Received June 11, 2021, accepted June 14, 2021, date of publication June 17, 2021, date of current version June 29, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3090196

User Driven FPGA-Based Design Automated Framework of Deep Neural Networks for Low-Power Low-Cost Edge Computing

TAREK BELABED[1,2,3], **MARIA GRACIELLY F. COUTINHO**[4], **MARCELO A. C. FERNANDES**[4], **CARLOS VALDERRAMA SAKUYAMA**[1], **(Senior Member, IEEE), AND CHOKRI SOUANI**[5]

[1]Université de Mons, Faculté Polytechnique, SEMi, 7000 Mons, Belgium
[2]Université de Sousse, Ecole Nationale d'Ingénieurs de Sousse, Sousse 4000, Tunisia
[3]Université de Monastir, Faculté des Sciences, Laboratoire de Microélectronique et Instrumentation, Monastir 5019, Tunisia
[4]Department of Computer and Automation Engineering, Federal University of Rio Grande do Norte, Natal 59078-970, Brazil
[5]Université de Sousse, Institut Supérieur des Sciences Appliquées et de Technologie de Sousse, Sousse 4003, Tunisia

Corresponding author: Tarek Belabed (belabed.tarek@gmail.com)

ABSTRACT Deep Learning techniques have been successfully applied to solve many Artificial Intelligence (AI) applications problems. However, owing to topologies with many hidden layers, Deep Neural Networks (DNNs) have high computational complexity, which makes their deployment difficult in contexts highly constrained by requirements such as performance, real-time processing, or energy efficiency. Numerous hardware/software optimization techniques using GPUs, ASICs, and reconfigurable computing (i.e, FPGAs), have been proposed in the literature. With FPGAs, very specialized architectures have been developed to provide an optimal balance between high-speed and low power. However, when targeting edge computing, user requirements and hardware constraints must be efficiently met. Therefore, in this work, we only focus on reconfigurable embedded systems based on the Xilinx ZYNQ SoC and popular DNNs that can be implemented on Embedded Edge improving performance per watt while maintaining accuracy. In this context, we propose an automated framework for the implementation of hardware-accelerated DNN architectures. This framework provides an end-to-end solution that facilitates the efficient deployment of topologies on FPGAs by combining custom hardware scalability with optimization strategies. Cutting-edge comparisons and experimental results demonstrate that the architectures developed by our framework offer the best compromise between performance, energy consumption, and system costs. For instance, the low power (0.266W) DNN topologies generated for the MNIST database achieved a high throughput of 3,626 FPS.

INDEX TERMS Deep learning, electronic design automation, edge computing, FPGA, low power systems.

I. INTRODUCTION

In the last half a century, many researches focus on building computational models allowed to exhibit what we call intelligence [1]–[5]. Since the beginning of artificial intelligence (AI) until these days [6], [7], much progress has been made. The Artificial Neural Network (ANN) [8], [9] emergency marked a breakthrough in AI, with the artificial neuron model to acquire knowledge based on the human brain. ANN techniques can be applied to many problems in many fields, such as classification and prediction problems. One improvement of ANN techniques is the Deep Neural Network (DNN) topology, also called Deep Learning technique.

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

Deep Learning (DL) [10], [11] is a particular type of machine learning [12] that explores the use of many non-linear processing layers to extract features in a supervised or unsupervised manner [13], [14]. These DL particulars made it possible to solve existing problems with shallow ANN architectures, which use only a few hidden layers. Among the various DL techniques found in the literature, those based on Stacked Autoencoders (SAEs), Deep Belief Networks (DBNs), Convolutional Neural Networks (CNNs), and Restricted Boltzmann Machines (RBMs) have great popularity [13], [15], [16]. DL models can perform with large amounts of data and can also be applied to a wide range of activities, including computational vision problems, audio and speech processing, natural language processing, robotics, bioinformatics, cyberattack security systems, recognition and classification applications, and finances, among others [13], [17]–[27]. DNN based on SAE, DBN and RBM techniques continues to grow, as it represents an excellent solution for many applications even with few hidden layer [26], [28]–[31]. As detailed in [26], DNNs in range of one-to-five hidden layers, can provide a great solution for cyberattack detection and classification.

The deployment of DNNs on Edge presents a challenge for engineers. In fact, high performance, flexibility, low energy consumption, and even sometimes real-time responses, are mandatory criteria to be met. Graphics Processing Units (GPUs) may actually meet these first two criteria, but they still suffer from delivering power consumption efficiency. On the contrary, Application-Specific Integrated Circuits (ASICs) can guarantee high performance with considerably low power consumption. However, ASICs are still not flexible enough to support changing and scalable topologies such as those used for Federated Learning.

Between these extremes, Field Programmable Gate Arrays (FPGAs) offer the best compromise between design flexibility and energy efficiency. ASIC development relies on FPGAs, which in addition to providing a faster time to market, in many applications offer similar benefits at lower development and manufacturing costs. In terms of energy consumption, FPGAs, optimized at the circuit level, can have better performance than GPUs [32]–[34]. However, the path to optimal implementation of DNN topologies on FPGAs remains complex, requiring expertise in several areas, DL algorithms and topologies, embedded and reconfigurable computing. Custom design can produce the best performance solutions, but it is an optimization that takes time and lacks flexibility [29], [35]. In this context, tools are available but mostly oriented towards mainframe applications, such as Intel Open Vino for Arria 10 GX [36] and Vitis-AI cards for Alveo or UltraScale available in collaborative environments such as Amazon Web Services EC2-F1 [37]. However, with the growth usage of Edge Computing, similar approaches are now emerging for embedded-FPGA. For that, leader re-configurable computing companies start releasing new AI tools for their embedded boards such as the Lattice sensAI [38] and the pre-build AI library for Kria Xilinx SOM [39]. Their differences rely on optimization alternatives for limited hardware resources. Optimizations can be provided at several abstraction levels with associated tradeoffs [34], [40], [41]. Thus, rapid results feedback is preferred due to the multiple user requirements.

For the reasons explained above, we propose an automated development framework allowing: an efficient deployment of DNN topologies on embedded FPGAs dedicated to Edge Computing; manage design complexity and tradeoffs transparently; combine custom hardware scalability with flexible optimization strategies; to meet user needs while respecting embedded system limitations; and to facilitate specification entry from Python that mimics the TensorFlow customization's way. In this work, we are currently looking at popular DNNs that can be implemented on Embedded Edge without losing performance or accuracy on the target FPGA. Our framework is dedicated to building different DNN topologies, particularly those based on the fully connected networks type such as SAE, DBN, and RBM. Our experiments were performed using the SSAE model. Our future work (see the Conclusion section) will focus on larger networks, with acceptable, performance or accuracy degradation.

In summary, the main contributions of this work are listed below:

- Managing the balance between pipelining and parallelism at the intra-layer level to optimize the performance in accordance with available resources. This technique will be used to create custom hardware IP library components that are optimized to meet performance requirements. We further provide a model to estimate the expected throughput after applying the desired optimization.
- Flexible interfacing alternatives combining stream and memory (off/on chip) to deal with latency, further improving throughput and asynchronous data exchange between layers. These techniques and their impact on the overall performance and architectural resources will be presented in the following sections.
- We propose an automated end-to-end design framework, with parameters (i.e, the balance between pipeline/Parallel optimizations and interface flexibility) allowing the user to get the best tradeoff for DNN deployment on the Edge (performance, power consumption, and size). For this, an easy to use Python library was proposed in order to specify DNN topologies with the standard parameters and to generate TCL scripts to fully automated and control the hardware implementation process. We purposely restricted the target FPGA to a small cheap SoC (Xilinx ZYNQ 7020), which is widely used for real-time and edge computing applications [42], [43].

The remainder of this paper is organized as follows. Section II explores some related cutting-edge works. Section III describes the functionality and base structure of our IP-layer. The optimization alternatives available to tune the IP layers and their impact on the modeled performance parameters are detailed in Section IV. In Section V, we develop custom layer interfacing strategies. The design framework, user entry and automatic generation of the custom hardware IP for the target FPGA, implementation is presented in Section VI. Sections VII and VIII evaluate the impact of the design parameters on the estimated and implementation results, as well as experimental results comparing similar state-of-the-art approaches. Finally, we conclude with the contributions and achievements in Section IX.

II. STATE OF ART

In the literature, many works proposing reconfigurable computing to accelerate DL algorithms exhibiting speed gains compared to implementations using CPUs and GPUs [32], [33], [44]–[47]. The latter is widely used to accelerate DNN topologies, as they offer superior results in terms of pure computational throughput, where, for example, 15 TFLOPS can be achieved by the new Tesla V100 GPU [48]. Nonetheless, a high-throughput GPU is still considered to be very inefficient in terms of the processing power / energy consumption relationship. This is why they are not popular for embedded edge computing, but their use also negatively impacts the overall energy costs in data centers [49]. This why the popularity of FPGAs is growing in both domains [44]. Indeed, FPGA-based implementations have proven to be as fast as some high-end GPUs while ensuring low power consumption [28], [29], [32]–[34], [44]–[47], [50]–[55].

Improving the parallel computing is fundamental as it will impact the network's throughput. However, the choice of a fully parallel technique as the target architecture will limit the size of the DNN that can be implemented for an embedded application. The systolic array technique [56] provides an approach between a fully parallel and serial architecture. This technique allows data to be received in a serial manner and the PEs to perform their operations in parallel [57], [58]. A systolic array is still the best choice regarding processing speed. For that, we used this technique in our work. Actually, it is used in several works in order to enhance the DNN performance [29], [58], [59].

Exploring edge computing solutions, Maria *et al.* [28] proposed DNN implementations in FPGA using SSAE to provide low power topologies for real-time object recognition in autonomous systems and robots. Their accelerators were modeled in OpenCL, a programming language used for heterogeneous parallel systems. In this work, a Stratix V D5 FPGA was used to accommodate a 3072-2000-750-10 stacked autoencoder to classify the CIFAR-10 color dataset. Their low power implementation, 357mW, although less efficient compared to CPU-GPU alternatives, reached 45 FPS. In order to optimize performance, in addition to power, another FPGA-based SSAE was proposed in [29]. Their systolic network architectures paired with custom RTL operators were 20× faster compared to [28], especially when memory resources were used to store network weights simultaneously. However, in both cases, high-performance FPGAs, Stratix V and Virtex 6, were required.

FPGA-based accelerators require a much longer development time than software solutions. They need a great deal of electronics experience and skills, especially for custom optimizations using Hardware Description Language (HDL). For this reason, in recent years, several works have focused on specializing frameworks and tools for automating DNN architecture designs for FPGAs combining custom RTL designs with high-level languages, as outlined in [45]–[47], [51], [53], [54]. Although valuable in terms of performance per watt compared to the CPU-GPU, the results obtained with these co-design approaches are still not targeting embedded applications, as high-end FPGAs, such as Arria 10 GX 1150 or Stratix V GXA7, were used in most cases. For example, the minimum power achieved by [45] was 25 W on a KU060 FPGA platform with two Intel E5-2609 processors.

The work in [51] proposes an RTL-level CNN compiler that automatically generates customized hardware for the inference phase of various CNNs. They developed a general-purpose library of RTL components, carrying out operations on each CNN layer. Library components consist of hand-coded Verilog templates, designed to minimize memory access and data movements whereas optimizing resource utilization. Their approach has been demonstrated on two standalone Intel FPGAs, Stratix V and Arria 10, with implementations of various CNN algorithms, e.g., VGG-16 and ResNet-152. They achieve 2× superior performance compared to state-of-the-art automation-related works. DNNBuilder [47] is a tool for creating high-performance DNN hardware accelerators on FPGA. DNNBuilder implements a fine-grained pipeline structure, a caching scheme between pipeline stages, and highly optimized RTL network layers with arbitrary quantizations. In order to ensure efficient resource usage, an automated exploration of parallelism optimization guidelines was provided. Results obtained with DNNBuilder are up to 4.35× more efficient than the GPU-based solutions, achieving a throughput performance of 4218 GOPS, which outperforms FPGA-based solutions. In [60], the authors presented 'HybridDNN', an environment that includes a flexible and scalable architecture, a design space exploration tool, and a design flow for the implementation. On a high-end FPGA (VU9P) and an embedded FPGA (PYNQ-Z1), experimental results demonstrate that the accelerators built by HybridDNN can yield 3375.7 and 83.3 GOPS, respectively. A high-level design automation framework to optimize the mapping of regular and irregular CNNs on FPGAs was proposed in [46]. Based on Synchronous Data Flow (SDF), their automated design methodology enables the efficient exploration of architectural alternatives. Designs using this framework achieved 6.65× faster than highly optimized GPUs within the same power budget and 2.94× higher performance compared to cutting edge CNN FPGA-based implementations. Mousouliotis and Petrou *et al.* [61] propose an automated framework to map a CNN on low-cost FPGA ZYNQ. They developed templates guides the user to create, verify, and converting a part of the algorithm into an HLS directives. However, the user must code the main file in C/C++ as well as the HLS pragmas for FPGA accelerator part.

These design environments [47], [51], [60], [61] are good examples of the different techniques that can be used to automate and optimize the implementation of hardware accelerators, but it still challenging to be managed by non-specialists looking for embedded solutions.

Inspired by TensorFlow, [54] also introduces a platform that automatically generates customized FPGA-based hardware accelerators for CNN models. The proposed platform allows the user to choose the data set and customize CNN models using a Graphical User Interface (GUI). In this work, five CNN models were created using Tensorflow and compared to each other. CNN models were trained using MNIST, CIFAR-10, and STL-10 datasets. The classic LeNet-5 architecture results show a latency/frame of 1.08ms and 0.58ms for 32-bit and 16-bit architectures respectively. The high-level design framework FP-DNN [53] enables TensorFlow DNN specifications to be mapped to FPGAs using HLS-RTL hybrid templates (RTL components written in Verilog and HLS in OpenCL). FP-DNN can also perform the DNN inference process. When it comes to energy efficiency, their results are better than those of the CPU for all models. With 16-bit fixed-point precision data types, FP-DNN implementations are 1.9× to 3.06× faster than CPUs but can only compete with GPUs with lower precision. In [45], a hardware/software co-design library, called Caffeine, has been proposed to facilitate the design of energy-efficient CNN acceleration on FPGAs. This approach, which uses High-Level Synthesis (HLS), is integrated into the Caffe DL framework. The results show that Caffeine can achieve a peak performance of 365 GOPS on the Xilinx KU060 FPGA and 636 GOPS on the Virtex7 690t FPGA, delivering 7.3× and 43.5× performance and power savings compared to Caffe on a 12-core Xeon server and 1.5× improved energy efficiency compared to a GPU. These design environments provide end-to-end DNN implementations facilitating hardware acceleration for non-specialists, and some providing impressive performance results. However, most are HPC oriented, thus do not take into account the restrictions of edge computing.

In this work, we combine the previous techniques to provide an end-to-end tool to automate the development of optimized DNNs for low-power embedded platforms for advanced applications. Thus, user-driven optimization methods, especially pipelining, parallel processing, and systolic array, as discussed in [28], [29]. This flexibility design method does not cover only the processing elements as detailed in [62], but also the interfacing of each customized IPs layer, in order to meet performance and the used board requirements. Data access stream and memory usage (off and on-chip) interfaces are provided to improve the throughput, and asynchronous the communications to reduce latency. As will be demonstrated later, these techniques are still carefully combined to limit their impact on energy consumption. To facilitate adoption, we provided a Python library to define the DNN at the software control layer-level that mimics the TensorFlow methods. Standard design parameters, such as the target platform and other configuration criteria, can be transparently overridden. Comparing to some works, our high-end python interface helps to overcome the barrier that is struggling non-experts developers, especially those presented in [47], [51], [60], [61]. To speed up hardware implementation and facilitate design exploration, high-level design tools (Xilinx Vivado) are driven by TCL scripts generated from the Python-based DNN configuration. In this way, the user can obtain rapid feedback and compare the results before the deployment of the created IP on the FPGA.

III. THE OVERVIEW OF THE PROPOSED HARDWARE ARCHITECTURE

In this Section, we describe the library components that were created to support the proposed automated design framework for the development of FPGA-based DNN architectures. As in other approaches, we developed a library of hardware components. These components allow the implementation of any DNN topology and can be configured to enhance performance under embedded platform constraints. Indeed, layers and operators can be tailored to requirements such as FPGA available resources, acceleration rate, or memory.

A. THE DEEP NEURAL NETWORK TOPOLOGY

A basic neural network is built by hooking together many neurons so that one neuron's output can be another input. Neurons and connections can be structured in layers with various forms and characteristics. Fig. 1 shows a typical DNN topology and the associated parameters used as a reference to understand the organization and structure of the proposed library.

The organization of the layers helps us to identify three types of high-level components, the IP-layers. At the input layer, the first type, the incoming data is directly connected to the input neurons, without any modification or transfer/activation function (AF). The second type is the output layer, which is characterized by the Softmax function. This function, which returns the probability of a data item belonging to an existing class, can be used for multiclass classification problems. The third type is the hidden layer, which can have the same input/output and neurons format, although its size (the number of neurons), bias, and AF vary from layer to layer. Non-linear AFs are the most commonly used. For this reason, we implemented the Sigmoid, which provides a smooth gradient analog activation. In addition, between -2 and 2 input values in the X-axis, output values in the Y-axis can be very steep, which means that minor changes in that region would cause significant changes in the output values.

B. EQUIVALENT MATHEMATICAL REPRESENTATIONS

Hardware customization and optimization parameters are defined in order to drive the implementation process. To facilitate their identification, we provide the mathematical representation characterizing the network layers in this Section, which follows labels and annotations shown in Fig. 1. We denote k to indicate the number of the hidden layers in the network; therefore, the first layer's label will be L_1, and L_k for the last hidden layer. We denote W_{ij}^{Ln} to specify the network weights between two successive layers in which i and j represent the neurons respectively on the layer $L_{(n-1)}$ and L_n. We denote b^{Ln} the bias for the $L_n - th$ layer's neurons and Wb_j^{ln} the weight bias for $j - th$ neuron.

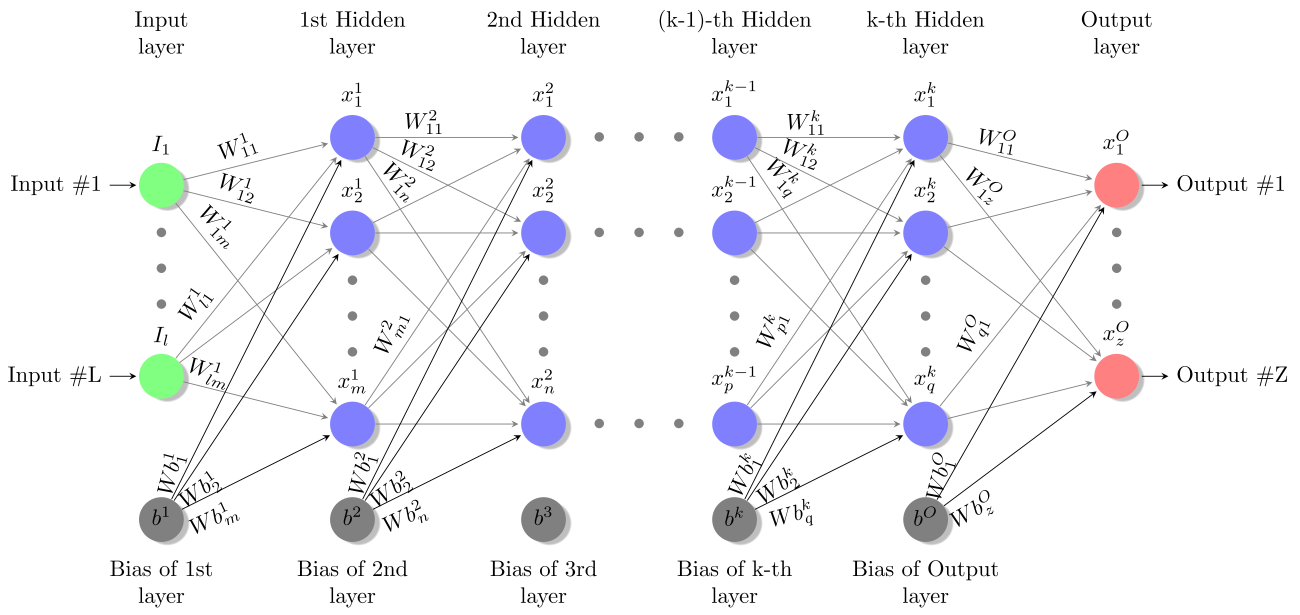

FIGURE 1. **A typical DNN topology. In this graph, circles represent network neurons: hidden layers (blue), and input (green) / output (red) layers. The gray circles represent the bias nodes. The leftmost layer of the network is called the input layer. The output layer is on the far right side. The hidden layers, considered as the core layers, are represented in the middle of the network between the input and output layers.**

We can describe the equation of the output result for the $j-th$ neuron at L_n-th layer as follows

$$\begin{aligned} x_j^{Ln} &= f(\sum_{i=1}^{Nx^{Ln-1}} (x_i^{Ln-1} * W_{ij}^{Ln}) + Wb_j^{ln} * b^{Ln}). \\ &= f(S_j^{Ln} + B_j^{Ln}). \end{aligned} \tag{1}$$

where x_i^{Ln-1} is the output result of $i-th$ neuron at the $L_{n-1}-th$ layer, S_j^{Ln} is the sum of weighted outputs of the $L_{n-1}-th$ layer connected to $i-th$ neuron at L_n-th layer, the variable $Nx_{L_{n-1}}$ is the number of neurons in $L_{n-1}-th$ layer, B_j^{Ln} is the weighted bias for $j-th$ neuron, and $f(*)$ is the AF.

The sigmoid AF of this output is represented as follow

$$Sig_j^{Ln}(S_j^{Ln} + B_j^{Ln}) = \frac{1}{1 + \exp(S_j^{Ln} + B_j^{Ln})}. \tag{2}$$

This equation is not considered in the input layer because it does not use AF. The same thing for the output layer; in that case, a softmax function is often used instead of the previous AF. The Softmax function is expressed as follows

$$Soft_j(S_j^O + B_j^O) = \frac{\exp(S_j^O + B_j^O)}{\sum_{n=1}^{Z} \exp(S_n^O + B_n^O)}. \tag{3}$$

where S_j^O is the sum of weighted outputs of the $k-th$ layer connected to $i-th$ neuron in the output layer, B_j^O is the weighted bias for $j-th$ neuron at the output layer, and Z represents the number of outputs, in another term, the number of classes.

These equations will be used to determine the set of basic components used to build the IPs-layer (see Fig. 2 and Fig. 3).

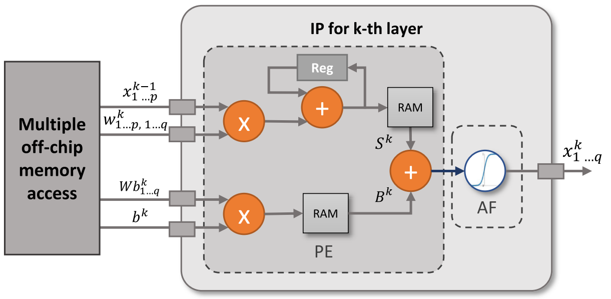

FIGURE 2. **Basic IP structure. The IP includes a Processing Element PE to compute S^k and B^k values and the Activation Function AF operator. The PE is considered as a sequential series of accumulated multiplications of x^{k-1} by its associated weight W^k plus the multiplication of b^k by its associated weight Wb^k.**

C. THE BASIC IP-LAYERS

We provide a library with basic IP hardware components that will be tailored, first according to the desired DNN topology. The Interfaces will then customized to create hidden and output layers. Each IP includes a basic Processing Element (PE) to compute the S^{Ln} and B^{Ln} values; and the AF operator. The PE of a $Ln-th$ layer is considered a sequential series of accumulated multiplications of x^{Ln-1} by its associated weight and weighted bias. The number of iterations required to compute the output results of the $Ln-th$ layer is equal to the number of its neurons. Fig. 2 depicts the IP for the k_th layer and the basic structure of the PE ones tailored. Pseudo-code 1 of our C/C++ IP template shows the iteration steps to compute the output of a layer of $k-th$ layer. The latter is divided into three iteration loops, the first of which is used to multiply the bias by its weight, as shown in figure 1. The second loop uses DNN weights to multiply and accumulate the input data, while the third is used to calculate layer outputs (i.e, activation function for each output neuron). This organization was developed with the aim of providing layer implementation flexibility and complete control over the RTL IP version. In order to get a hardware architecture for such DNN, the framework will modify the template according each customized parameter layer. Then, the framework will create a main C++ that calls the modified templates as C++ functions in order to build the DNN topology. After that, the framework will automate the generation process based on these new C++ files. More details are presented in section VI.

Pseudo-code 1: IP of the th k-th Layer. Once All k-1-th Layer Outputs Are Weighed for Neurons on k-th Layer and Accumulated, the AF Will Be Executed

```
for $j \leftarrow 1$ to $Q$ do
    $B[j] = Wb[j] * b$
end
for $i \leftarrow 1$ to $P$ do
    for $j \leftarrow 1$ to $Q$ do
        $S[j] += X(k-1-th)[i] * W[j]$
    end
end
for $j \leftarrow 1$ to $Q$ do
    $S[j] += B[j]$
    $X(k-th)[j] = AF(S[j])$
end
```

Configuration parameters are provided to explore implementation alternatives evaluated through estimation results. We will notice that some configuration parameters will impact the connectivity type as well as the resources used. Section V will detail these connectivity protocols as well as how they will be used to interconnect the IPs in order to implement such DNN. Section VI will detail how IP layers, driven by user parameters and target platform constraints, will be implemented automatically. This includes the encapsulation (wrappers) and a user application interface. As will be detailed later, each IP is also represented as a C++ library function to validate its behavior through a simulation from the software point of view.

IV. PROCESSING ELEMENTS OPTIMIZATION

The PEs are executed sequentially through iterations, and the intermediate results are stored on internal RAM, possibly being single-port. Although this operation mode needs fewer resources, it can be a bottleneck depending on data rate requirements. Thus, pipeline and parallelization techniques can be applied but driven by data rate and available resources. Indeed, individual PE operations (multiplication and addition) can be parallelized according to a constant factor, impacting the size of memories (BRAM, registers) and the number of operators (multiplier, accumulator) that will be used simultaneously.

A. DATA DISTRIBUTION FOR PARALLEL OPERATIONS

Design parameters and optimizations are driven by requirements and restrictions, such as data rate and size. Once determined, these values would impact the control-path (in clock cycles) as well as the resources of the data-path. Indeed, in a PE, computing operations (OPs) (multiplication and accumulation) are critical components that directly affect the processing speed and final throughput. Regarding data storage, intermediate results can be distributed over individual RAM blocks to maximize simultaneous read/write. This distribution is controlled by a constant factor α. According to this factor, OPs within PEs can be parallelized. Thus, bias and input data loop iterations can, in addition to being pipelined, be parallelized.

A single operation OP is carried out in three steps: reading data, execution, and writing results. The number of clock cycles for each step depends on the FPGA technology, the type and location of RAMs, and the hardware resources used (BRAM, DSPs, FF, etc). The parameter OP_{cc} defines the number of clock cycles needed for a single OP. After applying α factor, the total number of clock cycles - tOP to compute all OPs iterations for a such neuron would, therefore, be as follows

$$tOP(x_i^{Ln-1}) = Ceil(\frac{Nx^{Ln}}{\alpha}) * OP_{cc}. \quad (4)$$

where x_i^{Ln-1} is the $i-th$ neuron in the $L_{n-1}-th$ layer, and Nx^{Ln} is the number of neurons in $L_n - th$ layer. $Ceil(Nx^{Ln}/\alpha)$ returns the smallest integer value greater than or equal to Nx^{Ln}/α. This value reflects the number of iterations required to complete the processing of one operation for the $i-th$ neuron in $L_{n-1}-th$ layer after applying α factor. As shown in Fig. 2, S^{Ln} is composed of one multiplication OP(MUL) followed by an accumulation OP(ACC). Therefore, the delay of S^{Ln} for x_i^{Ln-1} will be computed as follows

$$tS^{Ln}(x_i^{Ln-1}) = Ceil(\frac{Nx^{Ln}}{\alpha}) * (OP_{cc}(MUL) + OP_{cc}(ACC)). \quad (5)$$

where $tS^{Ln}(x_i^{Ln-1})$ is the total number of clock cycles to compute all S operations in $L_n - th$ layer for the $i-th$ neuron in $L_{n-1}-th$ layer, $OP_{cc}(MUL)$ is the number of clock cycles for a multiplication operation, and $OP_{cc}(ACC)$ is the number of clock cycles for the accumulation operation.

Likewise, the bias B computation is considered as an additional neuron in the $L_{n-1}-th$ layer. Since it uses only a multiplication operation, the delay of the bias operation becomes the following

$$tB^{Ln} = Ceil(\frac{Nx^{Ln}}{\alpha}) * OP_{cc}(MUL). \quad (6)$$

where tB^{Ln} is the total number of clock cycles needed to compute B operations for $L_n - th$ layer.

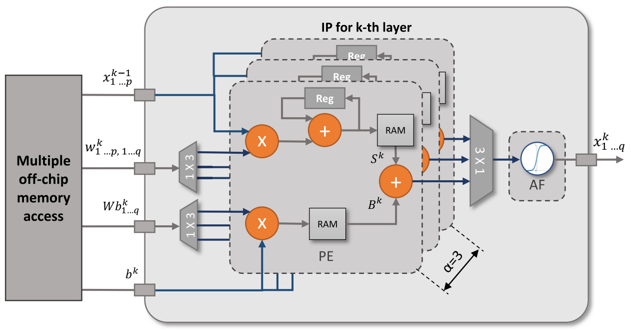

FIGURE 3. **The basic IP-layer after applying $\alpha = 3$ factor.**

Fig. 4 shows the processing schedule before and after parallelization, according to α factor. In the example, we use the $k-th$ layer of Fig. 1 to describe the execution order. Initially (serial $\alpha = 1$), the OPs for x_1^{k-1} and b^k are performed sequentially. Assuming $\alpha = 3$, the layer begins by computing in sets of 3 simultaneous operations. The scheduler shows the execution of 6 OPs for $S^k(x_1^{k-1})$ and B^k in 2 steps instead of 6. The rest of the figure includes the pipeline execution, which will be described in the next Section.

Fig. 3 shows the basic PE instantiated by a factor $\alpha = 3$. The layer will therefore receive sets of α weights W_{ij}^k and Wb_j^k for x_i^{k-1} and b^k, respectively, distributed to the inputs of the multipliers as well as a single input x_i^{k-1} and b^k. The output of each PE is sent to the AF.

The fully parallel configuration, determined by α_{max}, when it is equal to Nx^{Ln}, implies that all operations of x_i^{k-1} (tS and tB operations) will be performed in one OP_{cc}. However, this requires α_{max} PEs and block RAMs to be able to perform all of these operations simultaneously.

B. PIPELINE STRATEGY

In the event that built-in DSPs can be used, pipelined processing can be performed using sets of α OPs. In this case, PEs start receiving a new set during the processing of the previous one. We denote the Iteration Interval - II as the minimum time interval between two successive sets. We compute this value as the number of clock cycles. With a low II, the transfer rate is higher. When this value is equal to 1, it is possible to process a new set at each clock cycle. However, this value depends on the technology, available operators, and memory access. In order to pipeline the $S^{Ln}(x_i^{Ln-1})$, we need to accumulate the multiplication result of the actual input data x_i^{Ln-1} by its weight with the previous $S^{Ln}(x_{i-1}^{Ln-1})$ result. For that, the multiplication OP(MUL) and the accumulation OP(ACC) should be executed sequentially. Incoming values are still processed in groups of α PEs. The pipeline optimization will be, therefore, applied between each α set of these two operations (MUL+ACC).

Fig. 4 shows the scheduling of operations in groups of $\alpha = 3$, with an $II = 4$ as the initialization interval for each group of values, split in two as each input value x_2^{k-1} must be processed 6 times. The same procedure is applied for each α set bias multiplications, where $II = 1$.

With pipeline and parallel execution, the new total clock cycles number $t'OP(x_i^{Ln-1})$ for a specific operation of x_i^{Ln-1} will be calculated as follows

$$t'OP(x_i^{Ln-1}) = OP_{cc} + II * Ceil(\frac{Nx^{Ln}}{\alpha} - 1). \tag{7}$$

After applying these optimizations to parts S and B, the delay of $t'S^{Ln}$ for x_i^{Ln-1} (see (5)) and the delay of the bias $t'B^{Ln}$ (see (6)) will become the following

$$\begin{aligned} t'S^{Ln}(x_i^{Ln-1}) &= OP_{cc}(MUL) + OP_{cc}(ACC) \\ &\quad + II * Ceil(\frac{Nx^{Ln}}{\alpha} - 1). \end{aligned} \tag{8}$$

$$t'B^{Ln} = OP_{cc}(MUL) + II * Ceil(\frac{Nx^{Ln}}{\alpha} - 1). \tag{9}$$

As Fig. 3 shows, in our IP we do not adopt parallel processing for the AF to avoid complexity implementation in the final version of the IP. For this, we applied only the pipeline optimization, therefore, the total clock cycles of AF^{Ln} will become as follows

$$t'AF^{Ln} = OP_{cc}(Sig) + II * Nx^{Ln-1}. \tag{10}$$

where $OP_{cc}(Sig)$ is number of clock cycles of Sigmoid operation.

From (7), we conclude that $t'OP$ will decrease as a function of α, but is still restricted by the technology-dependent factor II.

Considering the example shown in Fig. 4, where we assume that an $OP_{cc}(MUL)$ requires 10 clock cycles, $II = 1$ and $\alpha = 3$, after optimizations $t'B^k$ becomes 11 clock cycles against the original 60 clock cycles. However, this will affect the number of operators and the block RAM size, which will be multiplied by α. Likewise, the improvement in $t'S^k$ will impact hardware resources, as shown in Fig. 3.

Fig. 5.a shows how $t'OP(MUL)$ will be improved according to the factor α from 1 (no parallel processing) to α_{max} (fully parallel) with Nx^{Ln} equal to 50 neurons. Observe how the pipeline alternative behaves for some II values. Fig. 5.b represents the percentage of speed optimization as a function of α where 100% represents a fully parallel multiplication operation (α_{max}). The optimization percentage is expressed as follow

$$Perc(\%) = \frac{OPcc}{t'OP} * 100. \tag{11}$$

Through Fig. 5 we can deduce that the variation of α value from 1 to 5 changes the optimization percentage significantly for all curves. In contrast, the change is smoother between 5 and 10, and almost imperceptible beyond 10. Parallelism and pipeline are both affected by II, keeping this value as small as possible allows us to improve speed without causing a significant impact on size. For instance, with $II = 1$ we can already obtain almost maximal speed at $\alpha = 2$. Therefore, we recommend to use these values on each new

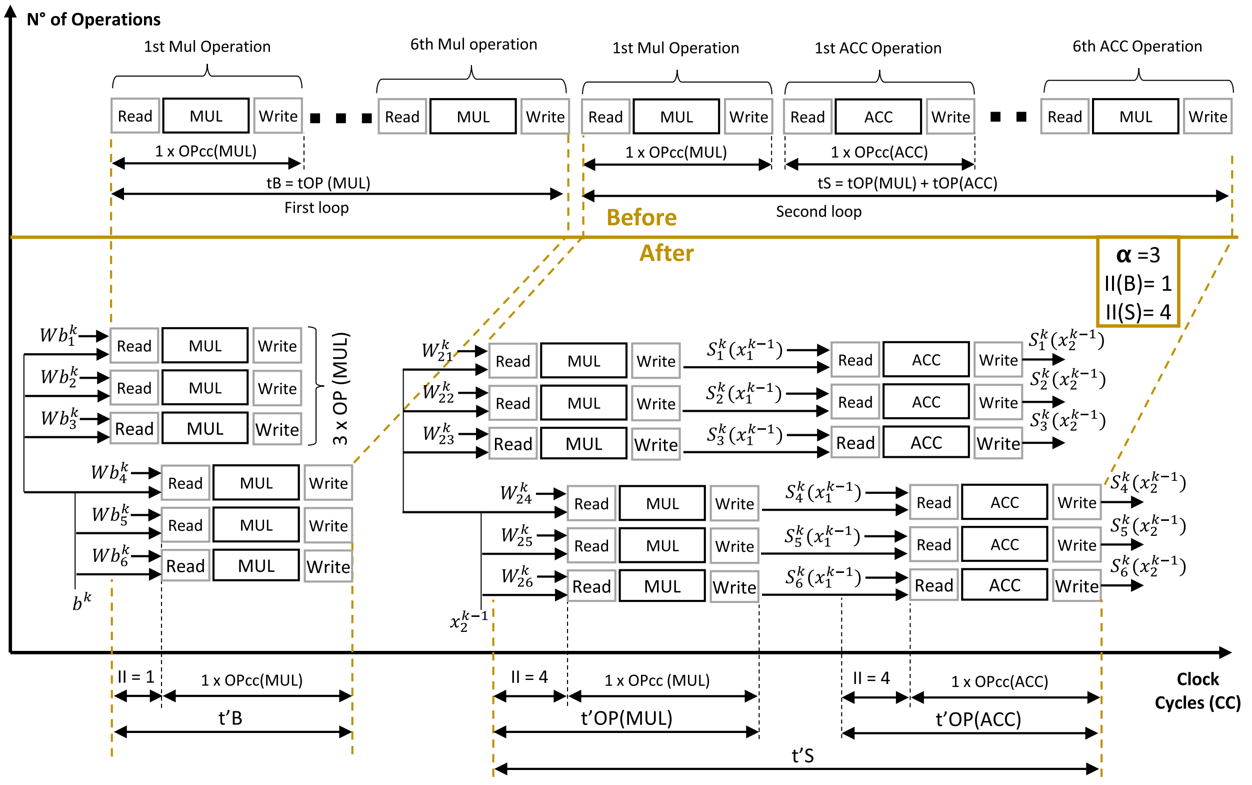

FIGURE 4. Scheduler of processing strategy of first 6 neurons before and after applying the parallelization factor $\alpha = 3$ and pipeline optimizations ($II(B) = 1$ and $II(S) = 4$).

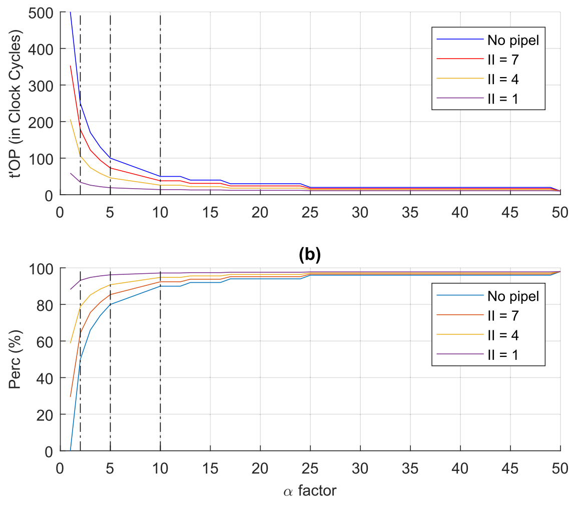

FIGURE 5. The estimation of $t'OP(MUL)$ and the percentage of acceleration $Perc(\%)$ as a function of α value (from 1 to $\alpha_{max} = 50$) for a $Ln - th$ layer with $Nx^{Ln} = 50$ neurons.

DNN implementation for the best trade-off between speed and hardware consumption.

V. INTERFACING AND COMMUNICATION PROTOCOLS

In this work we explore the alternatives for a systolic array organization for layers. In this approach, library IPs can be chosen and interconnected to build multiple DNN topologies. With limited storage resources, the first alternative is to gather data and intermediate results onto external memory accessed via interconnection busses. The second alternative, FIFO busses for data exchange between layers can be done for fast direct interconnection interfaces.

As we target embedded SoC, the system uses the AXI (Advanced eXtensible Interface) protocol (part of the ARM Advanced Microcontroller Bus Architecture (AMBA) specification [63], [64]). AXI standards are widely adopted as on-chip communication protocol, providing a universal IP reuse interface. Thus, to facilitate scalability and compatibility between IPs from different vendors, IP layers are encapsulated with standard AXI interfaces. Fig. 6 shows an example of a simple 3-layer DNN, in which the first layer has full external communication, while the others use a mixed-mode communication type. At the system level, the application running on an embedded CPU uses the Memory Mapped AXI (AXI-MM) interfaces for data exchange and AXI-Lite for control. Inter-layers data exchange uses AXI-Stream.

Each AXI interface has resources suitable for a specific type of communication. The AXI-MM, as well as the AXI4-Lite interface provides five communications channels for address, data, and control driven by the master interface. The control-oriented AXI-Lite allows one data transfer per transaction, whereas the data-oriented AXI-MM allows a burst transaction of up to 256 data transfers. The AXI-Stream, designed for high-speed streaming data and high

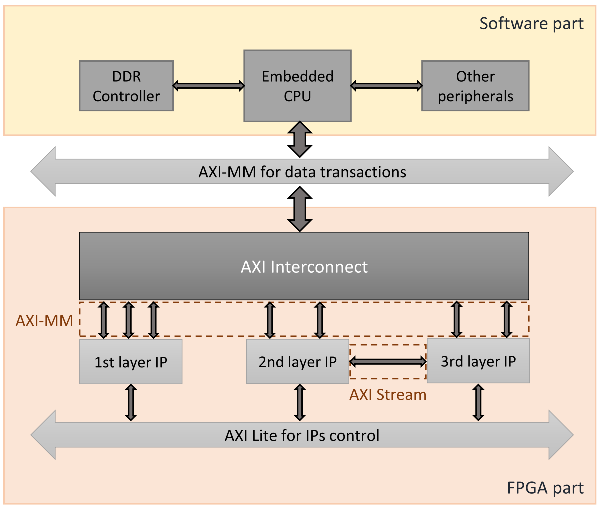

FIGURE 6. System level interconnection view of a software application and associated hardware accelerator. The application running on an embedded CPU uses the AXI-MM interconnection for data exchange and AXI-Lite for control. Inter-layers data exchange uses AXI-Stream.

bandwidth unidirectional non-deterministic burst (undefined length) data transfers, provides a single write channel. In the next Section, we will explore alternative implementations using these interfaces.

Fig. 6 shows the three types of IP layers available according to the interfaces used. The IP layer uses an AXI-Lite interface by default to activate (ap_start), control (ap_idle, ap_ready and ap_done status signals), and synchronizes data transfer (weight, bias, or I/O data). For data exchange, the layers use separate interfaces whose control and behavior depend on the type of layer being implemented. The first type, named full AXI-MM IO, uses an AXI-MM slave interface for input data and an AXI-MM master for output data. The second type, named mixed MM-Stream, uses AXI-MM interfaces for input data and AXI-Stream for output data.

Using AXI-MM, the data are stored and loaded from/to external memory. Four input data ports are created for x_i^{Ln-1}, b^{Ln}, W_{ij}^{Ln}, and Wb_j^{Ln}, as well as one output data port for x_i^{Ln}. AXI-MM can be used on any layer to exchange data between layers with the user application (built-in CPU). AXI-Lite controls the displacement memory of each AXI-MM. Fig. 7 shows an IP with a full AXI4-MM interface protocol for I/O ports and AXI-Lite for IP control.

Data exchange between layers can also be directly performed using AXI-Stream. In this case, there are no off-chip memory transfers for intermediate results and, eventually, improved throughput. However, FIFOs are required to adjust data rates. Fig. 8 shows an example with AXI-MM (W_{ij}^{Ln}, b^{Ln}, and Wb_j^{Ln}) and AXI-Stream (x_i^{Ln-1}) slaves for the input data, and AXI-Stream master for the output results (x_i^{Ln}).

VI. AUTOMATION OF THE HARDWARE IPs-LAYER CONFIGURATION AND FPGA IMPLEMENTATION

For the IP-layer encapsulation and synthesis, we use Xilinx's Vivado High-Level Synthesis HLS tool, facilitating the use

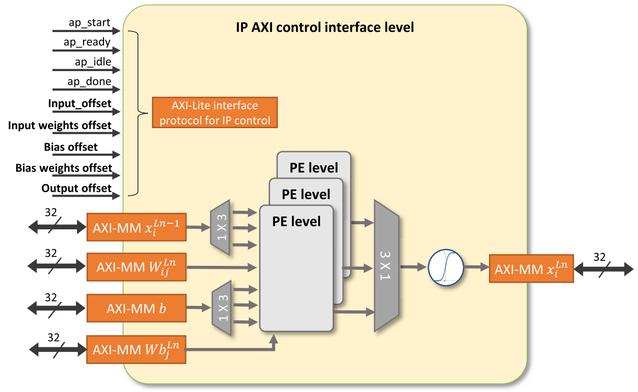

FIGURE 7. Full AXI4-MM interface: using AXI-MM slaves for inputs and master for output. AXI-Lite controls the IP and input data offsets.

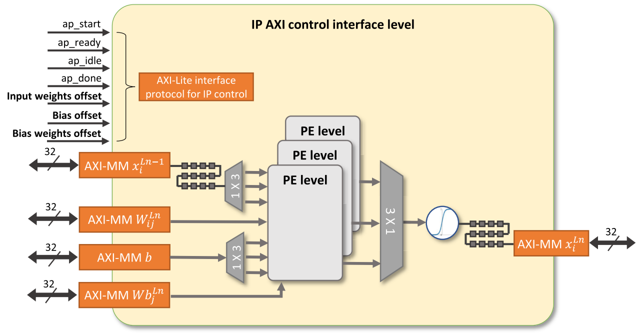

FIGURE 8. Mixed MM-Stream interface: using AXI-MM and AXI Stream slaves for inputs and AXI-Stream master for outputs. AXI-Lite controls the IP and input data offsets.

of standard AXI interfaces when targeting the ZYNQ SoC architecture. The complete design process, from configuring the IP layer to generating the bitstream, is driven by the use of TCL scripts.

Fig. 9 shows the design flow, starting from a Python application (in yellow). Indeed, this work focuses on generating a hardware accelerator from a DNN whose parameters and topology have already been defined, therefore, an equivalent C++ main function can be created for debugging and design exploration. Likewise, topology, optimizations, and interfaces can be used to configure and characterize each C++ IP-layer template. For this reason, we can split the design process (in gray) into two flows, C++ main for topology and system integration, and C++ layer template to configure, layers interfacing, and encapsulate each chosen layer. Note that, at the end of this second flow, the IP-layer can be exported to an IP library with, already available, custom/RTL/HLS layers.

Both design flows begin by using a Python library that extends TensorFlow-like functions for hardware implementation, as shown in pseudo-code 2, for a network with two hidden layers and an output layer. Hardware dependent configurations, such as target board or platform selection, as well as AXI interfaces, α, and II parameters, are hidden with

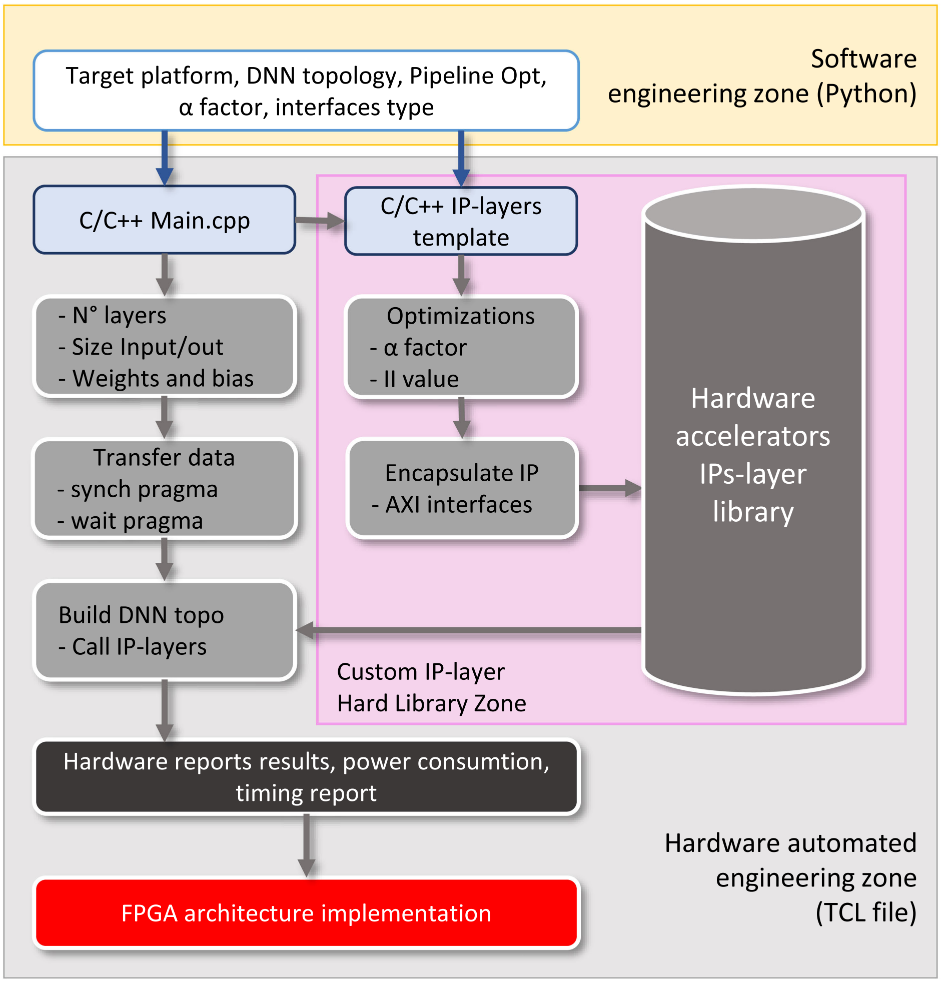
Target platform, DNN topology, Pipeline Opt, α factor, interfaces type
Software engineering zone (Python)
C/C++ Main.cpp
C/C++ IP-layers template
- N° layers
- Size Input/out
- Weights and bias
Optimizations
- α factor
- II value
Hardware accelerators IPs-layer library
Transfer data
- synch pragma
- wait pragma
Encapsulate IP
- AXI interfaces
Build DNN topo
- Call IP-layers
Custom IP-layer Hard Library Zone
Hardware reports results, power consumtion, timing report
Hardware automated engineering zone (TCL file)
FPGA architecture implementation

FIGURE 9. DNN hardware automate FPGA framework.

default values that can be overridden by the user for design exploration or platform migration purposes. These Python functions are then automatically rewritten in C++ in the form of calls for hardware execution and data exchange as well as directives (pragmas) driving the execution order or layers (see pseudo-code 3). Each function call is associated with an IP-layer.

The second flow, dedicated to the creation of each IP-layer, uses a C++ layer template that will be configured during the execution of the TCL script. Pseudo-code 4 depicts the template and pragmas used as the hardware synthesis directives. In addition to the inputs/outputs ports and their interface configurations, in this template, classic HLS directives, loop unrolling and pipeline, used to serialize, parallelize and pipeline operations, are enhanced with the optimization parameters α and II.

The α factor is applied to split the S and B operations results data into BRAM blocks distributed on each PE equally instantiated α times (see Fig. 3). The *unroll* pragma, is then used to partially unroll loops creating groups of α operations that can be executed simultaneously. The first loop uses bias and associated weights; then only the MUL operator is instantiated α times. The second loop uses inputs and weights, thus, the ACC operator is also instantiated. The *pipeline* pragma can then create a pipelined execution of the MUL, ACC and AF operators. However, pipeline optimization uses $II = N$, where N is equal to the initiation interval (see Section IV-B). The appropriate N value is extracted during the configuration based on the technology and platform chosen by the user. Once the parameters are configured, the script will continue with the synthesis process, which will provide performance results for each IP layer implementation. Still, the optimum factor must be extracted according to the size, types and topology to be adjusted to the target platform. Pragmas specifying the AXI ports to be used for data exchange between the CPU, memory and hardware accelerators are instantiated and used during the IP encapsulation step. After encapsulation, the IP-layer is ready to be instantiated as a library IP component.

The C++ main includes the directives to instantiate the IP-layers, already available in the library encapsulated with AXI interfaces, creating the desired topology ready for use as a hardware accelerator. Pseudo-code 3 shows the pragmas to synchronize data transfers between the CPU and hardware

Pseudo-code 2: The Python API Functions to Build the DNN

```
Setup_Board(board_name,..)

outData layer1 = First_hidden_layer (inputData
  layer1, weight 1, bias 1, Nx)
outData layer2 = Secon_hidden_layer(outData layer1,
  weight 2, bias 2, Nx)
outData layer3 = Output_layer(outData layer2, weight
  3 bias 3, Nx)

Configure layer(layer=1, α=2, data
protocol=”AXI-MM”, weights protocol=”AXI-MM”)

Configure layer(layer=2, α=4, pipeline=True, data
protocol=”AXI Stream”, weights
protocol=”AXI-MM”)

Configure layer(layer=3, data protocol = ”AXI
Stream”, weights protocol =”AXI-MM”)
```

Pseudo-code 3: Call IP-Layer as C++ Functions and Pragma for Asynchronous Data Transfer on the main.cpp

```
#pragma for async IP-layer 1
First_hidden_layer( bias1, weights, inputData layer1,
outData layer1)

#pragma for async IP-layer 2
Second_hidden_layer( bias2, weighs, outData layer1,
outData layer2)

#pragma for async IP-layer 3
Output_layer( bias3, weights, outData layer2, outData
layer3)

#pragma for wait the function 1
#pragma for wait the function 2
#pragma for wait the function 3
```

Pseudo-code 4: IP With HLS Pragmas

```
#pragma AXI-lite for control
#pragma AXI-Stream for X(k-1-th)
#pragma AXI-MM for W(k-th)
#pragma AXI-Stream for X(k-th)
#pragma AXI-MM for b(k-th)
#pragma AXI-MM for Wb(k-th)
for $i \leftarrow 0$ to $Q$ do
    #pragma for partition $B$ memory by factor=$\alpha$
    #pragma for multiply MUL by factor=$\alpha$
    #pragma for unrolling by factor=$\alpha$
    #pragma for pipeline $II = N$
    $B[i] = Wb[i] * b$
end
for $i \leftarrow 0$ to $P$ do
    for $j \leftarrow 0$ to $Q$ do
        #pragma for partition $S$ memory by factor=$\alpha$
        #pragma for multiply MUL by factor=$\alpha$
        #pragma for multiply ACC by factor=$\alpha$
        #pragma for pipeline $II = N$
        #pragma for unrolling by factor=$\alpha$
        $S[j] += X(k-1-th)[i] * W[j]$
    end
end
for $i \leftarrow 0$ to $Q$ do
    #pragma for pipeline $II = N$
    $S[i] += B[i]$
    $X(k-th)[i] = AF(S[i])$
end
```

IPs-layer accelerators. The pragmas *async* and *wait* define the execution synchronization and data exchange between IP-layers, CPU, and off-chip DDR. Each layer starts processing as soon as the previous one provides the results. To reduce latency, weights and biases are made available in advance during the current execution of the layer. For this purpose, the pragma *async* generates a barrier-free stub function for data transfer and the pragma *wait* is used to properly synchronize the IPs at specific execution steps. This information is used by the Xilinx Vivado SDSoC tool for the final architecture implementation, including CPU software drivers and IP-layer busses interconnections.

The methodology described offers several facilities to software developers: to start from a high-level specification of the DNN topology in Python, to configure target implementation parameters, including IP-layer optimization and interfaces as well as multi-layer execution scheduling. By using high-level configuration parameters, TCL scripts, templates, and library components, this automated design flow allows fast exploration of design alternatives using system and high-level EDA tools transparently.

VII. SYNTHESIS RESULTS

In section IV and section VI we proposed a theoretical optimization model to improve performances of the basic IP-layer as well as a custom parameters that will drive the automatic configuration and generation of each IP-layer. In this section, we will detail the synthesis results in order to verify the accuracy of those expected from the model.

For that, we will first validate the equation (7), proposed in section IV to estimate the throughput of the IP-layer generated by the framework, as well as the impact of the α factor on hardware resources. For this evaluation, synthesis and simulation results of the set of automatically generated IP layers were used, is confirmed by tests on the target FPGA. We take $t'B^{Ln}$ as an example for the validation. Additionally, we present the total clock cycles of each processing part of

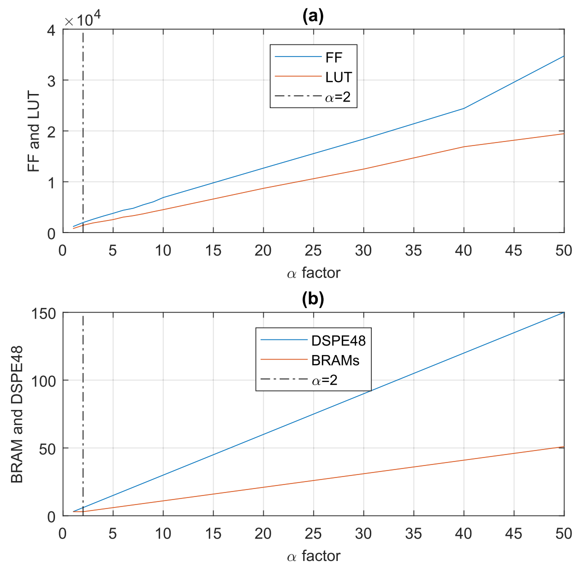

FIGURE 10. Hardware resources of PEs as a function of α factor: (a) represent the FF (blue curve) & LUT (red curve) and (b) represent the DSP48E (blue curve) & BRAM_18k (red curve) used.

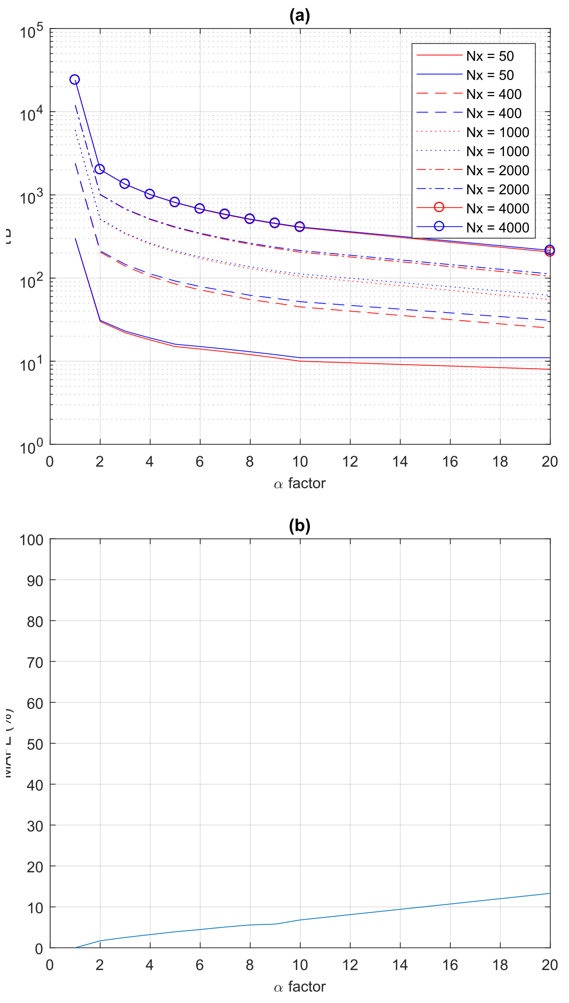

FIGURE 11. (a). Estimated (red curves) vs synthesis (blue curves) results of t'B with different IP size (Nx^{Ln} = 50, 400, 1000, 2000, *and* 4000). (b.) The Mean Absolute Percentage Error (MAPE) between estimated and synthesis results for the different IP size as function of α factor.

the IP-layer (S^{Ln}, B^{Ln}, and AF^{Ln}) before and after applying the optimization.

In the second part, we evaluate the throughput of the whole IP as well as the hardware resources according to the dimensions of the IP, before and after the application of optimizations proposed in section VI. Synthesis results were provided by the Vivado HLS tool.

A. OPTIMIZATION AND SYNTHESIS OF tB, tS, AND tAF

Fig. 10 shows the evaluations hardware resources of PE as a function of the α value. Fig. 10.a represents the resources needed, the blue curve in terms of Flip-Flops (FF), and the red curve for look-up tables (LUT). In Fig. 10.b, the DSP48E and 18k-bit RAMs (built-in blocks BRAM_18k) are shown by blue and red curves, respectively.

Fig. 11 shows the difference between the estimated results of $t'OP$ based on (7) and the synthesis results on a logarithmic scale. Fig. 11.a presents the evaluation for different IP sizes from Nx^{Ln} equal 50 to 4000. Since the differences are too small, we used a log Y-axis to observe the negligible differences for the selected set of IP sizes Nx^{Ln}. Fig. 11.b indicates the Mean Absolute Percentage Error - MAPE between estimated and synthesis results calculated by the equation below

$$MAPE(\alpha) = \frac{1}{n} * \sum_{t=1}^{n}(\left|\frac{V_{sy}(t) - V_{es}(t)}{V_{sy}(t)}\right|) * 100 \qquad (12)$$

where n is the number of curves, V_{sy} and V_{es} are the synthesis and the estimated values, respectively.

The previous figures represent hardware resources growth and latency as a function of α. Looking at clock cycles $t'B$, the throughput cannot be improved any further after $\alpha = 2$, which confirms the estimated values. This is in part due to the DSP48 and the iteration interval ($II = 1$). Thus, low α values are sufficient for an optimal balance of resources, performance and power consumption.

Fig. 12 (see appendix A Table 5) shows tS^{Ln} as a function of the number of neurons Nx^{Ln} in the $Ln - th$ layer before and after optimization with $\alpha = 2$. In both cases tS^{Ln} grows linearly with Nx^{Ln} keeping an almost constant $tS^{Ln}/t'S^{Ln}$ ratio of order 19.

Fig. 13 (see appendix A Table 6) shows the tAF^{Ln} to produce the output of each neuron according to the number of neurons Nx^{Ln} before and after optimization. In that case, while the required number of clock cycles is smaller than in the previous cases, the ratio $tAF^{Ln}/t'AF^{Ln}$ is more effective, growing from 20 until reaching 36 after $Nx^{Ln} = 600$.

B. SYNTHESIS OF THE ENTIRE IP LAYER

In this section, we will summarize the synthesis results of an entire IP regarding size and resources. Thus, we will evaluate the overall hardware resources needed (HR) as well as the throughput in terms of the total clock cycles (TCC) to carry out operations on IP layers of different dimensions.

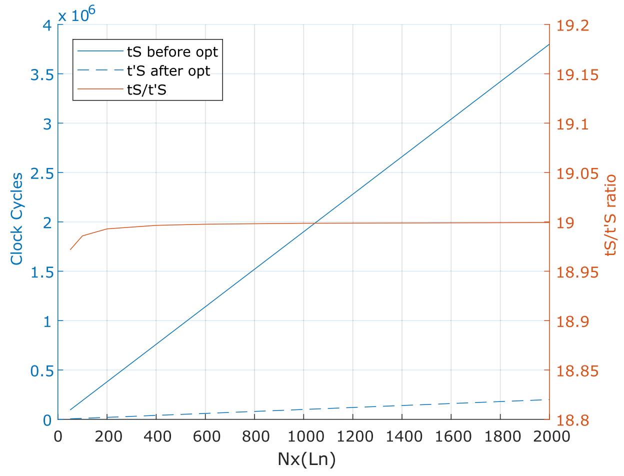

FIGURE 12. tS^{Ln} **(clock cycles) before and after optimization and** $tS^{Ln}/t'S^{Ln}$ **ratio as a function of** Nx^{Ln}.

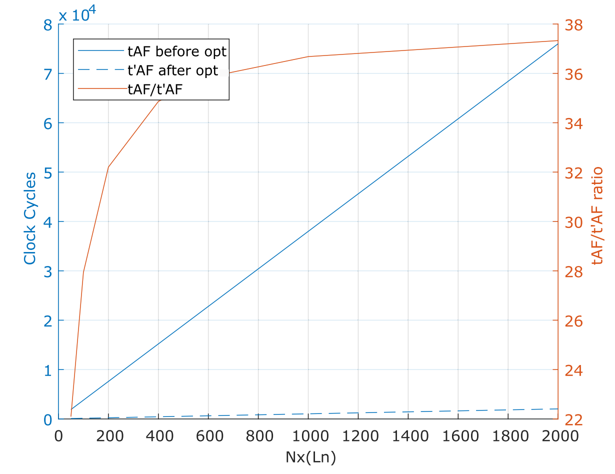

FIGURE 13. tAF^{Ln} **(clock cycles) before and after optimization and** $tAF^{Ln}/t'AF^{Ln}$ **ratio as a function of** Nx^{Ln}.

The first evaluation consisted of the synthesis of an IP layer of different dimensions, with Nx^{Ln-1} inputs in a range from 50 to 600 and Nx^{Ln} outputs from 50 to 2000, adopting 32-bit data representations. Table 7 (appendix A) presents the synthesis results without and with optimizations. The resources are computed mainly in terms of flip-flops and LUTs, as these are the most representative of the FPGA occupation, as shown in Table 1.

Fig. 14 summarizes the interpolation of the results obtained in Table 7 by comparing hardware resources according to the number of inputs and outputs before and after performance optimization. Outputs ranging from 50 to 2000 are represented twice, in the form of values 1 to 7 before optimization and 8 to 14 after performance optimization. Dot values represent synthesis results. The smallest implementation, the one without performance optimization, will be a single PE that iterates over inputs and outputs. In this case, synthesis results are almost independent of the input-output dimensions and the number of neurons. The implementation optimized in terms of parallelism and pipeline uses an $\alpha = 2$, meaning that the number of PE operations will be doubled. However,

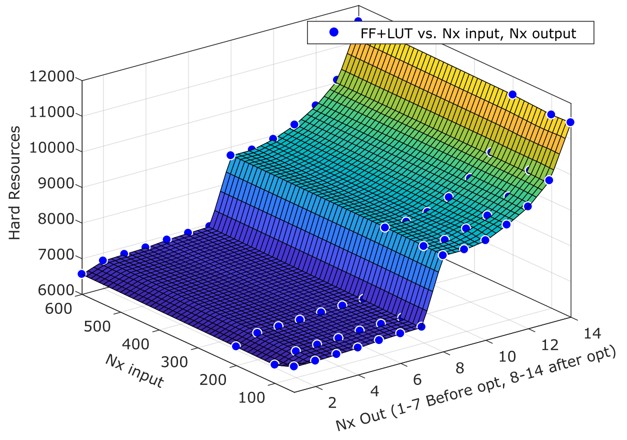

FIGURE 14. Hardware resources (HR) according to the number of inputs Nx^{Ln-1} **and output** Nx^{Ln} **before (from 1 to 7) and after performance optimization (from 8 to 14) using parallelism and pipeline (**$\alpha = 2$**).**

the impact will be a size increase of 66% in the worst case (600 - 2000), and the total size will still be less than 5% FFs and 11% LUTs of the target FPGA.

The synthesis results in terms of performance are detailed in Table 8 (appendix A). Performance is measured in terms of the number of clock cycles (TCC) to process all input data, before (left side) and after optimization (right side). Fig. 15.a and Fig. 15.b shows the interpolation of measured values. Notice that in the non-optimized implementation, the delay grows with both input and output parameters, reaching delays 18 times bigger with 1000 inputs when compared to 50 inputs, and 39 times bigger from 50 to 2000 outputs. In the optimized implementation, the delay follows the same pace growing 18 times from 50 to 1000 inputs and 39 times from 50 to 2000 outputs. However, the delay is almost 19× smaller in the optimized implementation when compared to the non-optimized one.

VIII. EXPERIMENTAL RESULTS

When looking at edge computing, it is important to provide an optimal DNN implementation with regard to size, performance, and power consumption. Although we can automatically generate hardware acceleration kernels supporting various topologies and dimensions, we are still limited by the size of the target reconfigurable architecture. In this regard, we evaluated and compared state-of-the-art DNNs using the MNIST (28 × 28 frame size) dataset [65] as a case study. The Xilinx ZedBoard ZYNQ 7020 was chosen as the target architecture for ease of comparison. This does not prevent to use other boards such as FLASH-based FPGAs or SOMs (e.g, Xilinx Kria) which might be more suitable for use in IoT environments.

A. IMPLEMENTATION RESULTS

A reference topology 784-100-50-10, already trained using the SSAE technique with 32-bit floating-point representations for inputs, weights, and bias, was selected for this

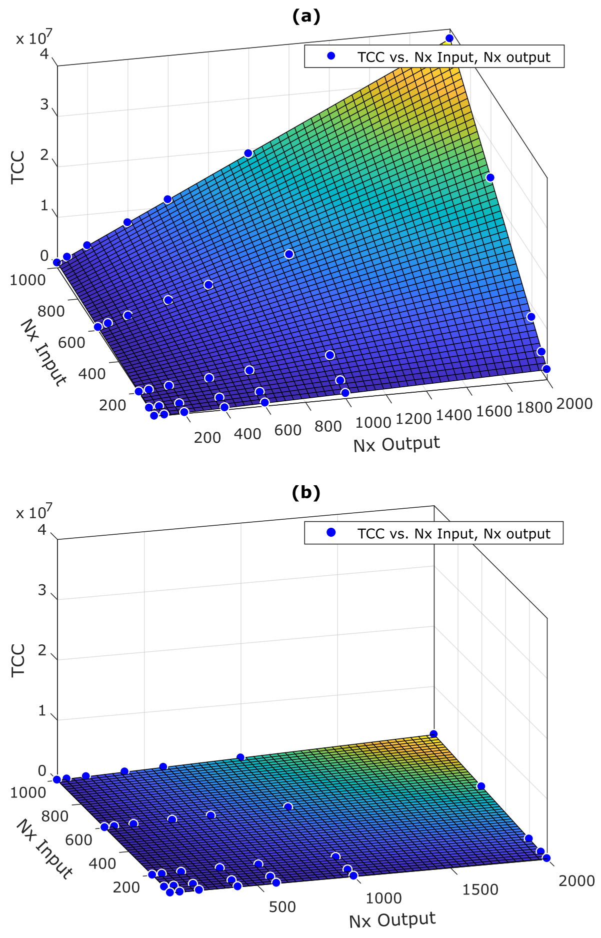

FIGURE 15. The interpolation of TCC vs IP size (Nx^{Ln-1} (inputs) and Nx^{Ln} (out)). Figures (a) and (b) represents the interpolation before and after optimizations, respectively. The delay is almost 19× smaller after optimizations.

study. Notice that this data-type representation is not the best choice for the hardware accelerator in terms of implementation size and processing speed. For this test, we used a Linux program in the host CPU, which interacts with the different layers of the DNN accelerated by the FPGA part. To measure the latency, we use the `perf_counter` class from the `sds-lib` library to collect timestamps and measure their differences [66]. Latency results are shown in clock cycles. To measure the throughput, including the CPU-FPGA data exchange, we use a timer function on the host CPU monitoring the classification of the test images set fetched from the MNIST database. The hardware resources as well as the power consumption are estimated using Xilinx's Vivado tools.

Table 1 lists the resources required to implement this topology. Note that the most important resources are LUTs 73% and FF 41%. Only 20% of BRAMs, 20% of DSPs and 11% of ULTRAM are used, this is due to processing one entry at a time.

Table 2 lists the execution frequency, processing time per image, throughput, and accuracy regarding performance

TABLE 1. Resources needed to implement the DNN 784-100-50-10 topology for the MNIST dataset on the ZYNQ 7020 series.

	LUT	LUTRAM	FF	BRAM	DSP
Units	38840	1916	43630	28	46
Percentage	73%	11%	41%	20%	21%

TABLE 2. Performance of the hardware accelerator.

Max frequency	Latency (in clock cycles)	Throughput (FPS)	Accuracy
100 MHz	578K	1160	99.2%

parameters. The frequency is still quite low, limiting power consumption. As we use 32-bit floating-point data, the accuracy is the same as in the software version, which is quite close to the 99.5% achieved with the 784-150-75-10 topology (see Table 3).

In order to evaluate design exploration capabilities as well as to allow subsequent comparisons with the state of the art, Table 3 shows the most representative topologies generated with our approach. As expected, accuracy is preserved, with state-of-the-art comparable numbers of 96.2% and 99.5%, respectively. However, with the larger dimension topology, we have almost hit the limit in terms of LUTs, 86%, while FFs, memory, and DSPs numbers are less than 50%, 27% and 28% of FPGA capacity, respectively. The estimated power figures are still reasonably low, with 0.26 W for the smallest topology and less than 0.44 W for the largest.

B. COMPARATIVE RESULTS EVALUATION

Keen to evaluate the role of our approach in the context of hardware acceleration, in addition to assess its embedded capabilities, we have selected some reference DL solutions. In the meantime, we have presented the results of four alternative implementations for the same 784-100-50-10 DNN topology while modifying some parameters (i.e. the balance between pipeline / parallel optimizations and interfaces) as shown in the pseudo-code 2.

Table 4 summarizes the results of state-of-the-art work using comparable dimensions for the MNIST (28 × 28 frame size). The table details some implementation and performance parameters such as topology, clock frequency, throughput in Frame per second (FPS), accuracy, power consumption and chip cost. Note that some topologies do not have exactly the same complexity, therefore, we additionally provide the data type, topology complexity (number of Network parameters), and processing speed in terms of Million parameters per second (Mps), which may help to understand the differences in processing speed regarding FPS.

Our first alternative presents an implementation without any optimizations. This is reflected in the throughput result, where only 1.59 Mps and 19 FPS are recorded, which present the lowest results. In the second alternative, we applied a pipeline and parallel optimizations with $\alpha = 2$. The throughput becomes 9.917 Mps and 118 FPS, which means more than 6× faster comparing with the first alternative. The third

TABLE 3. SoC implementation topologies for the MNIST DNN classifier on the ZYNQ 7020 series.

	FPGA implementation results									
Topologies	Throughput (FPS)	Latency (in clock cycles)	Accuracy	LUT	LUTRAM	FF	BRAM	DSP	Power consumption	Freq (MHz)
784-32-32-10	3626	192K	96.2%	55%	10%	32%	20%	21%	0.266W	100
784-100-50-10	1160	578K	99.2%	73%	11%	41%	20%	21%	0.380W	100
784-100-50-20-10	1150	571K	99.1%	86%	14%	49%	26%	27%	0.430W	100
784-150-75-10	751	1170K	99.5%	84%	11%	46%	20%	21%	0.400W	100

TABLE 4. Comparison with state-of-art DNN implementations. (*)All chip prices are taken from "digikey" web site [67]. (**) High is best.(***) Low is best.

	Topology	Chip/Cost*	Frequency	Throughput (FPS)	Data types	Accuracy	Network parameters	Throughput (Mps)**	Power Consumption in Watts and in (mW/Mps)***
M. Coutinho et al. [29]	784-100-50-10	Virtex 6/ $1821	100 MHz	1250	12-bit Fixed	93.3%	84.05K	105.062	0.3 (2.855)
A.Mazouz et al. [62]	1-2-4	Zynq-7100/ $4043	-	526	16-bit Fixed	98.6%	41.71K	21.934	0.578 (26.35)
C. Wang et al. [34]	784-256-256-10	Zynq 7020/ $125	200 MHz	12.45	32-bit Float	-	268.8K	3.346	0.234 (69.93)
Rivera-Acosta et al. [54]	LeNet-5	Cyclone IV EP4CE115/ $340	100 MHz	925	32-bit Float	-	60K	55.500	-
Alternative 1	784-100-50-10	Zynq 7020/ $125	100 MHz	**19**	32-bit Float	99.4%	84.05K	**1.59**	**0.478 (300.6)**
Alternative 2	784-100-50-10	Zynq 7020/ $125	100 MHz	**118**	32-bit Float	99.4%	84.05K	**9.917**	**0.49 (49.4)**
Alternative 3	784-100-50-10	Zynq 7020/ $125	100 MHz	**1160**	32-bit Float	99.4%	84.05K	**97.498**	**0.38 (3.89)**
Alternative 4	784-100-50-10	Zynq 7020/ $125	100 MHz	**1200**	32-bit Float	99.4%	84.05K	**100.86**	**0.39 (3.86)**

alternative is one of the best strategy, where in addition to the previous optimizations of Alternative 2, we adopted an AXI-Stream at the inter-layer connection in order to perform a systolic array technique. The throughout results were improved by 61.3× and 9.8× compared with the first and second alternatives, respectively. The power consumption has also been reduced (from 0.49W to 0.38W), since fewer AXI-MM buses have been implemented in the third alternative. In the fourth alternative, we adopted the same implementation strategy in alternative 3, but with $\alpha = 4$. The performance results remain almost the same as the third alternative with a small improvement in the throughput (from 97 Mps to 100 Mps). However, this implementation occupied almost 100% LUT.

The scalable accelerator for large-scale DL networks DALU by Wang *et al.* [34] references 3 implementation dimensions (784-64-64-10, 784-128-128-10 and 784-256-256-10) on a ZYNQ-7020. Note that instead of choosing the one with the closest dimensions, we have considered the largest, not only because it shows the best results, but also because what can be achieved in terms of implementation size on a small FPGA. However, their best solution in terms of performance is still not fast enough, just 12 FPS (according to values taken from [29]) and 3.346 Mps, due one single layer performed at once, so no chance for a pipeline or systolic array (as we did in our work) techniques between layers as well as the significant waste of time to configure the DLAU for each execution layer. Still, their work present a poor power efficiency (69.93mW/Mps) but, even with 3 times more parameters, and doubled clock frequency and FPGA power, their solution is a comparatively low power (234 mW).

The work in [62] proposes a design flow to automate reconfigurable DL models for FPGAs. They implemented several CNN topologies for the MNIST classifier. Table 4 shows the selected topology 1-2-4, three hidden convolutional layers with 7 filters and 41k parameters. Note that this work targets a much larger FPGA, the 4043 US$ ZYNQ-7100, but it uses 16-bit fixed-point data values instead of 32-bit (affecting the accuracy) and fewer parameters. However, all of this is not reflected in the throughput and the number of operations per second (both still low). Moreover, although the power is not that high, the power efficiency is somehow (26.35mW/Mps).

The work detailed in [29] proposes a SSAE optimized at RTL level to achieve the best system performance in terms of throughput and power consumption. Indeed, this work offers the best throughput and the lowest energy consumption. However, to achieve this performance, they use 12-bit data types, thus providing lower accuracy. Yet this work targets HPC, so powerful boards such as the Virtex 6 are used.

Rivera-Acosta *et al.* [54] presented a tool for automatically generate multiple CNN topologies based on RTL templates. They used the Cyclone IV FPGA for the outcome evaluation. With their LeNet-5 topology, with 32-bit data types, they

TABLE 5. The clock cycles of S^{Ln} before/after optimization and the $tS/t'S$ ratio for different layer size (Nx^{Ln}).

Nx(Ln)	50	100	200	400	600	1000	2000
CC before opt	95200	190200	380200	760200	1140200	1900200	3800200
CC after opt	5018	10018	20018	40018	60018	100018	200018
Ratio	18.97	18.98	18.99	18.99	18.99	18.99	18.99

TABLE 6. The clock cycles of AF^{Ln} loop before/after pipeline and the $tAF/t'AF$ ratio for different layer size (Nx^{Ln}).

NxLn	50	100	200	400	600	1000
CC AF Loop before opt	1900	3800	7600	15200	22800	38000
CC AF Loop after opt	86	136	236	436	636	1036
Rate of acceleration	22.09	27.94	32.2	34.86	35.84	36.67

TABLE 7. IP size in hardware resources (FF and LUT) before and after performance optimization according to the number of inputs Nx^{Ln-1} and outputs Nx^{Ln}. Performance optimization uses parallelism and pipeline ($\alpha = 2$).

IN - Out	Before optimizations (FF+LUT)					After optimizations (FF+LUT)				
	50	200	600	1000	2000	50	200	600	1000	2000
50	6712	6735	6802	6830	6852	8698	8795	9419	9991	11458
200	6526	6765	6832	6860	6882	8726	8837	9462	10026	11487
600	6560	6799	6877	6905	6927	8756	8893	9508	10062	11545
Perc (FF, LUT)	3%,6%	3%,6%	3%,6%	3%,6%	3%,6%	4%,7%	4%,7%	4%,7%	5%,9%	5%,11%

TABLE 8. IP-layer delay in clock cycles (TCC) before and after performance optimization according to the number of inputs Nx^{Ln-1} and outputs Nx^{Ln}. Performance optimization uses parallelism and pipeline ($\alpha = 2$).

	Before optimizations						**After optimizations (SpeedUp)**					
Input\ Out	50	200	600	1000	2000	50-2000 Difference	50	200	600	1000	2000	50-2000 Difference
50	49919	199319	597719	996119	1992119	38.9	2747 (18,2)	10773 (18,5)	32173 (18,6)	53573 (18,6)	107073 (18,6)	38
200	192719	769619	2308019	3846419	7692419	38.9	10247 (18,8)	40773 (18,9)	122173 (18,9)	203573 (18,9)	407073 (18,9)	38.7
600	573519	2290419	6868819	11447219	22893219	38.9	30247 (19)	120773 (19)	356138 (19,3)	603573 (19)	1207073 (19)	38.9
1000	954319	3811219	11429619	19048019	38094019	38.9	50247 (19)	200773 (19)	602173 (19)	1003573 (19)	2007073 (19)	38.9
50-1000 Difference	18.1	18.1	18.1	18.1	18.1		17.3	17.6	17.7	17.7	17.7	

achieved a high throughput of 925 FPS. However, with a small number of parameters compared to others, the number of parameters processed per second (Mps) is still low. Unfortunately, they did not provide information on power consumption.

Our automated approach provided comparatively outstanding results in terms of throughput and parameters processed per second (compared to [29] and [54]), even based on high-level synthesis and using 32-bit data types (as in [34] and [54]). Moreover, the power and energy figures are quite close to those based on optimized RTL models (as in [29]), thus offering the best compromise between performance, energy consumption, and system cost.

IX. CONCLUSION AND FUTURE WORK

In this work, we proposed an FPGA-based framework for the automated implementation of hardware-accelerated DNN architectures for edge computing embedded applications. This flexible development environment provides a fast and smooth implementation of DNN alternatives using commercial synthesis tools in a transparent manner. We provide a library of hardware components called IP layers, flexible enough to support the implementation of any DNN topology, as they provide the structure for the different layers of the DNN (input, hidden, and output layers). IP layers can be configured to enhance performance according to topology requirements and target platform constraints. IP layers can also be encapsulated and customized using HLS tools, facilitating AXI-based interfacing to commercial IP-cores.

In order to evaluate and compare our proposal with the state-of-the-art results regarding implementation size and performance, we used the MNIST dataset. Synthesis results were provided by VIVADO HLS using the ZedBoard (Xilinx ZYNQ 7020 SoC) as the target platform, as we are focusing on embedded SoCs. Some DNN topologies and variants were also built to evaluate estimation models. Comparisons with leading-edge work on throughput, chip cost, hardware occupancy, and power consumption, among other parameters, have shown that our proposed automated design framework offers the best compromise between performance, energy, and system costs. In this regard, the best performance was obtained for the 784-32-32-10 MNIST architecture, which achieved a throughput of 3626 FPS and consumed only 0.266 W.

It is important to note that the results already discussed were obtained using data types and networks of identical dimensions to make fair comparisons. However, data types can be adapted to fixed-point representations with custom bit dimensions to further improve performance and hardware resources, but this will sometimes negatively impact accuracy. Thus, data type resizing should be performed at front and before specification. In addition, as has been proposed by other approaches, same type of layers, as well as the full network, can be merged into a single reusable instance. However, this solution is preferable only for deep networks that cannot fit on a single FPGA, while still accepting performance degradation. These alternative designs,which may be proposed when hardware resource constraints outweigh performance degradation, will be considered as our future work on network resizing strategies.

APPENDIX

DETAILED EVALUATION RESULTS

Tables 5-8 detail the performance and synthesis results.

REFERENCES

[1] B.-H. Li, B.-C. Hou, W.-T. Yu, X.-B. Lu, and C.-W. Yang, "Applications of artificial intelligence in intelligent manufacturing: A review," *Frontiers Inf. Technol. Electron. Eng.*, vol. 18, no. 1, pp. 86–96, 2017. [Online]. Available: http://link.springer.com/10.1631/FITEE.1601885

[2] P. Hamet and J. Tremblay, "Artificial intelligence in medicine," *Metabolism, Clin. Experim.*, vol. 69, pp. S36–S40, Apr. 2017.

[3] Q.-V. Pham, D. C. Nguyen, T. Huynh-The, W.-J. Hwang, and P. N. Pathirana, "Artificial intelligence (AI) and big data for coronavirus (COVID-19) pandemic: A survey on the state-of-the-arts," *IEEE Access*, vol. 8, pp. 130820–130839, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9141265/

[4] S. Zeadally, E. Adi, Z. Baig, and I. A. Khan, "Harnessing artificial intelligence capabilities to improve cybersecurity," *IEEE Access*, vol. 8, pp. 23817–23837, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8963730/

[5] Y. Ma, Z. Wang, H. Yang, and L. Yang, "Artificial intelligence applications in the development of autonomous vehicles: A survey," *IEEE/CAA J. Automatica Sinica*, vol. 7, no. 2, pp. 315–329, Mar. 2020.

[6] K. Atkinson, T. Bench-Capon, and D. Bollegala, "Explanation in AI and law: Past, present and future," *Artif. Intell.*, vol. 289, Dec. 2020, Art. no. 103387. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0004370220301375

[7] Y. K. Dwivedi, L. Hughes, E. Ismagilova, G. Aarts, C. Coombs, T. Crick, Y. Duan, R. Dwivedi, J. Edwards, A. Eirug, and V. Galanos, "Artificial intelligence (AI): Multidisciplinary perspectives on emerging challenges, opportunities, and agenda for research, practice and policy," *Int. J. Inf. Manage.*, vol. 57, Aug. 2019, Art. no. 101994. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S026840121930917X

[8] A. Krogh, "What are artificial neural networks?" *Nature Biotechnol.*, vol. 26, no. 2, pp. 195–197, 2008. [Online]. Available: http://www.nature.com/articles/nbt1386

[9] D. J. Livingstone, *Artificial Neural Networks* (Methods in Molecular Biology), vol. 458, T. D. J. Livingstone, Ed. Totowa, NJ, USA: Humana Press, 2009. [Online]. Available: http://link.springer.com/10.1007/978-1-60327-101-1

[10] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, Jul. 2006. [Online]. Available: https://www.mitpressjournals.org/doi/abs/10.1162/neco.2006.18.7.1527

[11] P. Kim, *MATLAB Deep Learning*. Berkeley, CA, USA: Apress, 2017.

[12] I. El Naqa and M. J. Murphy, *What is Machine Learning?* Cham, Switzerland: Springer, 2015.

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[14] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Netw.*, vol. 61, pp. 85–117, Jan. 2015, doi: 10.1016/j.neunet.2014.09.003.

[15] P. Mamoshina, A. Vieira, E. Putin, and A. Zhavoronkov, "Applications of deep learning in biomedicine," *Mol. Pharmaceutics*, vol. 13, no. 5, pp. 1445–1454, May 2016.

[16] P. Baldi, "Autoencoders, unsupervised learning, and deep architectures," in *Proc. ICML Workshop Unsupervised Transf. Learn.*, Bellevue, WA, USA, Jun. 2012. [Online]. Available: http://proceedings.mlr.press/v27/baldi12a.html

[17] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, May 2015. [Online]. Available: http://www.nature.com/articles/nmeth.3707 and http://www.nature.com/articles/nature14539

[18] A. Voulodimos, N. Doulamis, A. Doulamis, and E. Protopapadakis, "Deep learning for computer vision: A brief review," *Comput. Intell. Neurosci.*, vol. 2018, pp. 1–13, Feb. 2018.

[19] H. Purwins, B. Li, T. Virtanen, J. Schlüter, S.-Y. Chang, and T. Sainath, "Deep learning for audio signal processing," *IEEE J. Sel. Topics Signal Process.*, vol. 13, no. 2, pp. 206–219, May 2019.

[20] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," *IEEE Comput. Intell. Mag.*, vol. 13, no. 3, pp. 55–75, Aug. 2018.

[21] H. A. Pierson and M. S. Gashler, "Deep learning in robotics: A review of recent research," *Adv. Robot.*, vol. 31, no. 16, pp. 821–835, Aug. 2017. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/01691864.2017.1365009

[22] Y. Li, C. Huang, L. Ding, Z. Li, Y. Pan, and X. Gao, "Deep learning in bioinformatics: Introduction, application, and perspective in the big data era," *Methods*, vol. 166, pp. 4–21, Aug. 2019.

[23] H. Ben Fredj, S. Bouguezzi, and C. Souani, "Face recognition in unconstrained environment with CNN," *Vis. Comput.*, vol. 37, no. 2, pp. 217–226, Feb. 2021.

[24] H. Faiedh, S. Hamdi, S. Bouguezzi, W. Farhat, and C. Souani, "Architectural exploration of multilayer perceptron models for on-chip and real-time road sign classification," *Proc. Inst. Mech. Eng., I, J. Syst. Control Eng.*, vol. 232, no. 6, pp. 772–783, Jul. 2018.

[25] J. Huang, J. Chai, and S. Cho, "Deep learning in finance and banking: A literature review and classification," *Frontiers Bus. Res. China*, vol. 14, no. 1, p. 13, Dec. 2020.

[26] R. Vinayakumar, M. Alazab, K. P. Soman, P. Poornachandran, A. Al-Nemrat, and S. Venkatraman, "Deep learning approach for intelligent intrusion detection system," *IEEE Access*, vol. 7, pp. 41525–41550, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8681044/

[27] Y. Xin, L. Kong, Z. Liu, Y. Chen, Y. Li, H. Zhu, M. Gao, H. Hou, and C. Wang, "Machine learning and deep learning methods for cybersecurity," *IEEE Access*, vol. 6, pp. 35365–35381, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8359287/

[28] J. Maria, J. Amaro, G. Falcao, and L. A. Alexandre, "Stacked autoencoders using low-power accelerated architectures for object recognition in autonomous systems," *Neural Process. Lett.*, vol. 43, no. 2, pp. 445–458, Apr. 2016. [Online]. Available: http://link.springer.com/10.1007/s11063-015-9430-9

[29] M. G. F. Coutinho, M. F. Torquato, and M. A. C. Fernandes, "Deep neural network hardware implementation based on stacked sparse autoencoder," *IEEE Access*, vol. 7, pp. 40674–40694, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8678408/

[30] D. Sheet, S. P. K. Karri, A. Katouzian, N. Navab, A. K. Ray, and J. Chatterjee, "Deep learning of tissue specific speckle representations in optical coherence tomography and deeper exploration for *in situ* histology," in *Proc. IEEE 12th Int. Symp. Biomed. Imag. (ISBI)*, Apr. 2015, pp. 777–780. [Online]. Available: http://ieeexplore.ieee.org/document/7163987/

[31] Y. Xiao, J. Wu, Z. Lin, and X. Zhao, "A semi-supervised deep learning method based on stacked sparse auto-encoder for cancer prediction using RNA-seq data," *Comput. Methods Programs Biomed.*, vol. 166, pp. 99–105, Nov. 2018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0169260718304553

[32] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC," in *Proc. Int. Conf. Field-Programmable Technol. (FPT)*, Dec. 2016, pp. 77–84. [Online]. Available: http://ieeexplore.ieee.org/document/7929192/

[33] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr, "Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC," in *Proc. 26th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2016, pp. 1–4. [Online]. Available: http://ieeexplore.ieee.org/document/7577314/

[34] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "DLAU: A scalable deep learning accelerator unit on FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 3, pp. 513–517, Jul. 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7505926/

[35] S. Lahti, P. Sjovall, J. Vanne, and T. D. Hamalainen, "Are we there yet? A study on the state of high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 5, pp. 898–911, May 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8356004/

[36] Intel. *OpenVINO*. Accessed: Jun. 10, 2021. [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html

[37] AWS. (2019). *Amazon EC2 F1 Instances*. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[38] L. Semiconductor. *Lattice sensAI Stack, Accelerate Integration of Flexible, Low Power Inferencing at the Edge*. Accessed: Jun. 10, 2021. [Online]. Available: https://www.latticesemi.com/Solutions/Solutions/SolutionsDetails02/sensAI

[39] Xilinx. *Kria K26*. Accessed: Jun. 10, 2021. [Online]. Available: https://www.xilinx.com/products/som/kria.html

[40] N. K. Jayakodi, S. Belakaria, A. Deshwal, and J. R. Doppa, "Design and optimization of energy-accuracy tradeoff networks for mobile platforms via pretrained deep models," *ACM Trans. Embedded Comput. Syst.*, vol. 19, no. 1, pp. 1–24, Feb. 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3366636

[41] W. Jiang, X. Zhang, E. H.-M. Sha, L. Yang, Q. Zhuge, Y. Shi, and J. Hu, "Accuracy vs. efficiency: Achieving both through FPGA-implementation aware neural architecture search," in *Proc. 56th Annu. Design Autom. Conf.* New York, NY, USA: ACM, Jun. 2019, pp. 1–6. [Online]. Available: https://dl.acm.org/

[42] W. Farhat, S. Sghaier, H. Faiedh, and C. Souani, "Design of efficient embedded system for road sign recognition," *J. Ambient Intell. Humanized Comput.*, vol. 10, no. 2, pp. 491–507, Feb. 2019.

[43] C. Yvanoff-Frenchin, V. Ramos, T. Belabed, and C. Valderrama, "Edge computing robot interface for automatic elderly mental health care based on voice," *Electronics*, vol. 9, no. 3, p. 419, Feb. 2020.

[44] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. O. G. Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh, "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?" in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*. New York, NY, USA: ACM Press, Feb. 2017, pp. 5–14. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3020078.3021740

[45] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 11, pp. 2072–2085, Nov. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8497058/

[46] 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-Programmable Custom Comput. Mach. (FCCM)*, May 2016, pp. 40–47.

[47] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-M. Hwu, and D. Chen, "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs," in *Proc. Int. Conf. Comput.-Aided Design*. New York, NY, USA: ACM, Nov. 2018, pp. 1–8. [Online]. Available: https://dl.acm.org/doi/10.1145/3240765.3240801

[48] Nvidia. *NVIDIA V100 TENSOR CORE GPU*. Accessed: Jun. 10, 2021. [Online]. Available: https://www.nvidia.com/en-us/data-center/v100/

[49] 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 Autom. Conf.* New York, NY, USA: ACM, Jun. 2016, pp. 1–6. http://dl.acm.org/citation.cfm?doid=2897937.2898003, http://dl.acm.org/citation.cfm?doid=2897937.2898002, and https://dl.acm.org/doi/10.1145/2897937.2898003

[50] (2016). *GPU vs FPGA Performance Comparison*. [Online]. Available: http://www.bertendsp.com

[51] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks," in *Proc. 27th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2017, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/document/8056824/

[52] A. G. Blaiech, K. Ben Khalifa, C. Valderrama, M. A. C. Fernandes, and M. H. Bedoui, "A survey and taxonomy of FPGA-based deep learning accelerators," *J. Syst. Archit.*, vol. 98, pp. 331–345, Sep. 2019, doi: 10.1016/j.sysarc.2019.01.007.

[53] 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-Programmable Custom Comput. Mach. (FCCM)*, Apr. 2017, pp. 152–159. [Online]. Available: http://vast.cs.ucla.edu/sites/default/files/publications/fccm2017.pdf and http://ieeexplore.ieee.org/document/7966671/

[54] M. Rivera-Acosta, S. Ortega-Cisneros, and J. Rivera, "Automatic tool for fast generation of custom convolutional neural networks accelerators for FPGA," *Electronics*, vol. 8, no. 6, p. 641, Jun. 2019. [Online]. Available: https://www.mdpi.com/2079-9292/8/6/641

[55] T. Belabed, M. G. F. Coutinho, M. A. C. Fernandes, V. Carlos, and C. Souani, "Low cost and low power stacked sparse autoencoder hardware acceleration for deep learning edge computing applications," in *Proc. 5th Int. Conf. Adv. Technol. Signal Image Process. (ATSIP)*. Sousse, Tunisia: IEEE, Sep. 2020, pp. 1–6.

[56] H. T. Kung and C. E. Leiserson, "Systolic arrays for (VLSI)," Carnegie-Mellon Univ., Pittsburgh, PA, USA, Tech. Rep., 1978, p. 29.

[57] H. Waris, C. Wang, W. Liu, and F. Lombardi, "AxSA: On the design of high-performance and power-efficient approximate systolic arrays for matrix multiplication," *J. Signal Process. Syst.*, vol. 93, no. 6, pp. 605–615, Jun. 2021.

[58] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *IEEE Trans. Neural Netw.*, vol. 1, no. 1, pp. 71–80, Feb. 2015.

[59] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Proc. Design Autom. Conf.*, vol. 12828. Piscataway, NJ, USA: IEEE, Jun. 2017.

[60] H. Ye, X. Zhang, Z. Huang, G. Chen, and D. Chen, "Hybrid-DNN: A framework for high-performance hybrid DNN accelerator design and implementation," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, Jul. 2020, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/9218684/

[61] P. G. Mousouliotis and L. P. Petrou, "CNN-grinder: From algorithmic to high-level synthesis descriptions of CNNs for low-end-low-cost FPGA SoCs," *Microprocessors Microsyst.*, vol. 73, Mar. 2020, Art. no. 102990, doi: 10.1016/j.micpro.2020.102990.

[62] A. Mazouz and C. P. Bridges, "Automated offline design-space exploration and online design reconfiguration for CNNs," in *Proc. IEEE Conf. Evolving Adapt. Intell. Syst. (EAIS)*, May 2020, pp. 1–9. [Online]. Available: https://ieeexplore.ieee.org/document/9122697/

[63] Xilinx, "UG1037, AXI reference guide, V4.0," Tech. Rep., Xilinx, San Jose, CA, USA, 2017.

[64] A. Limited, "Introduction to AMBA AXI4," A. Limited, London, U.K., Tech. Rep. 0101, 2020. [Online]. Available: https://developer.arm.com/architectures/learn-the-architecture/introduction-to-amba-axi/axi-protocol-overview

[65] Y. LeCun, C. Cortes, and C. J. Burges. *THE MNIST DATABASE of Handwritten Digits*. Accessed: Jun. 10, 2021. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[66] *SDSoC Environ. User Guide (V2016.4)*, Xilinx, San Jose, CA, USA, 2017.

[67] (2021). *Digikey*. [Online]. Available: https://www.digikey.com/

TAREK BELABED received the B.Sc. degree in applied science and technologies and the M.Sc. degree in science and technology from the ISSATSo, University of Sousse, Tunisia, in 2014 and 2016, respectively. He is currently pursuing the Ph.D. degree. He is currently a part of the Microelectronics and Instrumentation Laboratory, Faculty of Sciences of Monastir, Tunisia, and the Electronics and Microelectronics (SEMi) Unit, Polytech Faculty of Mons, Belgium. His main research is the new hardware acceleration architecture and automated framework methodologies for deep neural network (DNN) approaches based on FPGA and embedded systems. His profile and research interests are oriented toward edge computing, re-configurable hardware, embedded systems, EDA, unified hardware tools, RTL design, cloud computing using hardware accelerators, and deep learning solution approaches.

MARIA GRACIELLY F. COUTINHO was born in Natal, Brazil. She received the B.S. degree in computer science from the State University of Rio Grande do Norte, Natal, in 2017, and the M.Sc. degree in electrical and computer engineering from the Federal University of Rio Grande do Norte, Natal, in 2019, where she is currently pursuing the Ph.D. degree in electrical and computer engineering. She is currently a Team Member of the Research Group on Embedded Systems and Reconfigurable Computing. Her main research topics are the viral genome classification using deep learning techniques and the acceleration of deep learning algorithms through reconfigurable computing on FPGA. Her research interests include artificial intelligence, embedded systems, reconfigurable hardware, and viral genome analysis.

MARCELO A. C. FERNANDES was born in Natal, Brazil. He received the B.S. and M.S. degrees in electrical engineering from the Federal University of Rio Grande do Norte, Natal, in 1997 and 1999, respectively, and the Ph.D. degree in electrical engineering from the University of Campinas, Campinas, Brazil, in 2010. He is currently an Associate Professor with the Department of Computer Engineering and Automation, Federal University of Rio Grande do Norte. He is also a Visiting Scholar with the John A. Paulson School of Engineering and Applied Sciences, Harvard University, Cambridge, USA. From 2015 to 2016, he worked with a Visiting Researcher with the Centre Telecommunication Research (CTR), King's College London, London, U.K. He is the Leader of the Research Group on Embedded Systems and Reconfigurable Computing (RESRC) and Coordinator of the Laboratory of Machine Learning and Intelligent System (LMLIS). He is the author and coauthor of many scientific articles and practical studies with reconfigurable computing on FPGA to accelerate artificial intelligence algorithms. His research interests include artificial intelligence, digital signal processing, embedded systems, reconfigurable hardware, and tactile internet.

CARLOS VALDERRAMA SAKUYAMA (Senior Member, IEEE) received the Diploma degree in electrical-electronics engineering from the National University of Cordoba, Argentina, in 1989, the M.Sc. degree in diploma from the Federal University of Rio de Janeiro, Brazil, in 1993, and the Ph.D. degree in microelectronics from the TIMA Laboratory, Institute Nationale Polytechnique de Grenoble INPG (currently Grenoble Institute of Technology), France, in 1998. He was an invited Professor in several universities, the Catholic University of Cordoba, Argentina, the Federal University of Pernambuco, the Federal University of Rio Grande do Norte, Brazil, and the University of Castilla La Mancha, Spain. He has been the Director of the Electronics and Microelectronics Department, Polytechnic Faculty, University of Mons, Belgium, since 2004. He is a member of the New Media Art Technology and the Information Technology Institutes. He was responsible for the creation of the spinoff Nsilition (2009, funded by the Walloon Region). He has participated in more than 18 national and international research projects from the development of 4G chips, tracking devices and architectures for the IoT, HPC, and space. He serves as a technical reviewer and a committee member of multiple journals and international conferences. His research activity is supported by more than 180 publications on international conferences, more than ten books chapters, and more than 30 scientific journals.

CHOKRI SOUANI is currently a Professor in electronics and microelectronics with the Higher Institute of Applied Sciences and Technology Sousse, Tunisia. His research interests include software-defined systems, SDR, SD-SoC, MPSoC, embedded systems, computer vision, big data, the IoT, smart city, communicant vehicle, ITS, and small satellite and applications.