibits parallelism at different levels, there has been a significant amount of work investigating efficient dedicated parallel systems [17], [28], [36], [46]. While spatial parallelism can be explored greatly on hardware, there are application fields for which the use of specific hardware is a requirement. These requirements could include speeding up the processing [26]. Many examples of neural vision systems, which likewise cannot be used while attached to computers, can be found in [34]. Several designs based on FPGA [45], GPU [7], [35] and Application-Specific Integrated Circuits (ASICs) [10] have been prototyped to implement high performance deep learning systems.

The first FPGAs, launched in the 1980s [39], allowed one to explore, in an extended way, spatial parallelism. FPGAs outperform the systems based on general purpose processors by eliminating the paradigm of sequential execution. Ku proposed high-level synthesis to implement ASICs, which have the advantage of possessing a very minutely controlled and optimized power consumption [19]. Nevertheless, ASICs are not suitable for application fields where the designs might need to be upgraded frequently or even occasionally. Deep learning is an example of a field in fast pace evolution, for which the re-configurability of the hardware is useful. Contrary to FPGAs, ASICs have very high Non-Recurring Expenditure (NRE) costs [21].

NVidia's Compute Unified Device Architecture (CUDA) platform, first announced in 2007, was the earliest widely adopted programming model for General Purpose computing on a GPU (GPGPU) [13]. The GPU architecture adopts a Single Instruction Multiple Thread (SIMT) approach, which is more efficient than general-purpose Central Processing Units (CPUs) when exploring data parallelism. However, the power consumption of GPUs is relatively high. Contrariwise, the FPGAs architecture allows one to explore parallelism without the limitations imposed by an SIMT. On the one hand, FPGAs allow for more flexibility and are more energy efficient than GPUs [7], [25]. GPUs, on the other hand, have become more attractive for system designers because, unlike FPGAs, an in-depth understanding of the underlying hardware is not required. To counteract this tendency, Xilinx has been making a considerable effort to mitigate these constraints by providing tools such as Vivado HLS. Vivado HLS is a tool which has greatly facilitated the implementation of custom logic in the programmable logic (PL), starting from a high-level description in C language, which can be automatically translated to HDL.

IV. RELATED WORK

Table 1 presents a comparison of the platforms found in the literature for supporting neural network models, according to the front ends, FPGAs supported, operation precision, and the necessity of off-chip microinstructions to perform the network. Note that the platforms support the convolution, pooling and fully-connected layers.

The Automatic Neural GEnerator (ANGE) is one of the first platforms for developing artificial neural networks [4], [30], [31]. The first version of this platform allows only the mapping of ANNs, but a second version extended it to DNNs as well (Those with only two hidden fully-connected layers). The ANGE tool uses Matlab and the System Generator from Xilinx. Currently, several approaches towards the direction of automated mapping CNNs to FPGAs have been proposed. Platforms such as Haddoc2 [1], DeepBurning [42], and DnnWeaver [33] generate fully synthesizable Verilog or VHDL as output. The evolution of the HLS tools enables the emergence of platforms such as fpgaConvNet [41], FINN [40] and FP-DNN [16]. ALAMO, DeepBurning, DNNWeaver and the proposed platform all support the normalization layer. Most of the platforms focus on the automated implementation of CNNs, except FINN, which is focused on Binarized Neural Networks (BNNs).

Most of the platforms have been integrated in existing deep learning software libraries and frameworks. Thus far, Caffe has been the best-supported framework for CNN-to-FPGA mapping. Conversely, the fact that the designer is limited to a specific framework is a disadvantage since other or even newer frameworks may gain prominence. The design methodology used for each platform can also be analyzed based on the necessity of microinstructions required to implement the network. It allows one to have processing engines controlled by software through microinstructions. This process corresponds to the sequential execution of the layers or set of layers in a time-sharing manner. On the other hand, it is possible to find some platforms that implement the network on hardware without off-chip microinstructions.

TABLE 1. A comparison of the characteristics of different platforms (C - Convolution; P - Pooling; F - Fully-connected; N - Normalization; NL - Non-Linear Functions).

Platform Name	Supported FPGA	Supported front-ends	Quantization	Supported Operations	Supported Models	System controlled by off-chip instructions
ANGE	Xilinx	Proprietary Input Format	Fixed	NL,F	ANN, DNN	No
fpgaConvNet	Xilinx	Caffe,Torch	Fixed	C,P,NL,F	CNN	No
DeepBurning	Xilinx	Caffe	Fixed	C,P,NL,F,N	CNN, RNN, DNN	No
Angel-Eye	Xilinx	Caffe	Fixed	C,P,NL,F	CNN, DNN	Yes
ALAMO	Intel	Caffe	Fixed	C,P,NL,F,N	CNN, DNN	Yes
Haddoc2	Xilinx,Intel	Caffe	Fixed	C,P,NL,F	CNN, DNN	No
DnnWeaver	Xilinx,Intel	Caffe	Fixed	C,P,NL,F,N	CNN, DNN	Yes
Caffeine	Xilinx	Caffe	Fixed	C,P,NL,F	CNN, DNN	Yes
Finn	Xilinx,Intel	Theano	Binary	C,P,NL,F	BNN	No
FP-DNN	Intel	TensorFlow	Fixed	C,P,NL,F	CNN, RNN, DNN	Yes
FFTCodeGen	Intel	Proprietary Input Format	Float	C,P,NL,F	CNN	Yes
Our Platform	Xilinx	Proprietary Input Format	Float	C,P,NL,F,N	CNN,DNN,ANN	No

Haddoc2 and the herein proposed platform store the trainable parameters on-chip but the supported model size is constrained by the storage resources of the target device. To circumvent this constraint, the proposed platform allows one to extend a CNN implementation to multiple FPGAs. In this case, the input is inserted in one device and the output is obtained from another device. The front-end is relevant to provide platform accessibility to the developers. Most of the platforms present a user graphical interface associated with the framework to develop and train the networks. As mentioned previously, Caffe has been the best-supported framework by CNN-to-FPGA automated tools. However, these platforms do not integrate different deep learning tools. Conversely, ANGE, FFTCodeGen, and the proposed platform have up to this point adopted custom front ends. With the proposed platform, the network description is held by the designer. Thus, this tool can reach a wider community of deep learning researchers and practitioners. Furthermore, the proposed platform is a unique platform that provides alternative architectures to design a deep learning network on an FPGA: N-Fold and Flow architectures. The N-Fold architecture is a new and original architecture presented by this platform.

V. ARCHITECTURES TO DESIGN DEEP NEURAL NETWORKS

Two different architectures are proposed to implement deep learning on an FPGA, namely the Flow architecture and the N-Fold architecture.

A. N-FOLD ARCHITECTURE

The N-Fold architecture is depicted in Figure 2. It is composed of two sub-layers, which are iterated N times over time to process the complete network. The first sub-layer implements the main operation while the second sublayer applies a non-linear function.

In this architecture, the network can only initialize a new process after the previous iteration of the process is completely through. Consequently, it provides high latency and low throughput. On synthesizing this architecture, the non-linear operation is shared, reducing the circuit area required. Figure 2 illustrates this approach. In the first sub-layer, the design commutes to the respective module where the main operation is applied. Kernels, Bias and Weighs represent values that are stored in the memory to execute the convolution and fully connected networks. In the second sub-layer, the design commutes the signal to the respective module to compute the non-linearity. Here, the nonlinear part has access to two different memories: one of them stores exponential values while the other stores hyperbolic tangent values. An attractive feature of the N-Fold approach is the possibility of implementing several layers in hardware sharing the nonlinear modules.

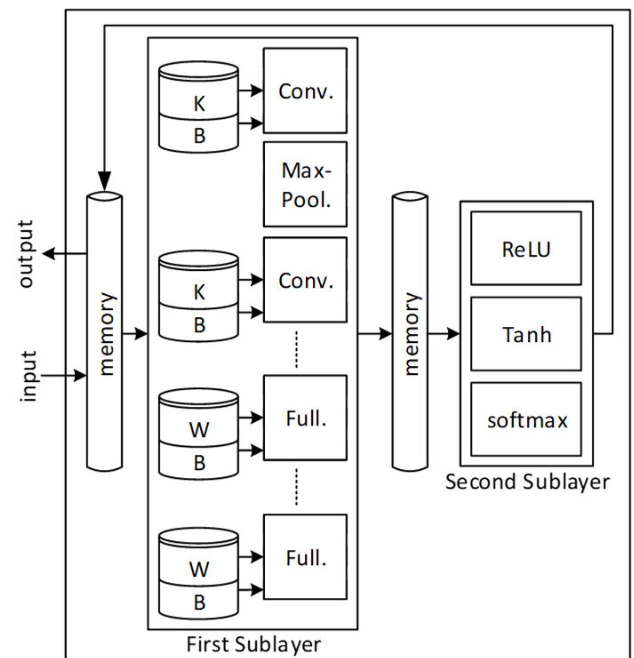


FIGURE 2. Design interpreted by HLS when the N-Fold architecture is used (Conv: convolution module; Max-Pool: max-pooling module; Full: fully-connected module; K: kernel, B: bias; W: weight; ReLU: ReLU module; Tanh: hyperbolic tangent; softmax: softmax module).

On the other hand, in Figure 2 one can also observe that the resulting output of each layer is temporarily stored. Even though the output of each sublayer is a 3D map with different sizes, a unique array was used to temporarily store these outputs during the application of the algorithm. Since the outputs resulting from the layers are temporarily buffered in a common memory, resources are saved since there is re-utilization among all the layers. The number of

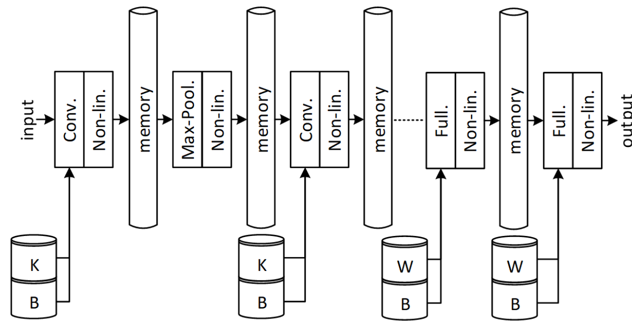


FIGURE 3. Flow architecture design (Conv: convolution module; Max-Pool: max-pooling module; Full: fully-connected module; Non-lin: Non-linear Module; K: kernel, B: bias; W: weight).

blocks RAM (BRAMs) required for data buffering between sub-layers can be calculated with equation (6),

$$N = \frac{N_{words} * N_{bits}}{S_{BRAM}} \quad (6)$$

where N_{words} is the number of values to store, N_{bits} is the number of bits used to represent a word, and S_{BRAM} is the capacity of a BRAM. To improve the performance, a dual-port RAM is used to allow read operations on one port and write operations on the other port. For this purpose, the directive < *#pragma HLS resources* > is used, specifying that array variable is mapped to the BRAMs. Each BRAM has two completely independent access ports. Each port has its own address, data in, data out, clock, clock enable, and write enable signals.

With the previous directive, the HLS tool considers the memory in Figure 2 as a single array and it is implemented as one large memory. The array representation becomes a bottleneck to achieve performance due to the limited memory ports. However, when an array is partitioned into multiple blocks, the single array is implemented as multiple RTL BRAMs. Partitioning helps with the performance, allowing an increase in the amount of read and write ports for the storage and, subsequently, an increase in the number of elements that can be accessed in parallel by the modules. Thus, there is a design trade-off between the performance and the number of RAMs required. Vivado HLS includes optimization directives for defining how arrays are implemented and accessed; the directive used in this architecture is < *#pragma HLS array_partition variable* = "*name*" *block factor* = "*int*" > where the *variable* = "*name*" specifies the array variable to be partitioned and the *factor* = "*int*" is the number of BRAMs used to implement the buffer.

B. FLOW ARCHITECTURE

A general view of the Flow architecture is illustrated in Figure 3. In the Flow architecture, the whole system is seen as a series of data transformations, where all operations are performed sequentially but individually independent of each other. This architecture emphasizes the incremental transformation of data by successive components. In this architecture, resources are not shared, which allows for pipelining at the cost of requiring more hardware and memory buffers between the layers. By using the directive < *#pragma HLS dataflow* >, the pipeline between the layers is implemented and, as a result, the flow data are optimized. In this case, data enter into the system and then flow through the layers across time in a data flow approach. If no directive is used, the Vivado HLS performs all the layers sequentially without pipeline operation.

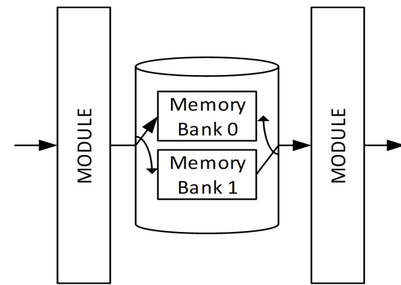


FIGURE 4. Ping-pong buffer (Double buffering).

When a pipeline between the layers is used, there may be a data conflict if a value is written through one port on a specific address and, at the same time, a value is read from the same address from the other port. To avoid conflicts between the layers in the pipeline, double buffering (also known as ping-pong buffering) is used as an intermedium memory. Two equal size memories with two independent access channels are shown in Figure 4. One memory bank is used to hold the previous data so that the forward layer can read it, while the backward layer creates and transfers new data to another memory bank. When the new data are completely transferred, the reader and writer layer alternates the two memory banks. The usage of double buffering increases the overall throughput of a CNN and helps to prevent bottlenecks.

The procedure to compute the number of BRAMs in each intermedium memory of the Flow architecture is similar to the one for the N-Fold architecture, but the result is multiplied by the number of memory banks, n_{bank}; for each intermedium memory, the n_{bank} is equal to 2.

VI. PROCESSING MODULES

Two different types of modules were designed to implement deep learning on an FPGA: the main operation modules and the non-linear modules.

A. MAIN OPERATION MODULES

The main operation modules process a feature map based on equation (1) for the convolution module, equation (2) for the max-pooling module, and equation (5) for the fully-connected layer module. The Batch-normalization layer computes each sample of the feature map by multiplying k_1 and adding k_2 (equation (7)). Note that equation (7) is derived from equations (3) and (4).

$$y_i = k_1 x_i + k_2, k_1 = \frac{\gamma}{\sqrt{\sigma_B^2 + \varepsilon}}; k_2 = \frac{\mu_B \gamma}{\sqrt{\sigma_B^2 + \varepsilon}} + \beta \quad (7)$$

The main operation modules are designed for HLS description by using optimization directives required to reach the

TABLE 2. Pseudo-code of the Main operation Modules, considering the directives for HLS Vivado 2016.4.

a) Convolution Module

```
N: layer size; NS: number of matrix inputs; KR×KC: kernel dimension; RO×CO: output matrix dimension; RI×CI: input matrix dimension;
C_Loop1: for (co=1 to CO) {
  C _Loop2: for (ro=1 to RO) { #pipeline directive
    C_Loop3: for (n=1 to N) { #unroll directive
      Out[n][ro][co] = bias[n]; // for Flow architecture
      Out[r+RO*(c+n*CO)] = bias[n]; // for N-Fold architecture
      C_Loop4: for (ns=1 to NS) { #unroll directive
        C_Loop5: for (r=1 to KR) { #unroll directive
          C_Loop6: for (c=1 to KC) { #unroll directive
            Out[r+RO*(c+n*CO)] += Kernel[n][ns][r][c] * Input[c+CI*(r+co+ns*RI)+ro]; // for N-Fold architecture
            Out[n][ro][co] += Kernel[n][ns][r][c] * Input[ns][r+ro*str][c+co*str]; //for Flow architecture
}}}}}}
```

b) Max-pooling Module

```
N: layer size; PR×PC: pooling-window dimension; RO×CO: output matrix dimension; RI×CI: input matrix dimension;
#define max (a, b) (((a) > (b))? (a) : (b))
MP_Loop1: for (ro=1 to RO) {
  MP_Loop2: for (co=1 to CO) { #pipeline directive
    MP_Loop3: for (n=1 to N) { #unroll directive
      Out[n][ro][co] = 0; // for Flow architecture
      Out[co+CO*(ro+n*RO)] = 0; // for N-Fold architecture
      MP_Loop4: for (r=1 to PR) { #unroll directive
        MP_Loop5: for (c=1 to PC) { #unroll directive
          Out[co+CO*(ro+n*RO)] = max (Out[co+CO*(ro+n*RO)],Input[(r+ro*PR+n*RI)*CI+co*PC+c]); //for N-Fold architecture
          Out[n][ro][co] = max (Out[n][ro][co],Input[n][r+ro*PR][c+co*PC]); //for Flow architecture
}}}}}
```

c) Fully-connected Module

```
N: layer size; NS: number of matrix inputs; IR×IC: input matrix dimension;
FC_Loop1: for (n=1 to N) { #pipeline directive
  Out[n] = bias[n]; // for Flow architecture and for N-Fold architecture
  FC_Loop2: for (ns=1 to NS) { #unroll directive
    FC_Loop3: for (c=1 to IC) { #unroll directive
      FC_Loop4: for (r=1 to IR) { #unroll directive
        Out[n] += Weight[n][ns][r][c] * Input [r+c*IR+ns*IC*IR]; //for N-Fold architecture
        Out[n] += Weight[n][ns][r][c] * Input[ns][r][c]; //for Flow architecture
}}}}
```

d) Batch-normalization Module

```
N: layer size; RO×CO: output matrix dimension;
N_Loop1: for (ro=1 to RO) {
  N_Loop2: for (co=1 to CO) { #pipeline directive
    N_Loop3: for (n=1 to N) { #unroll directive
      Out[n*RO*CO+co+ro*CO] = Out[n*RO*CO+co+ro*CO] * k1[n] + k2[n]; //for N-Fold architecture
      Out[n][ro][co] = Out[n][ro][co] * k1[n] + k2[n]; //for Flow architecture
}}}
```

best architecture. The objective of the design is to enable the efficient application of loop unrolling and hardware pipeline techniques, and thereby improve the performance while using the resources provided by the FPGA. The pseudo code, using loop unrolling and pipelining directives, is presented in Table 2.

A fundamental first step of HLS consists of detecting and resolving loop issues based on program directives for latency optimization (pipeline and unroll directives). The HLS tool determines the dependencies between computations and applies those techniques to achieve the specifications. Detecting such loop dependencies and applying transformation is a complex task. Loop dependencies can be classified as loop-independent and loop-carried dependencies. Loop-independent dependencies do not inhibit any parallelization of the outer loops, while loop-carried dependencies inhibit parallelization because the simultaneous execution of different iterations does not respect the dependencies.

At the algorithm level, the dimension of the feature maps processed by all the modules is different according to the architecture used, as can be verified from Table 2. The feature maps span through a three-dimensional (3D) space if the Flow architecture is used, whereas the feature maps are seen as a vector if the N-Fold architecture is used. Although BRAMs is used in both cases to store the intermedium data, these differences imply different interpretations of the HLS synthesizer. The Flow architecture allows, for instance, a parallelization of loop 3 in the Pseudocode of the convolution module

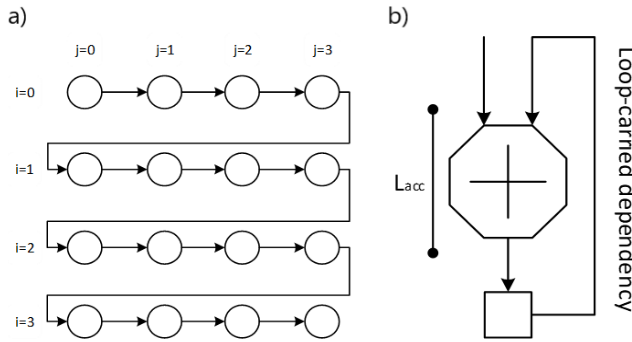


FIGURE 5. a) Example of a loop-carried dependency loop; b) Accumulation into a single register.

(see Table 2), because it has Loop-independent characteristics. For this purpose, the use of the directive "array partition" to the n loop-iterator is introduced at the top of the module. Thus, the buffer creates n smaller arrays from consecutive blocks of the original array. Consequently, direct and broadcast connections between the input and the processing elements are generated. In the N-Fold architecture, the resulting feature map is temporarily stored for each layer by using the same memory in the format of a vector, being the number of direct connections between the input and the processing elements dependent on the number of BRAMs needed to build the memory.

For both architectures, the loops from 6 to 4 in the pseudocode 1a) of Table 2 are examples of loop-carried dependencies, since each read operation cannot proceed until the write operation from the previous iteration is completed (Fig. 5a), so parallel calculation cannot be implemented. For example, in pseudocode 1a), the multiplication of the elements of Kernel and Input can be pipelined, but the respective addition requires the result of the addition in the previous iteration of the loop. This is a loop-carried dependency. The inner loop $c \in [1; KC]$ accumulates into a temporary register, which is written back to a temporary register at the end of each iteration $r \in [1; KR]$. In its turn, when these two loops are completed, it is accumulated and written back to a variable Out at the end of each iteration of $ns \in [1; NS]$. This is a common scenario when accumulating into a single register (Fig. 5b), in cases where the accumulation operation takes L_{acc} higher than 1 clock cycle (L_{acc} is the latency of a 32-bits floating point operation).

The carried dependencies loops are solved by a cascade of accumulations, allowing the pipeline to compute the output elements. The cascade of accumulations may generate interconnections with a complex topology; an iteration of a pipelined loop depends on a result produced by a previous iteration, accumulating partial results into registers.

B. NON-LINEAR MODULES

Three different non-linear modules are presented in this section: the hyperbolic tangent, the ReLU, and the softmax modules.

1) HYPERBOLIC TANGENT MODULE

The main challenge in designing the hyperbolic tangent module is in the range -6 and 6 of its domain, because for domain

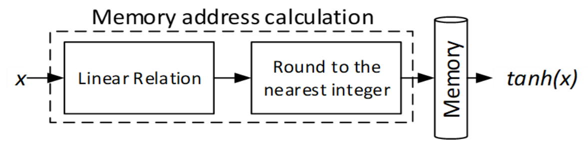


FIGURE 6. Implementation of the hyperbolic tangent module.

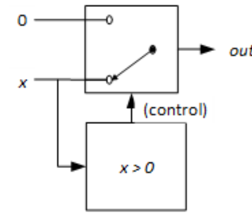


FIGURE 7. Implementation of the ReLU module .

$x > 6$ or $x < -6$ the hyperbolic tangent can be simply approached by $tanh(x) \approx 1$ or $tanh(x) \approx -1$, respectively. This function is an odd function [2], as it is, symmetric around the origin. Therefore, the function is computed for $x > 0$, and then for $x < 0$, and the hyperbolic tangent values are achieved by using equation (8).

$$tanh(-x) = -tanh(x) \tag{8}$$

The developed solution stores the hyperbolic tangent values for $0 \leqslant x < 6$ in memory, filling a Look Up Table (LUT). Figure 6 illustrates the circuit for computing the hyperbolic tangent, from the calculation of the memory address until the output of the value.

The calculation of the memory address from the x input value is based on a linear relation (equation (9)).

$$y = \lfloor (\frac{N_{tanh}}{x_1 - x_0})x \rceil \tag{9}$$

The slope, y, is given by the ratio between the number of memory elements, N_{tanh}, and the difference between the last, x_1, and the first, x_0, values of the domain ($x_1 = 6$ and $x_0 = 0$). Then, the quotient in equation (9) is calculated as the nearest integer of y to get the address, selecting the best value of the hyperbolic tangent.

2) ReLU MODULE

As depicted in Figure 7, a ReLU module provides an output equal to zero if the input is less than zero, otherwise, the output is equal to the input: if $x < 0$, then $\varphi(x) = 0$; otherwise $\varphi(x) = x$.

3) SOFTMAX MODULE

The softmax function is used at the last layer of a deep learning-based classifier [23] and it is given by equation (11).

$$P(y = j|x^{(i)}) = \varphi_{softmax}(x^{(i)}) = \frac{e^{x^{(i)}}}{\sum_{j=0}^{k} e^{x_k^{(i)}}} \tag{10}$$

This function, also named the normalized exponential function, is used for a categorical distribution representation, giving a probability distribution over k different probable outcomes. The developed architecture is presented in Figure 8.

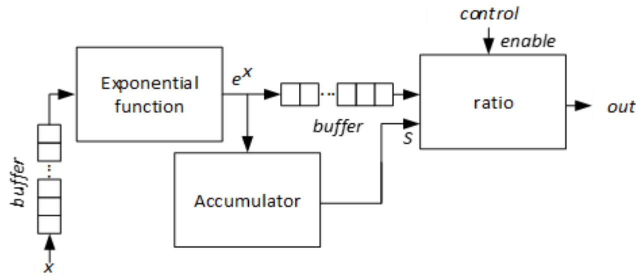


FIGURE 8. Implementation of the Softmax module.

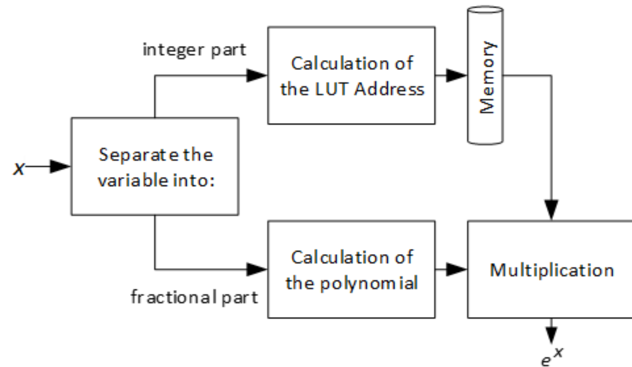


FIGURE 9. Implementation of the hybrid solution to compute the exponential function module.

The first step consists of calculating sequentially the exponential value of each input, x, with the result being stored in a buffer. Note that before storing these values into a buffer, the value of $S = \sum_{j=0}^{k} e^{x_k}$ is computed and accumulated. Then each exponential value is divided by S. To compute the exponential function on the hardware, a hybrid solution was used [18]. This solution decomposes the exponential function into an integer, int_x, and a fractional part, $frac_x$, of x, i.e.:

$$e^x = e^{int_x} \times e^{frac_x} \tag{11}$$

While e^{frac_x} is calculated by a polynomial interpolator, e^{int_x} is stored and uploaded from memory. Figure 9 shows a diagram with the adopted hybrid solution.

The int_x is used to compute the memory address. In this implementation, values between e^{-30} and e^{30} are stored in an LUT. The $frac_x$ is used to compute the value of e^{frac_x} through a 5th-order interpolating polynomial [2], [20].

VII. BENCHMARKING THE N-FOLD AND FLOW ARCHITECTURES

In this section, the two proposed architectures, the N-Fold and the Flow, are benchmarked. Three different networks, namely ANN, DNN, and CNN, are considered. For each network, the resources required to implement it, based on each architecture on a Kintex7 [43], are presented. Additionally, the latency and the throughput are presented. To further complement the results, the performance of the implementation of these networks on a GPU through the toolbox of Matlab is evaluated. The GPU is a GeForce Nvidia MX150 384 Compute Unified Device Architecture (CUDA).

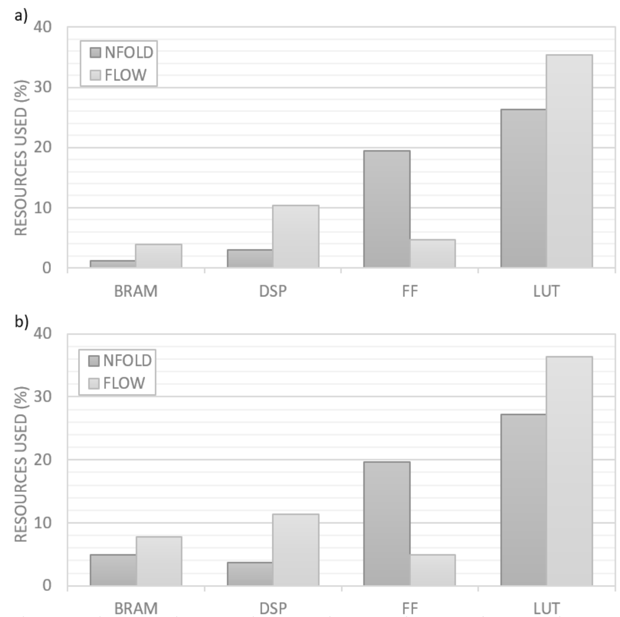


FIGURE 10. a) Resources required to implement the ANN-1; b) Resources required to implement the ANN-2 (LUT: Look-Up Table; FF: Flip-Flops; BRAM: Block of RAM; DSP: Digital Signal Processor).

TABLE 3. Description of the two ANNs for experimental assessment.

Layer	ANN-1[a]	ANN-2[a]
1st	Input image – I [32×32@1]	Input image – I [32×32@1]
2nd	FC[15] (Fully-connected module + ReLU module)	FC [15] (Fully-connected module + Hyperbolic Tan. module)
3rd	FC [10] (Fully-connected module + softmax module)	FC [10] (Fully-connected module + softmax module)

[a]FC[size] - Fully-connected Layer; I[size @ number of image] - Image;

A. ARTIFICIAL NEURAL NETWORK

Two ANNs, with the same topology, but using different non-linear functions in the hidden layer, are considered in the proposed architectures. The main characteristics of the two ANNs are presented in Table 3.

Figure 10 presents the resources required for implementing both ANNs, when adopting the different architectures. Figure 10 can be analyzed in two different perspectives: *i)* the resources required by both networks when the same architecture is used, and *ii)* the resources used by the two architectures when the same network is implemented.

Since an ANN contains just one hidden layer and one output layer, it is logical that the difference in resource utilization between both architectures is not large. However, as expected, an ANN implemented with a Flow architecture uses more resources than the same ANN implemented with an N-Fold architecture. The ANN-1 uses 13% and 14% of the FPGA resources for the N-Fold and Flow architectures, respectively, and the ANN-2 uses 14% and 15% of the FPGA resources for the N-Fold and Flow architecture, respectively. On the other hand, it can be observed that an ANN that applies the hyperbolic tangent as the non-linearity requires more resources than the same ANN using the ReLU as the non-linearity (1% increase of the total resources required). The increase in the total resources needed to implement an ANN using a hyperbolic tangent is essentially justified by the

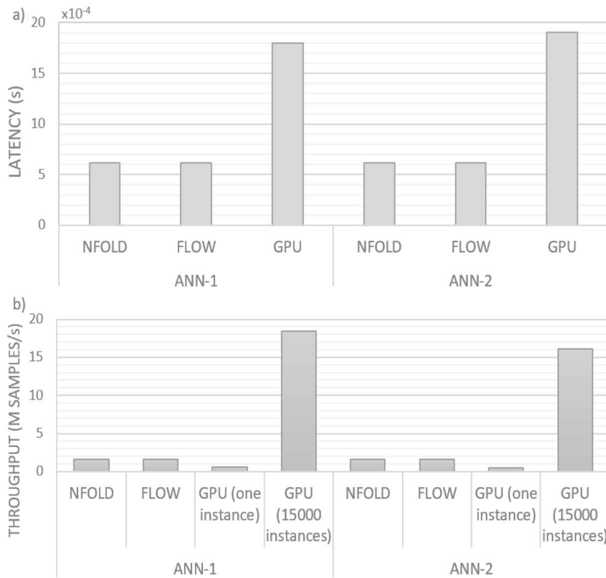


FIGURE 11. a) Latency of both ANNs using different architectures; b) Amount of images per second of both ANNs using different architectures.

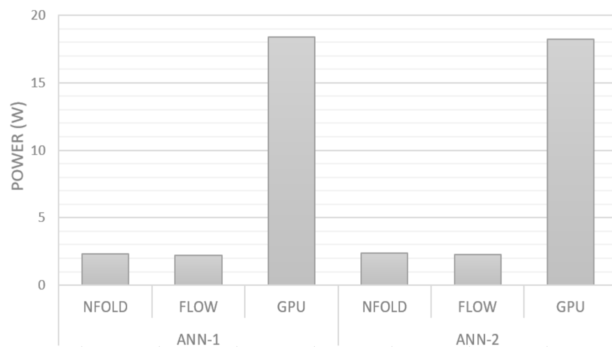


FIGURE 12. Power consumption of both ANNs implemented on a GPU and an FPGA using different architectures.

BRAMs used to store the hyperbolic tangent values and the DSPs applied to calculate the memory addresses.

Figure 11 presents the latency and the throughput of the implemented ANNs on a GPU and on an FPGA with the proposed architectures, respectively. Figure 12 presents the power consumption.

An ANN implemented with the N-Fold architecture has a latency slightly higher than the same ANN implemented with the Flow architecture. On the other hand, an ANN using a ReLU as non-linearity has a latency slightly lower than the same ANN using a hyperbolic tangent (not exceeding $0.1\mu s$). More time is required to calculate the memory address for accessing the hyperbolic tangent values. The pipeline effect between layers in the Flow architecture is not very noticeable, since the ANNs contains just one hidden layer and one output layer.

The GPU becomes more efficient when various instances of the networks are batched. For example, if 15000 instances of the ANN-1 are simultaneously processed in the GPU, the throughput of the GPU is 11.4× higher than that of

TABLE 4. Description of the two DNNs for experimental assessment.

Layer	DNN-1[a]	DNN-2[a]
1st	Input image – I [32×32@1]	Input image – I [32×32@1]
2nd	FC[15] (Fully-connected module + ReLU module)	FC [15] (Fully-connected module + Hyperbolic Tan. module)
3rd	FC[30] (Fully-connected module + ReLU module)	FC [30] (Fully-connected module + Hyperbolic Tan. module)
4th	FC[60] (Fully-connected module + ReLU module)	FC [60] (Fully-connected module + Hyperbolic Tan. module)
5th	FC [10] (Fully-connected module + softmax module)	FC [10] (Fully-connected module + softmax module)

[a]FC[size] - Fully-connected Layer; I[size @ number of image] - Image;

the FPGA. In the case of ANN-2, the throughput of the GPU is 9.96× higher than that of the FPGA. Larger batch sizes are almost always more efficient on GPUs since massive parallelism is explored to take advantage of all the GPU stream processors. However, sometimes batching inference work is not possible due to the characteristics of the application. In some common applications, such as a server that does inference per request, it is not possible to implement opportunistic batching. For each incoming request, one must wait for a time, and if other requests come in during that time, one must batch them together. Otherwise, one may continue with a single instance inference. From Figure 11, it is possible to verify that when exactly one instance of the ANN-1 is processed at a time in the GPU, the throughput is 2.9× higher when both architectures are used. In the case of ANN-2, the throughput is 3× higher when both architectures are used. As can be observed, situations where a single instance is processed do exist in practice but they are not suitable for using a GPU. Regardless of the number of instances entered simultaneously, the power consumption does not change significantly. This fact could be explained since a system combines CPUs and GPUs, and the GPU does not disconnect the resources which are not used. A comparison between the power consumption of the implementation solely on the GPU and the FPGA is presented in Figure 12. The power consumption when the ANNs are implemented on the GPU [38] through the toolbox of Matlab is higher than when the same networks are implemented with the proposed architectures. In the ANN-1 case, the GPU consumes 7.9× and 8.2× more power when the N-Fold and Flow are used, respectivaly. In the ANN-2 case, the GPU consumes 7.7× and 8.1× more power when the N-Fold and Flow are used, respectively.

B. DEEP NEURAL NETWORK

The DNNs in Table 4 were considered to apply the two proposed architectures.

Figure 13 presents the resources required for implementing both DNNs, by using the two proposed architectures, on FPGAs. Figure 13 shows that a DNN with 3 hidden layers implemented using the Flow architecture uses more resources than when the same DNN is implemented using the N-Fold architecture. The DNN-1 spends 14% and 31% of the total resources available when implemented with the N-Fold and Flow architectures, respectively. The total resources needed to implement the DNN-2 using an N-Fold and a

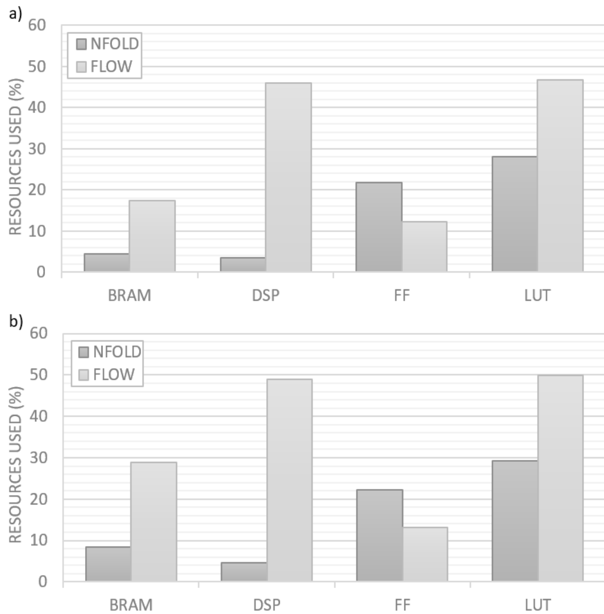


FIGURE 13. a) Resources required to implement the DNN-1; b) Resources required to implement the DNN-2 (LUT: Look-Up Table; FF: Flip-Flops; BRAM: Block of RAM; DSP: Digital Signal Processor).

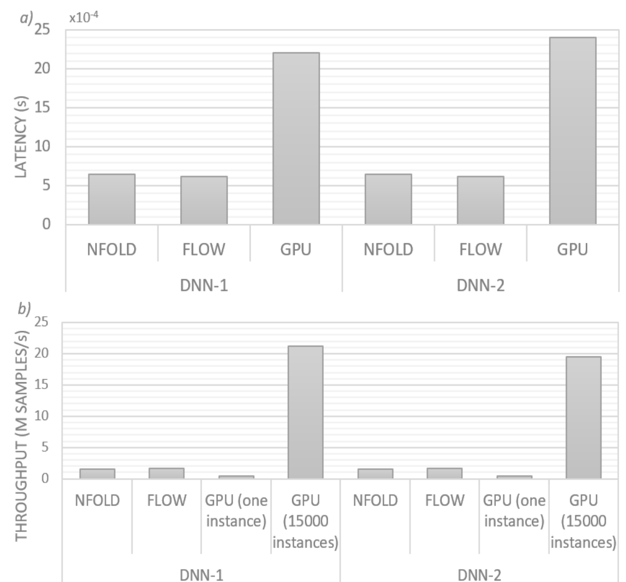


FIGURE 14. a) Latency of both DNNs using different architectures; b) Throughput of both DNNs using different architectures.

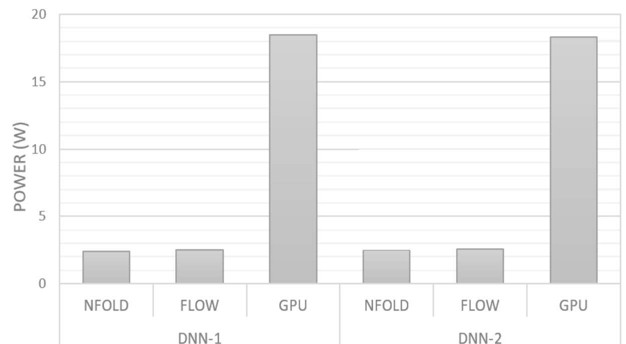


FIGURE 15. Power consumption of both DNNs implemented on a GPU and an FPGA using different architectures.

Flow architecture are 16% and 35%, respectively. Moreover, a DNN using the hyperbolic tangent as the non-linearity requires more resources than when using the ReLU as the non-linearity (an additional 2% in case of the N-Fold and 4% in case of the Flow architecture). Figure 14 presents the latency and the throughput of the DNNs implemented on a GPU and with the proposed architecture. Figure 15 presents the power consumption. In a generic overview, a DNN using a ReLU as the non-linearity (DNN-1) exhibits a latency slightly higher than the same DNN using the hyperbolic tangent as the non-linearity instead (DNN-2). On the other hand, the DNN implemented with the Flow architecture has a higher throughput than when implemented with the N-Fold architecture. The throughput increases by 5.1% for DNN-1 and 4.5% for DNN-2 when the Flow architecture is used. When 15000 instances of the DNN-1 are batched and simultaneously processed in the GPU, the throughput is 13.1× and 13.8× higher than the Flow and the N-Fold FPGA architectures, respectively. In the case of DNN-2, the throughput of the GPU is 12× and 12.6× higher than the Flow and the N-Fold architectures, respectively. Futhermore, can be verified that when exactly one instance of the DNN-1 is processed at a time in the GPU, the throughput is 3.4× higher and 3.6× higher when the N-Fold and the Flow architectures are used on an FPGA, respectively. In the case of DNN-2, the throughput is 3.7× higher and 3.9× higher when the N-Fold and the Flow architectures are used, respectively. The power consumption of the DNNs implemented on the GPU through the toolbox of Matlab is higher than that on the FPGA. In the DNN-1 case, the GPU consumes 7.6× and 7.4× more power if the N-Fold and the Flow architectures are used, respectively. In the DNN-2 case, the GPU consumes 7.5× and 7.2× more power if the N-Fold and the Flow architecture are used, respectively.

C. CONVOLUTION NEURAL NETWORKS

Two CNNs were chosen to experimentally evaluate each of the proposed architectures. The main features of the CNNs applied for the experimental assessment are presented in Table 5.

Figure 16 presents the FPGA resources required for implementing the CNN-1 and the NCNN-1 using the two proposed architectures. The CNN-1 implemented with the N-Fold architecture uses 39% of the total resources available, while the CNN-1 implemented with the Flow architecture uses 49% of the total resources. As expected, the CNN-1 designed with the N-Fold architecture uses fewer resources than those designed with the Flow architecture. In Figure 16b, we can see an extreme situation: the impossibility of "fitting" an network implemented with the Flow architecture due to the lack of resources. The NCNN-1 implementation using the N-Fold architecture requires around 47% of the total resources available. However, an impossibility of implementing the NCNN-1 is verified if the Flow architecture is used, due to the lack of LUTs and DSPs. In these situations, the N-Fold architecture is the only one that can be used for implementing the NCNN-1 in a single FPGA.

TABLE 5. Description of the two CNNs for experimental assessment.

Layer	CNN-1[a]	NCNN-1[a]
1st	Input image – I [32×32@1]	Input image – I [32×32@1]
2nd	C[6@5×5|str=1] (Conv. module + ReLU module)	C[8@5×5|str=1] (Conv. module + ReLU module)
3rd	MP[6@2×2|str=2] (max-pool. module + ReLU module)	N[8] (Norm. module + ReLU module)
4th	C[8@5×5|str=1] (Conv. module + ReLU module)	MP[8@2×2|str=2] (max-pool. module + ReLU module)
5th	MP[8@2×2|str=2] (max-pool. module + ReLU module)	C[12@3×3|str=1] (Conv. module + ReLU module)
6th	C[10@3×3|str=1] (Conv. module + ReLU module)	N[12] (Norm. module + ReLU module)
7th	FC[10] (Fully-connected module + SoftMax module)	MP[12@3×3|str=3] (max-pool. module + ReLU module)
8th	-	C[12@3×3|str=1] (Conv. module + ReLU module)
9th	-	N[12] (Norm. module + ReLU module)
10th	-	FC[10] (Fully-connected module + softmax module)

[a]FC[size] – Fully-connected Layer; I[size @ number of image] - Image; C [size layer @kernel window | stride] - Convolution Layer; P [size layer @ pooling window]-Max-pooling Layer; N[size] – Normalization Layer;

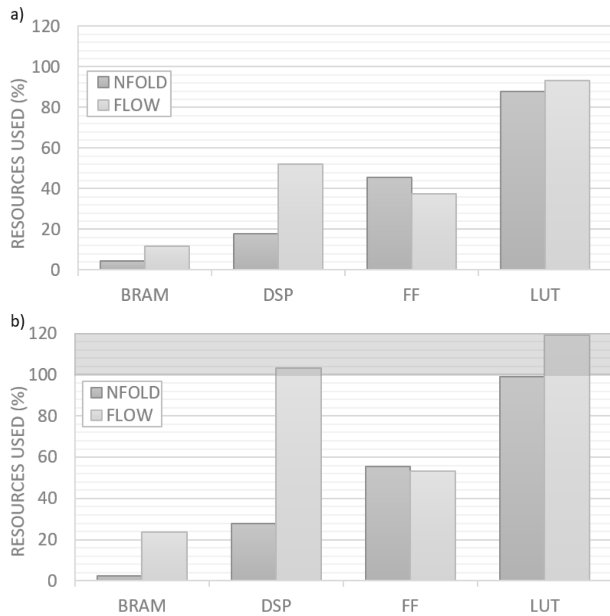


FIGURE 16. a) Resources required to implement the CNN-1; b) Resources required to implement the NCNN-1 (LUT: Look-Up Table; FF: Flip-Flops; BRAM: Block of RAM; DSP: Digital Signal Processor).

Figure 17 presents the latency and the throughput of the CNN-1 and the NCNN-1 implemented on a GPU, and the CNN-1 and the NCNN-1 implemented with the proposed architecture, respectively, with Figure 18 presenting the corresponding power consumption.

With this experience, one may come to realize the advantage of using the Flow architecture, in terms of performance improvement. The Flow architecture achieves significantly higher throughput than the N-Fold architecture. The increase of throughput is 2.12× and 4.44× for the CNN-1 and the NCNN-1, respectively. The GPU becomes more efficient when various instances are batched. For example, if 15000 instances of the CNN-1 are simultaneously

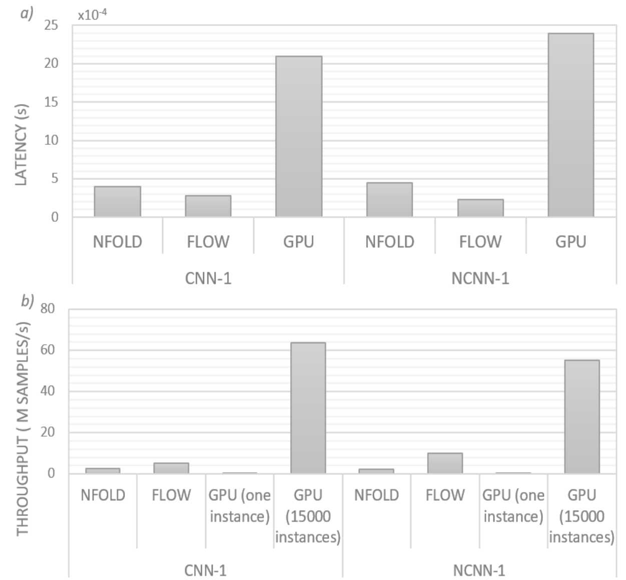


FIGURE 17. a) Latency of the CNN-1 and the NCNN-1 using different architectures; b) Throughput of the CNN-1 and the NCNN-1 using different architectures.

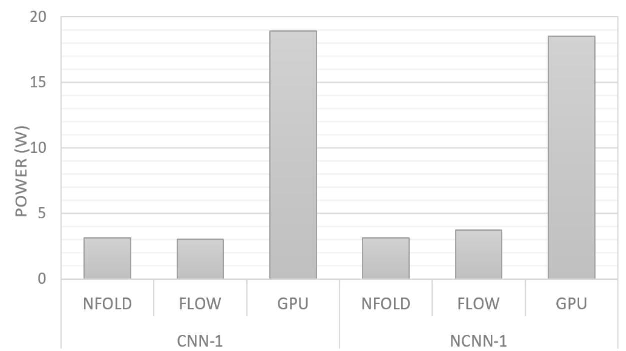


FIGURE 18. Power consumption of both DNNs implemented on a GPU and an FPGA using different architectures.

processed in the GPU, the throughput of the GPU is 12.5× and 26.4× higher than the Flow and the N-Fold architectures, respectively. In the case of the NCNN-1, the throughput of the GPU is 5.6× and 25× higher than the Flow and the N-Fold architectures, respectively. It can be verified that when only one instance of the CNN-1 is processed at a time in the GPU, the throughput is 5.1× higher and 10.7× higher when the N-Fold and the Flow FPGA architectures are used, respectively. In the case of the NCNN-1, the throughput is 5.3× and 23.52× higher than when the N-Fold and the Flow architectures are used, respectively. As can be is observed, situations where a single instance is processed exist in practice but are not suitable for using a GPU. The power consumption of the CNN-1 and the NCNN-1 implemented on a GPU through the toolbox of Matlab is higher than the same networks implemented with the proposed architectures. In the CNN-1 case, the GPU consumes 6.1× and 6.2× more power if the N-Fold and the Flow architecture are used, respectively. In the NCNN-1 case, the GPU consumes 5.9× and 4.9× more power if the N-Fold and the Flow architectures are used, respectively.

TABLE 6. A comparison with the published results.

	FPGA	$f(MHz)$	Quantization	GOP/s
[11]	Spartan3A	125	Fixed	5.25
[5]	Virtex5	200	Fixed	16.0
[32]	Virtex5	115	Fixed	6.74
[44]	Virtex7	100	Float	61.62
[6]	Virtex5	200	Fixed	7.0
[27]	Virtex6	150	Fixed	17
[15]	Zynq7	150	Fixed	23.18
[41]	Zynq7	100	Fixed	12.73
N-Fold	Kintex7	100	Float	6.3
Flow	Kintex7	100	Float	20.7

VIII. COMPARATIVE EVALUATION WITH STATE-OF-THE-ART

The CNN applied for the performance comparison between the proposed architectures and the state-of-the-art FPGA implementations has in the first layer 11 convolutions, with 3×3 kernels and 3 feature maps as input, producing maps of size 24×24. The next layer computes 12 high-level features by performing 3×3 convolutions. The third layer performs 2×2 spatial pooling. The fourth layer performs 3×3 convolutions resulting in 10 feature maps followed by a 2×2 pooling layer. Finally, the last layer is a linear classifier having 10 neurons and applying the softmax function as the activation function. Two different implementations were tested, one of them supported on the N-Fold architecture and the other one on the Flow architecture. On one hand, 49% of the total resources are used to implement the CNN with the Flow architecture. On the other hand, 43% of the total resources are used to implement the CNN with the N-Fold architecture.

Table 6 compares the performance of some CNNs from state-of-the-art implemented on the FPGAs with the one implemented with the platform investigated in this paper.

Most of the implementations use fixed-point arithmetic, while the implementation proposed in this Thesis uses floating-point arithmetic. Nevertheless, [44] presents an implementation that uses floating-point but, in contrast with the solution proposed in this Thesis, requires external memory to store/retrieve kernels and on-/off-chip interconnect. As a result, this makes both proposed architectures in this Thesis more cost-effective.

In general, the two architectures developed in this work presents a competitive throughput in comparison to the existing solutions in Table 6. However, the solution presented in [44] is the most noteworthy as this implementation presented a throughput of 61.62 GOP/s, which is better than the implementation using the Flow architecture. One reason which could explain why the proposed architecture does not achieve a higher throughput is directly related to the number of access ports available in the memory between the layers. In [44] the CNN design is composed of PEs, an on-chip buffer, external memory, and on-/off-chip interconnect. All the intermedium data are stored in the external memory and the PE is the basic computation unit for convolution which is tiled to fit in the PL part. The tile data are first transferred from the external memory to the on-chip buffer before being fed to PEs. This buffer contains several independent buffer banks and the number of these buffer banks is equal to the number of inputs in the tile data. In this way, it is possible to access all the inputs simultaneously to process the tile, increasing its calculation speed. In the case of this paper's research, the intermedium data between the layers are stored in a buffer. The size of that buffer is sufficient to allocate all the intermediate values, but each input does not have an exclusive port. The fact that there are not enough ports to simultaneously access all intermediate values may cause a delay in the processing values. On the other hand, the implementation using the N-Fold architecture ranks in a favorable position, considering that the N-Fold architecture saves resources at the cost of throughput.

IX. CONCLUSION

A platform to deploy a generic parameterizable-based deep learning on an FPGA was proposed in this paper. In this platform, the parameterization of the networks is applied to easily design a deep learning system that adopts convolution, max-pooling, batch-normalization, and fully connected layers. The parameterization provides tools to the designer for configuring a deep learning in a "lego" approach and deploying it automatically on an FPGA.

Two different architectures were proposed in this paper to design and implement those networks on FPGAs, namely the N-Fold architecture and the Flow architecture. While the N-Fold architecture is composed of a single hardware layer, which iterates N times for the multiple layers, the Flow architecture processes "flows" between hardware layers that operate in a pipeline way. In both architectures, each layer is composed of two different modules: the main operation and the non-linearity modules. Loop unrolling and pipelining were applied to improve the performance and the efficiency of the networks synthesized with HLS.

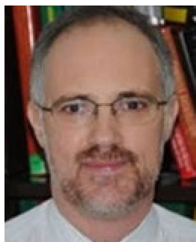
The performance of the networks implemented with the proposed architectures has been evaluated and compared with the performance achieved with GPUs. The proposed architectures present a better performance when compared with the implementation on a GPU. Comparing both architectures, the N-Fold architecture requires fewer FPGA resources than the Flow architecture, since the N-Fold reuses resources. On the other hand, the performance of networks using the Flow architecture is higher than those using the N-Fold architecture. The deeper the network, the more significant the increase in latency and the decrease of the throughput of the networks implemented with the N-Fold architecture. This characteristic is presented as an advantage of Flow over the N-Fold architecture. However, the herein architecture requires all the weights and kernels to be stored on-chip and, as consequence, the supported model size is constrained by the storage resources of the target device.

REFERENCES

[1] K. Abdelouahab, C. Bourrasset, M. Pelcat, F. Berry, J.-C. Quinton, and J. Serot, "A holistic approach for optimizing DSP block utilization of a CNN implementation on FPGA," in *Proc. 10th Int. Conf. Distrib. Smart Camera (ICDSC)*, 2016, pp. 69–75.

[2] D. Baptista and F. Morgado-Dias, "Low-resource hardware implementation of the hyperbolic tangent for artificial neural networks," *Neural Comput. Appl.*, vol. 23, nos. 3–4, pp. 601–607, Sep. 2013.
[3] D. Baptista, F. Morgado-Dias, and L. Sousa, "Configurable N-fold hardware architecture for convolutional neural networks," in *Proc. Int. Conf. Biomed. Eng. Appl. (ICBEA)*, Jul. 2018, pp. 1–8.
[4] F. D. Baptista and F. Morgado-Dias, "Automatic general-purpose neural hardware generator," *Neural Comput. Appl.*, vol. 28, no. 1, pp. 25–36, Jan. 2017.
[5] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf, "A programmable parallel accelerator for learning and classification," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, 2010, p. 273.
[6] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, p. 247, 2010.
[7] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding performance differences of FPGAs and GPUs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2018, pp. 93–96.
[8] J. C. de Sousa, "Projecto e síntese em alto nível de circuitos digitais," Ph.D. dissertation, Univ. de Brasília-UnB, Brasília, Brazil, 2017.
[9] *Hitting the Accelerator: The Next Generation of Machine-Leaning Chips*, Deloitte, London, U.K., 2017.
[10] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *Proc. IEEE Int. Symp. Circuits Syst.*, May 2010, pp. 257–260.
[11] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "CNP: An FPGA-based processor for convolutional networks," in *Proc. Int. Conf. Field Program. Log. Appl.*, Aug. 2009, pp. 32–37.
[12] D. D. Gajski, A. Gerstlauer, S. Abdi, and G. Schirner, *Embedded System Design: Modeling, Synthesis and Verification*. Boston, MA, USA: Springer, 2009.
[13] J. Ghorpade, "GPGPU processing in CUDA architecture," *Adv. Comput., Int. J.*, vol. 3, no. 1, pp. 105–120, Jan. 2012.
[14] G. Gielen and W. Sansen, *Symbolic Analysis for Automated Design of Analog Integrated Circuits*. Boston, MA, USA: Springer, 1991.
[15] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 G-ops/s mobile coprocessor for deep neural networks," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Workshops*, Jun. 2014, pp. 696–701.
[16] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *Proc. IEEE 25th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2017, pp. 152–159.
[17] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 243–254.
[18] I. Nascimento, R. Jardim, and F. Morgado-Dias, "A new solution to the hyperbolic tangent implementation in hardware: Polynomial modeling of the fractional exponential part," *Neural Comput. Appl.*, vol. 23, no. 2, pp. 363–369, Aug. 2013.
[19] C. Ku, "Size, speed, and power analysis for application specific integrated circuits using synthesis," M.S. thesis, Univ. Tennessee, Knoxville, TN, USA, 2003.
[20] J. C. Mason and D. C. Handscomb, *Chebyshev Polynomials*. London, U.K.: Chapman & Hall, 2003.
[21] M. Mason, "Considering the total cost of FPGAs," *Elektron*, vol. 22, no. 6, pp. 24–26, 2005.
[22] W. Meeus, K. Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," *Design Automat. Embedded Syst.*, vol. 16, pp. 31–51, Aug. 2012.
[23] R. Memisevic and C. Zach, "Gated softmax classification," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2011, pp. 1–9.
[24] J. Nagi, F. Ducatelle, G. A. Di Caro, D. Ciresan, U. Meier, A. Giusti, F. Nagi, J. Schmidhuber, and L. M. Gambardella, "Max-pooling convolutional neural networks for vision-based hand gesture recognition," in *Proc. IEEE Int. Conf. Signal Image Process. Appl. (ICSIPA)*, Nov. 2011, pp. 342–347.
[25] E. Nurvitadhi, S. Subhaschandra, G. Boudoukh, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. G. Hock, Y. T. Liew, K. Srivatsan, and D. Moss, "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?" in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2017, pp. 5–14.
[26] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Res.*, vol. 2, no. 11, pp. 3–6, 2015.
[27] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Proc. IEEE 31st Int. Conf. Comput. Design (ICCD)*, Oct. 2013, pp. 13–19.
[28] J. Qiu, S. Song, Y. Wang, H. Yang, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, and N. Xu, "Going deeper with embedded FPGA platform for convolutional neural network," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2016, pp. 26–35.
[29] P. Ramachandran, B. Zoph, and Q. V. Le, "Swish: A self-gated activation function," 2017, *arXiv:1710.05941v1*. [Online]. Available: https://arxiv.org/abs/1710.05941v1
[30] L. Reis, L. Aguiar, D. Baptista, and F. Morgado-Dias, "A software tool for automatic generation of neural hardware," *Int. Arab J. Inf. Technol.*, vol. 11, no. 3, pp. 229–235, 2014.
[31] L. Reis, L. Aguiar, D. Baptista, and F. M. Dias, "ANGE: Automatic neural generator," in *Proc. 21st Int. Conf. Artif. Neural Netw. (ICANN)*. Berlin, Germany: Springer-Verlag, 2011, pp. 446–453.
[32] 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 *Proc. 20th IEEE Int. Conf. Appl.-Specific Syst., Archit. Processors*, Jul. 2009, pp. 53–60.
[33] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to FPGAs," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Oct. 2016, pp. 1–12.
[34] T. Stieglits and J.-U. Meyer, "Biomedical microdevices for neural implants," in *Proc. BioMEMS*, vol. 16, 2007, pp. 71–137.
[35] D. Strigl, K. Kofler, and S. Podlipnig, "Performance and scalability of GPU-based convolutional neural networks," in *Proc. 18th Euromicro Conf. Parallel, Distrib. Netw.-Based Process.*, Feb. 2010, pp. 317–324.
[36] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-S. Seo, and Y. Cao, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2016, pp. 16–25.
[37] V. Sze, Y.-H. Chen, J. Emer, A. Suleiman, and Z. Zhang, "Hardware for machine learning: Challenges and opportunities," in *Proc. IEEE Custom Integr. Circuits Conf. (CICC)*, Apr. 2018, pp. 1–8.
[38] *GPU-Z v2.33.0*, TechPowerUp, New York, NY, USA, Jul. 2020.
[39] S. M. Trimberger, "Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology," *Proc. IEEE*, vol. 103, no. 3, pp. 318–331, Mar. 2015.
[40] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2017, pp. 65–74.
[41] S. I. Venieris and C.-S. Bouganis, "FpgaConvNet: A framework for mapping convolutional neural networks on FPGAs," in *Proc. IEEE 24th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2016, pp. 40–47.
[42] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "DeepBurning: Automatic generation of FPGA-based learning accelerators for the neural network family," in *Proc. 53rd Annu. Design Automat. Conf. (DAC)*, 2016, pp. 1–6.
[43] *Xilinx UG810—KC705 Evaluation Board for the Kintex-7 FPGA User Guide*, Xilinx, San Jose, CA, USA, 2019.
[44] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2015, pp. 161–170.
[45] X. Zhang, S. Das, O. Neopane, and K. Kreutz-Delgado, "A design methodology for efficient implementation of deconvolutional neural networks on an FPGA," 2017, *arXiv:1705.02583*. [Online]. Available: http://arxiv.org/abs/1705.02583
[46] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional neural networks with software-programmable FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2017, pp. 15–24.

DARÍO BAPTISTA received the master's degree in telecommunications and networks from the University of Madeira, Portugal, in 2009. He is currently pursuing the Ph.D. degree called NETSyS with the Instituto Superior Técnico, Lisbon. He has been involved in research projects, since 2010, with the Madeira Interactive Technologies Institute and the Centre of Exact Sciences and Engineering of the University of Madeira. His research interests include automation, artificial neural networks, deep learning, and field programmable gate array implementations.



LEONEL SOUSA (Senior Member, IEEE) received the Ph.D. degree in electrical and computer engineering from the Instituto Superior Técnico (IST), Universidade de Lisboa (UL), Lisbon, Portugal, in 1996. He is currently a Full Professor with Universidade de Lisboa (UL). He is also a Senior Researcher with the Research and Development Instituto de Engenharia de Sistemas e Computadores (INESC-ID). His research interests include VLSI architectures, computer architectures, parallel computing, computer arithmetic, and signal processing systems. He has contributed to more than 200 papers in journals and international conferences, for which he got several awards, such as DASIP'13 Best Paper Award, SAMOS'11 'Stamatis Vassiliadis' Best Paper Award, DASIP'10 Best Poster Award, and the Honorable Mention Award UTL/Santander Totta for the quality of the publications, in 2009. He has contributed to the organization of several international conferences, namely as program chair and as general and topic chair, and has given keynotes in some of them. He has edited four special issues of international journals, and he is currently Senior Editor of the IEEE JETCAS, Associate Editor of the IEEE TRANSACTIONS ON COMPUTERS, IEEE ACCESS, and Springer JRTIP. He is Fellow of the IET and Distinguished Scientist of ACM.

FERNANDO MORGADO-DIAS (Member, IEEE) received the master's degree in microelectronics from the University Joseph Fourier, Grenoble, France, in 1995, and the Ph.D. degree from the University of Aveiro, Portugal, in 2005. He is currently an Assistant professor with the University of Madeira and a Researcher with the Madeira Interactive Technologies Institute and Larsys/ITI. His research interests include renewable energy, artificial neural networks, and FPGA implementations. His affiliation is now at the University of Madeira, ITI/Larsys, and Madeira Interactive Technologies Institute.