Spatio-Temporal Optimization of Deep Neural Networks for Reconfigurable FPGA SoCs

Biruk Seyoum, *Student Member, IEEE,* Marco Pagani, *Student Member, IEEE,* Alessandro Biondi, *Member, IEEE,* Sara Balleri, and Giorgio Buttazzo, *Fellow, IEEE*

Abstract—This paper proposes a technique for optimizing the timing performance and the resource consumption of hardware accelerators for deep neural network (DNN) inference on FPGA-based system-on-chips (SoC). When required, the accelerators are decomposed into chunks, each exploiting at best the available FPGA area, and dynamic partial reconfiguration (DPR) is leveraged to schedule such chunks at run-time. To this end, the paper presents accurate models of the resource consumption and timing of DNN accelerators provided by the Xilinx FINN framework. The models are then used to formulate an optimization problem that computes the optimal decomposition of DNN accelerators (and their configuration) by minimizing the inference time while ensuring area constraints on the FPGA. Experimental results on Zynq-7000 platforms demonstrate that the proposed technique provides consistent improvements with respect to both stock configurations of the accelerators and other configurations that can be obtained with a static FPGA allocation.

1 Introduction

Over the last few years, deep neural networks (DNNs) reached impressive performance in several application scenarios such as image processing, data analysis, and control [1]. Due to these developments, DNNs are increasingly taken in consideration for developing new functionality in *cyber-physical systems* (CPS) such as automated vehicles [2], [3], next-generation factory automation [4], [5] advanced robotics [6], [7], and anomaly detection in edge devices [8]. As DNNs are characterized by a high computational complexity, some form of hardware acceleration is required to use them in real time. System-on-chips (SoC) that integrate both classical multiprocessors and a field-programmable gate array (FPGAs) are promising candidates to build CPS that require hardware-accelerated DNNs. Indeed, they allow deploying flexible, powerful, time-predictable, and energy-efficient hardware accelerators on the FPGA fabric.

Two major approaches have been proposed [9] to accelerate DNNs on FPGAs. One consists in deploying a series of accelerators that can perform the fundamental mathematical operations required by DNNs (such as convolutions) in an efficient way. The other one consists in deploying the entire DNN on the FPGA, i.e., with a series of pipelined accelerators specialized for a given network structure, even already including the DNN's weights. The latter is clearly less flexible but tends to provide better efficiency as network-specific optimized accelerators can be synthesized. The FINN framework [10] by Xilinx follows this second approach and is the focus of this paper. The accelerators produced by FINN can be tuned to trade FPGA area consumption with latency and throughput. At a high level, being the operations performed by the accelerators largely suitable to parallelization, performance can be improved by increasing the number of processing engines inside each accelerator. This however comes at the cost of a larger area consumption.

Unfortunately, the area consumption of FINN accelerators cannot be arbitrarily reduced. This is because they require resources to store the DNN's parameters and other accessory logic that do not change when the degree of parallelism is reduced. This originates a lower bound on the area consumption required by FINN, with the consequence that it may be impossible to deploy it on resource-constrained systems. For instance, on PYNQ-Z1 by Xilinx, it might be impossible to deploy a FINN-based accelerator along with other standard IPs such as HDMI encoder and the associated video DMA engines.

Contribution. This work addresses this issue by proposing a technique to optimize the timing performance and the resource consumption of FINN accelerators. When required, the accelerators are decomposed into chunks such that each chunk is configured to exploit at best the available FPGA area. Then, *dynamic partial reconfiguration* (DPR) of the FPGA is leveraged to execute the chunks at run-time. The DNN decomposition and the configuration of the accelerators are based on accurate models of the resource consumption and timing for FINN that have been experimentally derived. Both decomposition and configuration are performed by means of a *mixed-integer linear program* (MILP) that aims at optimizing the timing performance while ensuring area constraints. The proposed technique is validated with experiments on a PYNQ-Z1 by Xilinx. Experimental results show that **(i)** our approach makes possible to deploy FINN accelerators when it would be impossible without DPR, and that **(ii)** it provides consistent improvements with respect to both stock FINN configurations and other configurations that can be obtained with a static FPGA allocation.

- *B. Seyoum, M. Pagani, A. Biondi and G. Buttazzo are with the TeCIP Institute of the Scuola Superiore Sant'Anna, Pisa 56124, Italy. E-mail: {biruk.seyoum, marco.pagani, alessandro.biondi, giorgio.buttazzo}@santannapisa.it.*
- *M. Pagani is also with with the Embedded Real-Time Adaptative System Design and Execution (Émeraude) team from the Centre de Recherche en Informatique, Signal et Automatique (CRIStAL) based in Lille*

2 Background

This section provides a concise discussion on reduced-precision DNNs and the general architecture of the FINN framework.

2.1 Reduced-precision DNNs

Several works have faced the problem of optimizing DNNs to the purpose of reducing their inference time and memory footprint: please refer to [11] for a survey on the topic. One of the most common approaches to address this problem consists in reducing the precision of network parameters by optimizing the size (in bits) of the weights. While DNNs typically come with 32-bit floating point weights, several researches found that comparable performance can be obtained if the precision of the weights is reduced, e.g., shifting to 16-bit floating point weights by simply reducing their precision, or adopting integer representations by quantizing the numerical domain of the weights. As an extreme application of such an approach, several researches [10], [12], [13], [14] also proposed to adopt DNN models with *binary* parameters (i.e., weights and biases). The resulting DNNs are called *binary neural networks* (BNNs) and are notably the most time- and memory-efficient class of DNNs. The efficiency of BNNs is originated by two main properties. On one hand, due to the extreme reduction of the size (bit-width) of the weights, memory access times during inference and the memory footprint are significantly reduced. On the other hand, the arithmetic floating-point computations required by standard DNNs can be replaced with simpler and faster bitwise operations, which also significantly improves power efficiency [12]. Both these properties make BNNs well suited for an efficient implementation on FPGA platforms, which often have a limited on-chip memory (OCM) to store the weights but provide a high flexibility in tailoring the accelerator hardware according to the required precision. Recently, BNNs have also achieved accuracies very close to the ones of full-precision networks for some applications [12], [13].

2.2 The FINN framework

FINN is an experimental framework from Xilinx Research Labs [10] to support the development of scalable accelerators to perform the inference of quantized DNNs on FPGAs. It relies on a heterogeneous streaming architecture arranged as a pipeline, where each layer of a DNN is mapped to a dedicated processing engine named *matrix-vector-threshold unit* (MVTU). The MVTUs communicate among themselves using data streams. The MVTU is built from parameterizable building blocks that can be scaled according to a set of requirements. The majority of operations in fully-connected BNNs can directly be expressed as matrix-vector product of the weights and the input activations followed by thresholding. In convolutional layers, operations can also be reduced as matrix-matrix operations by suitably arranging the weights and the incoming feature maps in each layer [10]. The MVTU implements fully-connected layers as standalone components, while convolutional layers are implemented by adding a sliding window unit that pre-arranges the incoming feature map before performing a matrix-matrix multiplication.

Internally, an MVTU consists of an array of *processing elements* (PEs) that correspond to hardware neurons. Please refer to Figure 2(a). The synapses of each neuron are processed via *single instruction multiple data* (SIMD) lanes. The MVTU in the i^{th} layer can be represented by the tuple $< P_i, S_i >$, where P_i denotes the number of PEs in the i^{th} layer and S_i denotes the number of SIMD lanes (synapses) of each PE in the same layer. FINN accelerators are designed such that a single PE with S_i SIMD lanes can simultaneously process S_i synapses in one clock cycle. For each layer, the parameters P_i and S_i can be configured according to the desired throughput and latency, but also have a direct impact in terms of area consumption of the resulting accelerator. In other words, the configuration of the parameters P_i and S_i requires facing with a trade-off between performance (throughput and latency) and area consumption. For instance, by increasing the number of PEs and SIMD lanes it is possible to reduce the inference time of the network; however, the resulting accelerator requires a larger amount of resources on the FPGA fabric.

3 Modeling FINN accelerators

This paper considers a set of BNNs developed with the BNN-FINN framework [15]. In particular, it focuses on the CNVW1A1 and CNVW2A2 quantized networks [15]. Both neural networks use quantized parameters except for the input and output layers. Despite both networks have similar architectures, the CNVW1A1 network uses a 1-bit precision for weights and activations parameters, while the CNVW2A2 uses a 2-bit precision.

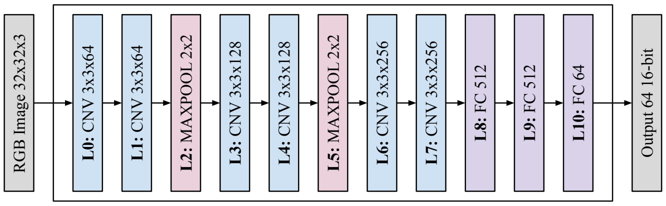


Fig. 1. Illustration of the CNVW1A1 BNN from Xilinx. The characteristics of each layer are specified by the numbers with which they are labeled. For example, CNV 3x3x64 denotes a convolutional layer with a 3x3 filter and 64 channels, FC 512 denotes a fully-connected layer with 512 neurons, and MAXPOOL 2x2 denotes a max-pooling layer that works with a 2x2 window.

Figure 1 illustrates the CNVW1A1 network, whose topology is inspired by the VGG-16 architecture and consists of 6 convolutional layers (CNV in the figure), 2 max-pooling layers (MAXPOOL in the figure), and 3 fully connected layers (FC in the figure). All layers are implemented in hardware through a feed-forward dataflow architecture. The parameters of all layers are stored inside the FPGA on-chip memory. In these implementations, all layers are quantized except for the input and the output layers. The former performs a fixed-precision convolutional operation on a 32x32 image with 8-bit RGB channels, while the latter produces output values with a 16-bit representation. The network can be trained to classify up to 64 classes.

Number of operations in each layer. In the following we denote by *inference operation* (IOP) the computation required to process a synapse. The number of IOPs performed by each layer depends on the type of the layer. If the i^{th} layer is *convolutional*, it can be represented by the tuple $< K_i, \text{IFM}_i, \text{OFM}_i, \text{IFM\_CH}_i, \text{OFM\_CH}_i >$, where K_i is the size of the convolutional kernel, IFM_i is the size of the input feature map, OFM_i is the size of the output feature map, and IFM_CH_i and OFM_CH_i are the number of channels in the input and the output of the layer, respectively. Hence the total number $\#\text{IOPs}_i$ of IOPs performed by the i^{th} layer is given by (please refer to [15] for details)

$$\#\text{IOPs}_i = K_i^2 \cdot \text{OFM}_i^2 \cdot \text{IFM\_CH}_i \cdot \text{OFC\_CH}_i. \quad (1)$$

If the i^{th} layer is fully connected, it can be represented by the tuple $< H_i, W_i >$, where H_i denotes the number of neurons of the layer (height), and W_i denotes the number of synapses per each neuron in the layer (width). Hence the total number $\#\text{IOPs}_i$ of IOPs performed by the i^{th} layer is given by

$$\#\text{IOPs}_i = H_i \cdot W_i. \quad (2)$$

The max-pooling layer in FINN is implemented with just a boolean OR operator (it does not expose the configuration parameters P_i and S_i). The IOPs performed by max-pooling layers are independent of the configuration of the other layers. Furthermore, their contribution to the overall inference latency has been experimentally found to be negligible via profiling. For this reason, the number of IOPs performed by max-pooling layers is not modeled here.

Folding. To explain folding, let us consider the case of the CNVW1A1 network (similar considerations apply to the CNVW2A2 network). To make the network fully parallel, i.e., to process one input image every clock cycle at each layer, the MVTUs in each layer must be configured with a number of PEs and SIMD lanes equivalent to the number of neurons and synapses in the corresponding layer. Clearly, this results in a very large demand of FPGA resources (area). If the FPGA resources available on a platform are not enough to deploy a fully-parallel configuration of the network, the number of PEs and SIMD lanes in each layer must be reduced to implement an accelerator that performs time-multiplexed computations within each layer. In this case, the neurons and the synapses must be correctly partitioned between the PEs inside the layer to implement a correct matrix-vector multiplication. This partitioning is called *folding* and must be performed according to the configuration of PEs and SIMDs. The number of PEs and SIMD lanes in each layer are also called the *folding parameters* of the layer.

To demonstrate the effect of folding on latency and throughput, let us consider a simple example. Figure 2(b) shows two layers, L_0 and L_1, of a sample fully-connected binary network. Remember that each PE in the i^{th} layer can process S_i IOPs in one clock cycle. Also, in each layer multiple neurons can be served in parallel depending on the number of available PEs. If $P_1 = 2$ (number of PEs in L_1) and $S_1 = 3$ (number of SIMD lanes for each PE in L_1), as shown in Figure 2(e), then the throughput of L_1 will be 6 IOPs every clock cycle. If the number of SIMD lanes on each PE is reduced to $S_1 = 1$ as in Figure 2(d), then the throughput of L_1 is reduced to 2 IOPs every clock cycle. As another example, if $P_1 = 1$ and $S_1 = 1$ as in Figure 2(c), the throughout of the layer is reduced to just 1 IOP every clock cycle. In general, the latency lat_i of the i^{th} layer in clock cycles is *inversely proportional* to the product $P_i \cdot S_i$. Given the i^{th} layer, the relationship between its latency (in clock cycles), the number of IOPs per layer, and the folding parameters of the layer can be formalized as follows. Note that the product of P_i and S_i denotes the number of IOPs that can be processed in parallel by the i^{th} layer. Hence, as each IOP takes one clock cycle, the latency introduced by the i^{th} layer (either a convolutional or fully connected layer) is given by

$$lat_i = \left\lceil \frac{\#\text{IOPs}_i}{(P_i \cdot S_i)} \right\rceil. \quad (3)$$

The additional computations required to compute the output of each neuron after processing the synapses is already accounted within the per-IOP clock cycle due to internal pipelining of MVTUs.

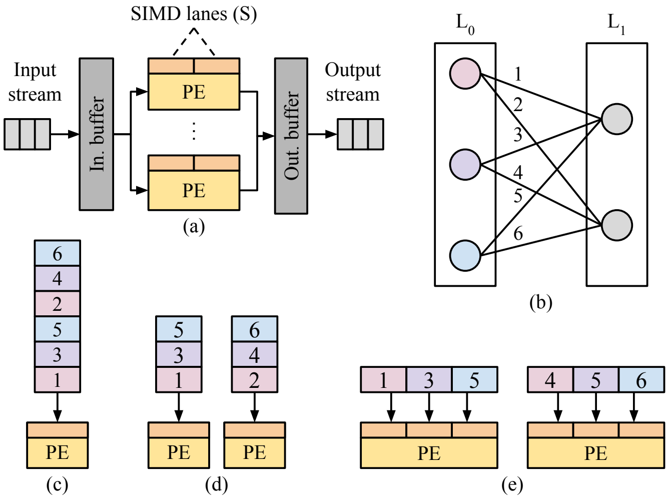


Fig. 2. Inset (a) illustrates an MVTU, while inset (b) shows an example two-layer network. Insets (c), (d), and (e) illustrate three different implementations of layer L_1.

Due to the pipelined architecture of FINN accelerators, the total throughput of the network depends on the layer with the highest latency [10]. Let N be the total number of layers in the network. The maximum latency among all layers is given by

$$lat_{\text{max}} = \max\{lat_1, \ldots, lat_N\}, \quad (4)$$

while the total latency to perform the inference of one input image is given by $lat_{\text{tot}} = \sum_{i=1}^{N} lat_i$.

To perform the inference of a batch of B images, the total throughput th of the network can finally be computed as

$$th = \frac{B}{(B-1) \cdot lat_{\text{max}} + lat_{\text{tot}}}. \quad (5)$$

As it can be observed from the above equation, maximizing the throughput of the network requires minimizing the latency of the slowest layer, i.e., minimizing lat_{max}. This is equivalent to increasing the number of PEs and SIMDs in the slowest layer at cost of a larger demand of FPGA resources.

Demand of FPGA resources. The range of variation of the folding parameters (PEs and SIMDs) that control the

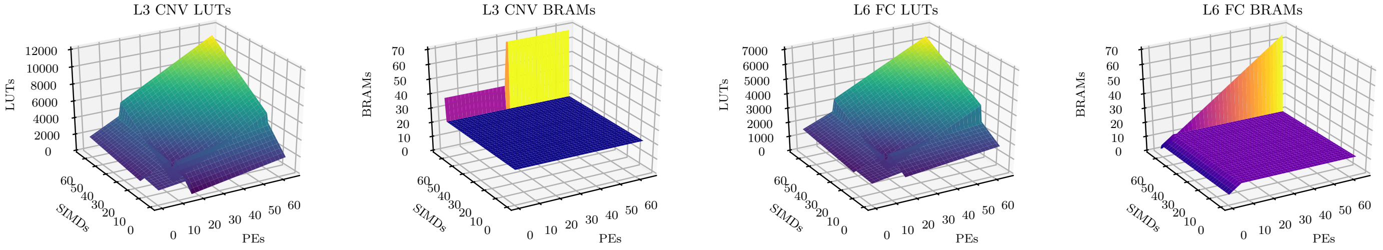


Fig. 3. Plots of functions $f_{i,t}(P_i, S_i)$ for some representative layers and resource types (see the captions above the plots).

throughput of the network is limited by the total number of available resources in the area of the FPGA where the BNN will be deployed. This can be mathematically stated as follows. Let t denote the *type* of a resource in the FPGA fabric where $t \in$ {CLB, BRAM, DSP, FF}. Furthermore, let R_t denote the total number of resources of type t in the FPGA fabric, and let $c_{i,t}$ denote the number of resources of type t demanded by the i^{th} layer. To be feasibly deployed on a given FPGA fabric, a network configuration must satisfy the following necessary condition: $\forall t \in \{\text{CLB, BRAM, DSP, FF}\}, \quad \sum_{i=1}^{N} c_{i,t} \leq R_t$. Note that, in each layer, $c_{i,t}$ depends on parameters P_i and S_i (the larger their values the larger the resource demand). This means that for every resource type t, it is possible to derive a function $c_{i,t} = f_{i,t}(P_i, S_i)$ to characterize the dependency between parameters P_i and S_i, and the resource demand of each layer. In this work, the dependency between the resource demand and the parameters P_i and S_i has been experimentally derived for each layer by **(i)** varying the two parameters, **(ii)** automatically synthesizing the resulting configuration of the FINN accelerator, and **(iii)** finally profiling the corresponding resource demand. Technical details on how this experimental data has been obtained are reported in Section 3.1.

To enable the encoding of such functions in a mathematical optimization framework (see next section), the collected data has then been fitted to piece-wise linear functions with four pieces of the following form. The demand for resources of type t by the i^{th} layer is expressed as:

$$f_{i,t}(P_i, S_i) = \begin{cases} f^1_{i,t}(P_i, S_i) & \text{if } P_i \leq \text{PE}^t_{th} \wedge S_i \leq \text{SIMD}^t_{th} \\ f^2_{i,t}(P_i, S_i) & \text{if } P_i > \text{PE}^t_{th} \wedge S_i \leq \text{SIMD}^t_{th} \\ f^3_{i,t}(P_i, S_i) & \text{if } P_i \leq \text{PE}^t_{th} \wedge S_i > \text{SIMD}^t_{th} \\ f^4_{i,t}(P_i, S_i) & \text{if } P_i > \text{PE}^t_{th} \wedge S_i > \text{SIMD}^t_{th} \end{cases} \tag{6}$$

where PE^t_{th} and SIMD^t_{th} are constant thresholds to determine the various pieces $f^1_{i,t}(P_i, S_i), \ldots, f^4_{i,t}(P_i, S_i)$ of each t type of resource. Each piece $f^p_{i,t}(P_i, S_i)$ is expressed as $f^p_{i,t}(P_i, S_i) = \xi^p_{i,t} \cdot P_i + \varrho^p_{i,t} \cdot S_i + \upsilon^p_{i,t}$, where $\xi^p_{i,t}$, $\varrho^p_{i,t}$, and $\upsilon^p_{i,t}$ are coefficients empirically determined by fitting experimental data as explained above.

3.1 Profiling FINN accelerators

As anticipated above, the resource demand and timing performance of the layers of the two considered networks, i.e., CNVW1A1 and CNVW2A2, have been profiled by varying the folding parameters. For each network, the profiling was performed by disabling all the layers in the network, except the one that was profiled, by modifying the C++ HLS code of the accelerator. The HLS synthesis of each configuration and the updating of the folding parameters was automated using a bash script. After the completion of each HLS synthesis, the script was also responsible for synthesizing the generated RTL in Vivado and logging the synthesis reports. The profiling of the layers (HLS synthesis + RTL synthesis) was performed using Vivado 2018.3 running on a machine with Ubuntu Linux 18.04, and equipped with 26 Intel Xeon cores @2.20 GHz and 132 GB of RAM. Then, for each network, a piece-wise linear regression of the logged data was performed with Matlab to build the functions of Equation (6). Note that the resource consumption model in Equation (6) also accounts for the resources utilized for storing intermediate data, i.e., the on-chip streaming FIFO buffers. The plot of the obtained resource models for some representative layers are reported in Figure 3, while the *mean average percentage error* (MAPE) of the model for each type of network is reported in Table 1. The latency of each layer of the networks has also been profiled with the purpose of validating Equations (1)-(5).

TABLE 1
Mean average percentage error (MAPE) of the resource estimation model $f_{i,t}(P_i, S_i)$ obtained via linear regression.

	MAPE (%)		
Network	LUT	BRAM	FF
CNVW1A1	4.85	2.99	4.2
CNVW2A2	5.11	1.88	5.7

3.2 Beyond FINN

It is worth noticing that, despite the analysis in this section is focused on the VGG-inspired quantized accelerators from the FINN framework, other types of DNN accelerators composed of regularly stacked convolutional and/or fully connected layers, such as AlexNet [16], YOLO [17], SEGNet [18], etc., can also be modeled using the same approach, provided that they have a pipelined hardware implementation with scalable computational units in each layer. In fact, to make the modeling applicable for such networks, we only need to modify Equation (3) as:

$$lat_i = \left\lceil \frac{\#\text{IOPs}_i \cdot n}{(P_i \cdot S_i)} \right\rceil . \tag{7}$$

where n denotes the number of clock cycles per IOP.

4 Problem Definition

This paper proposes a method to *optimize* the timing performance and resource consumption of FINN accelerators under constrained FPGA resources. As it emerges from the model presented in the previous section, the throughput and resource consumption of a FINN accelerator vary as a function of the folding parameters (P_i and S_i) of each layer. Clearly, under constrained FPGA resources, configuring such parameters requires facing with a throughput vs. area trade-off.

A possible solution to optimize a FINN accelerator consists in configuring the folding parameters of each layer such that **(i)** the throughput is maximized, and **(ii)** the utilization of the FPGA resources is maximized (provided that a synthesis of the resulting accelerator is actually possible). This optimization problem is solved by this work as a special case of a more general approach (see the next sections) but, unfortunately, the implied solution is likely to have evident limits. Indeed, note that FINN accelerators must include memories to store the parameters of the neurons (weights, biases, and thresholds) and comprise accessory logic that is independent of the folding parameters. Hence, even setting the lowest possible values for parameters P_i and S_i, a FINN accelerator is characterized by a considerable resource demand. For instance, with a target clock frequency of 100MHz for the FPGA, by setting $P_i = 1$ and $S_i = 1$ for each layer, the post-synthesis resource requirement of the CNVW1A1 network amounts to 2358 LUTs, 92 BRAMs, and 3145 FFs (obtained with Vivado 2018.3). This resource demand may not be satisfiable, especially on resource-constrained FPGAs or when the area is used to deploy other modules, e.g., other FINN neural network accelerators.

To address this issue, this work proposes to opportunistically split the network pipeline into *chunks* that are dynamically configured at run-time by means of DPR. For instance, if an accelerator is split into two chunks, e.g., as it is illustrated in Figure 4, the inference process works by **(i)** configuring and executing the first chunk, and then **(ii)** reconfiguring the FPGA with the second chunk and execute the latter. This approach allows working with accelerators that require less resources than the minimal ones demanded by a stock FINN accelerator, hence enabling the deployment of FINN accelerators in platforms in which *it would be impossible*. Clearly, this comes at the cost of larger latencies due to reconfiguration times.

4.1 The optimization problem addressed in this work

Optimizing FINN accelerators under the more general approach based on DPR requires facing with additional challenges: deciding *how many* chunks an accelerator has to be split and *where* it has to be split. Contextually, the folding parameters of the layers must be optimized to better exploit the available FPGA area on which the chunks are configured, with the end of maximizing the throughput (taking reconfiguration times into account). To address this challenge, the following section proposes a formulation of the problem as a *mixed-integer linear program* (MILP). Instead of developing a custom heuristic algorithm to solve the problem, a MILP formulation allows leveraging well-established and powerful optimization algorithms developed by experts of optimization (mature commercial solvers such as CPLEX and Gurobi are available, while open-source solutions such as GLPK are also effective, both with APIs for popular programming languages such as C and C++). Furthermore, MILP formulations come in a form that is modular, hence easier to extend, validate, and maintain.

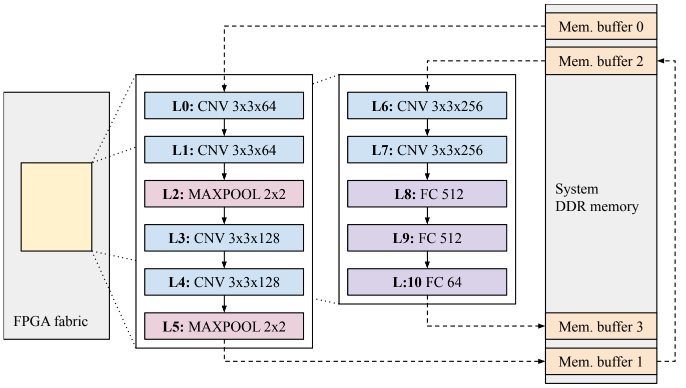


Fig. 4. Block diagram illustrating the execution flow of a network split into two chunks.

4.2 Preliminary considerations

Before presenting our solution, it is necessary to discuss a few aspects related to the timing performance of FINN accelerators. In stock implementation of FINN accelerators [15], the parameters of the neurons are loaded by the software support into the hardware accelerator from DRAM memory at run-time. The software is also responsible for pre-processing a batch of input images for classification as well as the result of the classification when the inference is completed. Therefore, the total execution time of the inference is composed of three phases: (i) loading of neuron parameters from memory, (ii) image pre-processing, and (iii) hardware inference. For example, by profiling the stock configuration of the CNVW1A1 network running on the PYNQ-Z1 platform by Xilinx and operating on a batch of 256 CIFAR-10 images, we found that the loading of the parameters of the neurons can take up to 450.3 ms, while the longest observed time to perform the inference took about 85.8 ms. We have also found that, despite its low memory footprint, loading the parameters of the FINN accelerators takes a large portion of the total execution time due to unoptimized memory transfers. However, in the case in which a FINN accelerator is configured only once, and then used for multiple inferences, such overhead may be acceptable as it corresponds to an initialization phase of the accelerator that has to be done only once at the system startup. Conversely, it becomes a severe disadvantage if the FPGA area is subject to DPR as proposed by our approach. Indeed, every time an area hosting a chunk is reconfigured, all the neuron parameters would be lost.

To address this issue, the stock FINN accelerators of both networks have been modified to embed the neuron parameters directly into the accelerator during the generation of the bitstream. In this way, the loading of the parameters can be skipped, and, once configured on the fabric, the accelerator is ready to run. Despite this reduces flexibility, as a new accelerator has to be synthesized every time the network parameters are changed, we argue that it does not represent a critical limitation at the stage of deployment, when the network is likely not to change unless a major update of the system is performed. We have also experimentally confirmed that embedding the parameters into the accelerators has a negligible effect on the reconfiguration time, since the total memory footprint of the parameters of the networks is in the order of a few kilobytes, while the size of the partial-bitstream is usually in the order of a few megabytes. Under this choice, the time required to perform the inference is given by the sum of the times to execute the chunks and the corresponding accesses to memory. Both these times are related to the inputs and outputs of the network, and the ones required to write and read the intermediate data that flow on the cut between the chunks. The input to the network are 32x32 RGB images, each amounting to 3 KB, while the output of the network comprises 64 16-bit values, for a total of 128 bytes. On a PYNQ-Z1, using the high-performance memory ports and on for a batch of 256 images, the total memory transfers for such inputs and outputs require less than 1 ms. Since the reconfiguration and inference times are in the order of a few tens of milliseconds, the time required by such memory transfers tend to be negligible. The size of the intermediate data that are written and read from memory is proportional to the size of the feature map at the split point. An analysis of FINN accelerators revealed that the largest feature map has a size of 57600 bits (it corresponds to the link between the first and the second layers). The corresponding memory access times amount to about 16 μs and are hence also negligible. Overall, our profiling revealed that the inference time can be effectively approximated as the sum of the execution times of the chunks plus their reconfiguration times. For instance, the inference time with an accelerator split into two chunks can be expressed as $T_{\text{split}} = T_{\text{acc1}} + T_{\text{acc2}} + 2T_{\text{reconf}}$, where T_{acc1} and T_{acc2} are the execution times of the chunk, and T_{reconf} is the reconfiguration time of the FPGA slot in which the two chunks are programmed.

5 MILP FORMULATION

This section presents the proposed MILP formulation. The *inputs* of the optimization problem are the structure of the BNN, the functions that model its resource demand presented in Section 3, and the amount of available FPGA resources. The MILP variables and the constraints that define the optimization problem are then presented as a function of such inputs. The constraints are categorized into (i) general constraints, (ii) resource constraints, and (iii) latency constraints. The *outputs* of the optimization problem are the values of the folding parameters for each layer and the chunks into which the BNN accelerator is split. The objective is to maximize the throughput.

5.1 Optimization variables

To begin, note that the folding parameters P_i and S_i cannot take arbitrary values, but rather only powers of two (this is imposed by FINN). This observation is useful to efficiently formulate the optimization problem. That is, instead of directly modeling parameters P_i and S_i as a pair of integer variables for each layer, we adopt a set of binary variables to identify the powers of two of the corresponding parameter value.

The largest value allowed for parameters P_i and S_i is $2^{M+1} = 64$ (with $M = 5$). Hence, for each layer i and $\forall m = 1, \ldots, M$, we define: **(i)** $\beta_{i,m} \in \{0, 1\}$, a binary variable such that $\beta_{i,m} = 1$ *if and only if* $P_i = 2^{m+1}$; and **(ii)** $\delta_{i,m} \in \{0, 1\}$, a binary variable such that $\delta_{i,m} = 1$ *if and only if* $S_i = 2^{m+1}$.

For each layer i, we also define $\tau_i \in \mathbb{Z}_{\geq 1}$ as an integer variable that represents the product of the folding parameters in the i^{th} layer i.e., $\tau_i = P_i \cdot S_i$, which is particularly useful to deal with the latency of the layer (see Eq. (3)). Consequently, for each layer i and $\forall m = 1, \ldots, 2M + 1$, we define a binary variable $\lambda_{i,m} \in \{0, 1\}$ such that $\lambda_{i,m} = 1$ *if and only if* $\tau_i = P_i \cdot S_i = 2^{m+1}$.

Other variables are required to encode the cuts and the grouping of layers in chunks. To this purpose, for each layer i we define a binary variable $x_i \in \{0, 1\}$ such that $x_i = 1$ *if and only if* there is a cut between the i^{th} layer and the $(i + 1)^{\text{th}}$ layer (for consistency, we implicitly set $x_N = 0$ as no cuts are possible after the last layer). Note that the maximum number of chunks that can be obtained by optimization cannot be larger than the total number N of layers, i.e., in the limit case, each chunk includes just one layer. Hence, also the number of chunks is bounded by N. This observation allows defining the following variables. For each chunk j we define **(i)** a binary variable $y_{i,j} \in \{0, 1\}$ for each layer i, such that $y_{i,j} = 1$ *if and only if* the i^{th} layer belongs to the j^{th} chunk; and **(ii)** two real variables $\Phi_j^{max} \in \mathbb{R}_{\geq 0}$ and $\Phi_j^{tot} \in \mathbb{R}_{\geq 0}$ that model the maximum and the total latency of the chunk, respectively.

Some of the following constraints make use of a numerical constant $\mathcal{M}$ that denotes a very large positive number to represent *infinity* (in our implementation we set $\mathcal{M} = 10^9$).

5.2 General constraints

A set of constraints are required to impose certain restrictions on the values taken by the optimization variables in order to guarantee the consistency of their definition.

First of all, we define a simple constraint to limit the upper bounds of P_i and S_i. As stated above, the folding parameters can only take values in $\{1, 2, 4, 8, 16, 32, 64\}$. Hence, to enforce the consistency of variables $\beta_{i,m}$ and $\delta_{i,m}$, the following constraint is provided:

Constraint 1. $\forall i = 1, \ldots, N$

$$P_i = \sum_{m=1}^{M} 2^{(m+1)} \cdot \beta_{i,m} \quad \wedge \quad \sum_{m=1}^{M} \beta_{i,m} = 1 \tag{8}$$

$$S_i = \sum_{m=1}^{M} 2^{(m+1)} \cdot \delta_{i,m} \quad \wedge \quad \sum_{m=1}^{M} \delta_{i,m} = 1 \tag{9}$$

In Eq. (8), the first term connects variable $\beta_{i,m}$ to P_i, while the second term enforces the fact that P_i can have one and at most one of the powers of two as its value. That is, in the range $m = 1, \ldots, M$, variables $\beta_{i,m}$ are always 0 except once. The same holds for Eq. (9) with respect to variables $\delta_{i,m}$ and S_i.

Remember that τ_i was defined as the product of the folding parameters of the i^{th} layer, i.e., $\tau_i = P_i \cdot S_i$. This definition is enforced by the following constraint:

Constraint 2. $\forall i = 1, \dots, N, \qquad \forall m = 1, \dots, M$

$$\begin{aligned} \tau_i &\geq 2^{(m+1)} \cdot S_i - (1 - \beta_{i,m}) \cdot \mathcal{M}, \\ \tau_i &\leq 2^{(m+1)} \cdot S_i + (1 - \beta_{i,m}) \cdot \mathcal{M}. \end{aligned} \tag{10}$$

Constraint 2 is an efficient encoding of the product of P_i and S_i as MILP formulation allows linear constraints only, i.e., they cannot contain products between optimization variables. Its rationale is the following. For each value of m for which $\beta_{i,m} = 0$, the constraint reduces to $\tau_i \geq -\infty$ and $\tau_i \leq \infty$, and hence enforces no restrictions. For the *only* value of m such that $\beta_{i,m} = 1$, it holds $P_i = 2^{(m+1)}$, and hence the constraint reduces to $\tau_i \geq P_i \cdot S_i$ and $\tau_i \leq P_i \cdot S_i$, which is equivalent to $\tau_i = P_i \cdot S_i$.

The consistency of variables τ_i and $\lambda_{i,m}$ can be enforced exactly as done for Constraint 1.

According to the definition of chunks, each layer must be a member of *one and only one* chunk. With respect to the optimization variables, this is equivalent to say that each layer i has variable $y_{i,j} = 1$ only for one j. This is enforced using the following constraint:

Constraint 3. $\forall i = 1, \dots, N, \quad \sum_{j=1}^{N} y_{i,j} = 1.$

Unless there is a cut between the i^{th} layer and the $(i+1)^{\text{th}}$ layer, i.e., $x_i = 1$, the two (consecutive) layers must always belong to the same chunk, i.e., $y_{i,j} = y_{i+1,j}$ only for the same j. The following constraint enforces this simple property:

Constraint 4. $\forall i = 1, \dots, N-1, \quad \forall j = 1, \dots, N$

$$\begin{aligned} y_{i,j} &\leq y_{i+1,j} + x_i \cdot \mathcal{M}, \\ y_{i+1,j} &\leq y_{i,j} + x_i \cdot \mathcal{M}. \end{aligned} \tag{11}$$

The rational behind Constraint 4 is the following. When $x_i = 0$, i.e., when there is no cut between the i^{th} layer and the $(i+1)^{\text{th}}$ layer, Constraint 4 reduces to $y_{i,j} \leq y_{i+1,j}$ and $y_{i+1,j} \leq y_{i,j}$, which is equivalent to $y_{i+1,j} = y_{i,j}$. This correctly forces the two layers to belong to the same chunk. Conversely, when $x_i = 1$, i.e., when there is a cut between the i^{th} layer and the $(i+1)^{\text{th}}$ layer, the constraint reduces to $y_{i,j} \leq \infty$ and $y_{i+1,j} \leq \infty$, hence posing no restrictions.

Meanwhile, if there is a cut between the i^{th} layer and the $(i+1)^{\text{th}}$ layer, then the two consecutive layers should never be in the same chunk. The following constraint is used to enforce this property:

Constraint 5. $\forall j = 1, \dots, N, \qquad \forall i = 1, \dots, N-1,$

$$y_{i,j} + y_{i+1,j} \leq 1 + (1 - x_i) \cdot \mathcal{M}. \tag{12}$$

If $x_i = 1$, i.e., there is a cut between the i^{th} layer and the $(i+1)^{\text{th}}$ layer, then the above constraint is reduced to $y_{i,j} + y_{i+1,j} \leq 1$. This effectively prohibits the case in which both the layer i and $i{+}1$ belong to the j^{th} chunk. Conversely, if $x_i = 0$, i.e., there is no cut between the i^{th} layer and the $(i+1)^{\text{th}}$ layer, then the constraint reduces to $y_{i,j} + y_{i+1,j} \leq \infty$, hence posing no restrictions.

It should be noted that Constraint 4 and Constraint 5 are not sufficient to guarantee a correct allocation of the layers into chunk. Indeed, we still need some constraint to prevent different chunks from ending up being modeled by the same index j, i.e., to avoid that layers that belong to different non-adjacent chunks have variables $y_{i,j}$ set for the same j. Such a restriction can be achieved by noting that, once there is a cut along the network pipeline, a layer on the left cannot belong to any of the chunks hosting the layers on the right. This property is enforced as follows:

Constraint 6. $\forall j = 1, \dots, N, \qquad \forall i = 1, \dots, N,$

$$\sum_{k=i+1}^{N} y_{k,j} \leq (1 - x_i) \cdot \mathcal{M} + (1 - y_{i,j}) \cdot \mathcal{M}. \tag{13}$$

Constraint 6 is based on two conditions: (i) there is a cut between the i^{th} layer and the $(i+1)^{\text{th}}$ layer ($x_i = 1$), and (ii) the layer to the left of the cut, i.e., the i^{th} layer, belongs to the j^{th} chunk ($y_{i,j} = 1$). If both these conditions are true, then the constraint is reduced to $\sum_{k=i+1}^{N} y_{k,j} \leq 0$, effectively asserting that all the layers to the right of the cut cannot be members of the j^{th} chunk, i.e., $\forall k = i+1, \dots, N, \; y_{k,j} = 0$. If any of the two conditions is false ($x_i = 1$ and/or $y_{i,j} = 1$), the constraint is reduced to $\sum_{k=i+1}^{N} y_{k,j} \leq \infty$, hence posing no restrictions.

5.3 Resource constraints

This section presents the constraints required to enforce that the amount of resources demanded by each chunk does not exceed the available resources on the FPGA fabric (or either a given slot on the fabric). As discussed in Section 2.2, the resource demand of each layer depends on the folding parameters P_i and S_i.

The amount of resources $c_{i,t}$ of type t demanded by the i^{th} layer is given by the piece-wise linear functions $f_{i,t}(P_i, S_i)$ in Equation (6). Here, the challenge consists in efficiently encoding such functions in the MILP formulation. To this end, we define two auxiliary binary variables, $\alpha_i^t \in \{0, 1\}$ and $\gamma_i^t \in \{0, 1\}$, whose value will be determined by P_i and S_i, respectively, and whose combination will be used to determine which of the four pieces of Equation (6) is used by a given pair of the folding parameters. In essence, variables α_i^t and γ_i^t are used to achieve a binary encoding of the index of the four pieces, i.e., 00, 01, 10, or 11. The following constraint is hence enforced:

Constraint 7. $\forall i = 1, \dots, N, \forall t \in \{\text{CLB, BRAM, DSP, FF}\}$

$$\begin{aligned} c_{i,t} &\geq f_{i,t}^1(P_i, S_i) - (\alpha_i^t + \gamma_i^t) \cdot \mathcal{M} \\ c_{i,t} &\geq f_{i,t}^2(P_i, S_i) - (1 - \alpha_i^t + \gamma_i^t) \cdot \mathcal{M} \\ c_{i,t} &\geq f_{i,t}^3(P_i, S_i) - (1 + \alpha_i^t - \gamma_i^t) \cdot \mathcal{M} \\ c_{i,t} &\geq f_{i,t}^4(P_i, S_i) - (2 - \alpha_i^t - \gamma_i^t) \cdot \mathcal{M} \end{aligned} \tag{14}$$

For example, if γ_i^t = 1 and α_i^t = 0, all sub-constraints will have no effect on $c_{i,t}$ except the third one, which is equivalent to the third piece in Equation (6).

The problem is now how to assign suitable values to the auxiliary variables α_i^t and γ_i^t such that the ranges of each piece is reflected. In other words, their values must be constrained in such a way that $\alpha_i^t = 1$ *if and only if* $P_i > \text{PE}_{th}^t$, and similarly $\gamma_i^t = 1$ *if and only if* $S_i > \text{SIMD}_{th}^t$. In this way, all four combinations of the values of the auxiliary variables can reflect the ranges of the four pieces as in Equation (6). This is achieved by means of the following auxiliary constraint.

Constraint 8. $\forall i = 1\ldots, N, \forall t \in \{\text{CLB, BRAM, DSP, FF}\}$

$$P_i - \epsilon \geq \text{PE}^t_{th} - \mathcal{M}\cdot(1-\alpha^t_i) \quad \wedge \quad P_i - \epsilon \leq \text{PE}^t_{th} + \mathcal{M}\cdot\alpha^t_i \tag{15}$$

$$S_i - \epsilon \geq \text{SIMD}^t_{th} - \mathcal{M}\cdot(1-\gamma^t_i) \quad \wedge \quad S_i - \epsilon \leq \text{SIMD}^t_{th} + \mathcal{M}\cdot\gamma^t_i \tag{16}$$

where $\epsilon > 0$ is an arbitrarily-small positive number (e.g., 0.0001).

Eqs. (15) and (16) have the same form, hence it suffices to discuss one of them. There are two cases when constraining the values of α^t_i according to PE^t_{th} in Eq. (15). If $P_i > \text{PE}^t_{th}$, then $\alpha^t_i = 1$ and the equation becomes $P_i - \epsilon \geq \text{PE}^t_{th}$, which matches the definition of α^t_i, and $P_i - \epsilon \leq \infty$, which is always true. If $P_i \leq \text{PE}^t_{th}$, then $\alpha^t_i = 0$, and the equation becomes $P_i - \epsilon \geq -\infty$, which is always true, and $P_i - \epsilon \leq \text{PE}^t_{th}$, which matches the definition of α^t_i.

The total amount of resources C^t_j of type t in the j^{th} chunk is equivalent to the sum of the demand of resources of type t by all the layers that are part of the chunk. As variables $y_{i,j}$ denote whether a layer is part of a chunk or not, C^t_j can be expressed as follows: $C^t_j = \sum_{i=1}^N c^t_i \cdot y_{i,j}$. Note that this equation is not a linear and hence cannot be directly encoded in a MILP constraint. In order to linearize it, we define an auxiliary real variable $a^t_{i,j} \in \mathbb{R}_{\geq 0}$ such that $a^t_{i,j} = c^t_i \cdot y_{i,j}$ and enforce the following set of auxiliary constraints.

Constraint 9. $\forall j = 1,\ldots,N,\ \forall i = 1,\ldots,N,\ \forall t \in \{\text{CLB, BRAM, DSP, FF}\}$, $a^t_{i,j} \leq c^t_i$ and

$$a^t_{i,j} \leq y_{i,j}\cdot\mathcal{M} \quad \wedge \quad a^t_{i,j} \geq c^t_i - (1-y_{i,j})\cdot\mathcal{M} \tag{17}$$

Here, $a^t_{i,j}$ denotes the amount of resources of type t demanded by the i^{th} layer when placed in the j^{th} chunk. There are two cases to consider: $y_{i,j} = 0$ and $y_{i,j} = 1$, i.e., when the i^{th} layer is not a member of the j^{th} chunk and when it is, respectively. If $y_{i,j} = 0$, the constraint enforces $a^t_{i,j} \leq c^t_i$, $a^t_{i,j} \leq 0$ and $a^t_{i,j} \geq -\infty$, which are equivalent to $a^t_{i,j} = 0$. This effectively forces the i^{th} layer not to contribute to the resource consumption of the j^{th} chunk. Meanwhile, when $y_{i,j} = 1$, the constraint reduces to $a^t_{i,j} \leq c^t_i$, $a^t_{i,j} \leq \infty$, and $a^t_{i,j} \geq c^t_i$, which, when combined, are equivalent to $a^t_{i,j} = c^t_i$, hence quantifying the correct resource demand of the layer. In this way, C^t_j can be expressed in a linear form as just $C^t_j = \sum_{i=1}^N a^t_{i,j}$. Hence, it is finally possible to enforce the main constraint to impose that the amount of resources demanded by the chunks must not be larger than those available on the FPGA (terms R_t).

Constraint 10. $\forall j = 1,\ldots,N,\ \forall t \in \{\text{CLB, BRAM, DSP, FF}\}$ $C^t_j \leq R_t$.

5.4 Latency constraints

Thanks to the definition of variables $\lambda_{i,m}$, the latency lat_i of each layer (as given by Equation (3)) can be directly encoded as a MILP constraint as follows:

Constraint 11. $\forall i = 1,\ldots,N, \quad \forall m = 1,\ldots,2M+1$

$$\begin{aligned} lat_i &\geq \frac{\#\text{IOPs}_i}{2^{m+1}} - (1-\lambda_{i,m})\cdot\mathcal{M}, \\ lat_i &\leq \frac{\#\text{IOPs}_i}{2^{m+1}} + (1-\lambda_{i,m})\cdot\mathcal{M}. \end{aligned} \tag{18}$$

The above constraint is due to the following reason. First, by profiling the layers, we found that $\#\text{IOPs}_i$ is always a power of two, hence Equation (3) can be safely used without the ceiling operator $\lceil\ \rceil$. Second, remember that, given the i^{th} layer, only one binary variable $\lambda_{i,m}$ is set for a given m, such that $P_i \cdot S_i = 2^{m+1}$. Hence, when $\lambda_{i,m} = 1$, the constraint reduces to Equation (3). Conversely, for all the other values of m such that $\lambda_{i,m} = 0$, the constraint reduces to $lat_i \geq -\infty$ and $lat_i \leq \infty$, hence imposing no restrictions.

The total latency of the j^{th} chunk, modeled by variable Φ^{tot}_j, is the sum of the latency of each layer in the chunk, and can be expressed as $\Phi^{\text{tot}}_j = \sum_{i=1}^N y_{i,j} \cdot lat_i$. Similarly, the maximum latency among the layers in the j^{th} chunk, modeled by variable Φ^{max}_j, can be encoded by enforcing the following condition: $\forall i = 1,\ldots,N$, $\Phi^{\text{max}}_j \geq y_{i,j} \cdot lat_i$.

Note that both the latter equations are not linear as they comprise products of optimization variables. Hence, they must be linearized to be encoded in a MILP formulation. This can be accomplished as done in Constraint 9.

Thanks to the above results, it is finally possible to formulate the throughput th of the accelerator.

Constraint 12.

$$th = \frac{B}{\sum_{j=1}^N((B-1)\cdot\Phi^{\text{max}}_j + \Phi^{\text{tot}}_j) + T_{\text{reconf}}\cdot(\Gamma + \sum_{k=1}^N x_k)}. \tag{19}$$

This constraint is obtained by rewriting Equation (5) as a function of the optimization variables and by adding the additional latency originated by the reconfiguration of the chunks. Here, Γ is an auxiliary variable that denotes whether the network is cut at least once ($\Gamma = 1$) or not ($\Gamma = 0$), which is defined $\Gamma = \max_k\{x_k\}$ and expressed in an equivalent form via the following auxiliary constraint: $\forall k = 1,\ldots N, \Gamma \geq x_k$. Note that $\sum_{k=1}^N x_k$ gives the number of cuts. Hence, $(\Gamma + \sum_{k=1}^N x_k)$ gives the number of chunks. The total reconfiguration time can be computed by simply accounting a reconfiguration time T_{reconf} for each of such chunks. The computation of T_{reconf} depends on the amount of available FPGA resources and is discussed in the next section.

5.5 Objective function

The objective of the optimization problem is to maximize the throughput of the accelerator for a given batch size B. Hence, the objective function of the optimization problem is **maximze** $\{th\}$, with th being defined as in Constraint 12.

5.6 Extending the MILP formulation

Note that the MILP formulation presented in this section is not limited to the FINN accelerators only, but can also be applied to optimize the performance of other DNN accelerators with regularly stacked convolutional and/or fully connected layers as mentioned in Section 3.2. The main prerequisite is the availability of the resource and latency models of the accelerators as the ones derived for FINN in this paper. With a few additional constraints, the proposed MILP optimization can also support the optimization of networks with irregular architecture, such as GoogLeNet [19], Resnet [20] etc., using a coarse-grained approach for splitting layers into chunks. For example, we can reduce the architectural irregularity of a GoogLeNet architecture by forcing the optimizer to always put layers inside an inception unit of the network into the same chunk, i.e., by considering an inception unit as indivisible into different chunks. Similarly, we can reduce the irregularity of a ResNet-like architecture by constraining the optimizer not to split the residual blocks into different chunks. Despite both of these approaches increase the granularity of a layer and therefore reduce the search space, they extend the domain of DNN accelerator architectures that can be optimized by our approach and consequently allow the deployment of such types of networks on FPGAs with constrained resources.

6 Experimental evaluation

This section presents a set of experiments that were conducted to assess the effectiveness of the proposed optimization technique on a real hardware platform.

The experiments were conducted in two sessions. The objective of the first session is to show how the proposed approach can improve the throughput of the stock CNVW1A1 and CNVW2A2 FINN accelerators without splitting them, i.e., no DPR was used. To this end, we disabled the MILP constraints related to splitting (which is equivalent to forcing the number of chunks to 1), hence generating optimal folding parameters only. From here on, we will refer to such types of optimal configurations of the CNVW1A1 and CNVW2A2 networks as static_OPT_CNVW1A1 and static_OPT_CNVW2A2, respectively. The inference time and overall resource utilization of both static_OPT_CNVW1A1 and static_OPT_CNVW2A2 were compared against their respective stock configurations provided by FINN [15], and against heuristic (manual) optimizations, which are taken as a baseline. The latter are referred to as static_baseline_CNVW1A1 and static_baseline_CNVW2A2, and work by simply halving the folding parameters of the CNVW1A1 and CNVW2A2 networks (starting from their values in their stock configuration) until it is possible to deploy the accelerators on a given area.

In the second experimental session, we compared the timing performance of the static_OPT_CNVW1A1 and static_OPT_CNVW2A2 approaches against the cases in which the networks are split into chunks, henceforth respectively referred to as dynamic_OPT_CNVW1A1 and dynamic_OPT_CNVW2A2, under different amounts of FPGA resources. The objective of the second experimental session is specially targeted at determining the optimal settings, i.e., both the folding parameter configurations and the chunk splitting, when the networks are to be deployed on FPGAs with constrained resources.

Experimental setup. The experimental evaluation has been performed using the PYNQ-Z1 board as the reference platform. The PYNQ board is built around the Zynq-7020 SoC from Xilinx, which includes a dual-core Cortex-A9 processor coupled with an Artix-7 family FPGA fabric. The FPGA fabric of the Zynq-7020 comprises R_{CLB} = 53200 CLBs, R_{BRAM} = 140 BRAMs, R_{DSP} = 220 DSPs, and R_{FF} = 106400 FFs. The SoC is connected to a shared DRAM memory of size 512MB. This evaluation uses an implementation of the CNVW1A1 network [15] trained with the CIFAR-10 dataset provided by FINN. The network has been configured to process a batch of $B = 256$ images on each run. The hardware inference process performed by the accelerator is supported by a software support layer written in Python and C++. The MILP-based optimization has been implemented using Gurobi v. 7.0.2 via its C++ API. The different MILP optimization strategies and techniques inside the solver, such as presolve, cutting planes, heuristics, etc., were set to their default parameters. Meanwhile, the integer feasibility tolerance of the solver was set to $1\text{e}{-9}$ and the optimization timeout was set to 1800 seconds. The optimizer was executed on a machine with an 8-core Intel i7 CPU @4.5 GHz. In all the experiments, the accelerators are implemented to run at 100 MHz.

6.1 Results

The notion of *FPGA scaling factor* is introduced to study the cases in which only a fraction of the total FPGA resources are available. We say that the amounts of FPGA resources are scaled by $\sigma \in [0, 1]$ (also expressed as a percentage in the following) if the resource availability for each resource of type $t \in$ {CLB, BRAM, DSP, FF} is $\lfloor \sigma \times R_t \rfloor$. The FPGA scaling factor is also correlated to the FPGA area subject to partial-reconfiguration, hence it can be used to model the reconfiguration time. As a preliminary experiment, the reconfiguration time of the FPGA fabric has been profiled as a function of the FPGA scaling factor σ: the profiling revealed that the reconfiguration time can be safely modeled as a linear function $T_{\text{reconf}}(\sigma) = 48087 \times \sigma + 951\ \mu s$. This linear model is used in Constraint 12.

First experimental session. The stock accelerators for the CNVW1A1 and CNVW2A2 networks have a total LUT, BRAM, and FF utilization that amounted to {37%, 87%, 27%} and {68.1%, 100%, 48.7%} of the total resources available on the Zynq-7020, respectively, and an inference time of 85.8 ms and 86.34 ms, respectively, on a batch of 256 images. Note that the stock accelerators are characterized by an evident asymmetry in resource utilization, as BRAMs are highly utilized in both accelerators, while other types of resources are utilized less. This asymmetry, which is caused by the sub-optimal configuration of the folding parameters in each layer, limits the throughput of the accelerators. In particular, we found that FINN accelerators tend to consume a lot of BRAMs, while underutilizing the other types of FPGA resources. With our approach, it is possible (via static_OPT_CNVW1A1 and static_OPT_CNVW2A2) to produce optimal accelerators with maximal throughput that can be deployed on the same area: this is achieved thanks to the optimal configuration of the folding parameters, which enables a more balanced utilization of the resources.

For instance, by scaling the amount of FPGA resources to the 87% (the maximum of the three per-resource-type utilizations of the stock configuration of the CNVW1A1 network), the static_OPT_CNVW1A1 approach produced a configuration with optimal throughput that required the 85.8%, 85.9%, and 81.6% of LUTs, BRAMs, and FFs, respectively. The corresponding inference time of the accelerator is reduced to 18.60 ms, which corresponds to an improvement of more than 4x. Similarly, by scaling the amount of FPGA resources to 100% (the maximum of the three per-resource-type utilizations of the stock configuration of the CNVW2A2 network) the inference time of the static_OPT_CNVW2A2 approach was reduced to 48.75 ms, which corresponds to an improvement of 1.7x. In this case, the static_OPT_CNVW2A2 utilized 95.6%, 100% and 76.8% of LUTs, BRAMs and FFs, respectively. The same strategy has been applied for FPGA scaling factors in $\{0.3, 0.35, 0.45, \ldots, 1.0\}$: the results are reported in Figure 5. No data were reported in the case in which it was not possible to produce a feasible configuration (i.e., below a scaling factor of 70%).

To also provide a taste of what one can achieve with an empirical optimization of the folding parameters of the CNVW1A1 and CNVW2A2 networks, the results of the static_baseline_CNVW1A1 and static_baseline_CNVW2A2 approaches are also reported in Figure 5, by halving five times all folding parameters of the stock configuration of each network. As it can be observed from the figure, the results clearly show the benefits of using an optimization-based approach for both networks rather than a heuristic configuration. Figure 6(a) shows the LUT utilizations under different BRAM utilizations for static_OPT_CNVW1A1 and static_baseline_CNVW1A1 approaches, while Figure 6(b) shows the same comparison for static_OPT_CNVW2A2 and static_baseline_CNVW2A2 approaches. In both cases, the optimized networks, i.e., static_OPT_CNVW1A1 and static_OPT_CNVW2A2, have demonstrated a balanced resource utilization against their respective unoptimized counterparts. For instance, in Figure 6(a), for a BRAM utilization (on the x-axis) of 73.5%, the solution produced by static_baseline_CNVW1A1 required 30% of the LUTs, while 72.9% of the LUTs are required by the one generated by static_OPT_CNVW1A1. Such a difference in resource utilization provides a large difference in inference time: as it can be seen from Figure 5, the accelerator generated by static_baseline_CNVW1A1 requires 4745 ms, while the one generated by static_OPT_CNVW1A1 requires just 73.94 ms. In a similar manner, in Figure 6(b), a 79% BRAM utilization in static_baseline_CNVW2A2 results in a 56% LUT utilization, whereas the LUT utilization in static_OPT_CNVW2A2 is 76%. Also in this case, the resource utilization asymmetry leads to a difference in inference time: from Figure 5, the accelerator generated by static_baseline_CNVW2A2 requires 6125 ms, while the one generated by static_OPT_CNVW2A2 requires 107 ms.

Second experimental session. In this session we performed two sets of comparisons: in the first set we compared static_OPT_CNVW1A1 with dynamic_OPT_CNVW1A1, while in the second set, the static_OPT_CNVW2A2 was compared with dynamic_OPT_CNVW2A2 for the same FPGA scaling factors tested in the first session. The results of these comparisons are reported in Figure 5. In the tested cases for both sets, a static configuration (whenever it is possible) is preferable to chunk splitting. However, as it can be seen from the figure, below a scaling factor of 70% for the first set and 85% for the second set, no static approach was capable of generating a feasible configuration. This means that it is not simply possible to statically deploy the networks under those area constraints. Conversely, dynamic_OPT_CNVW1A1 is capable of producing feasible accelerators for FPGA scaling factors as low as 30%, while also maintaining a pretty much stable inference time, whereas the inference time of accelerators produced using the dynamic_OPT_CNVW2A2 approach linearly increases by decreasing the FPGA scaling factor. It was no longer feasible to produce accelerators for dynamic_OPT_CNVW2A2 below an FPGA scaling factor of 45%. Figure 7 also shows the breakdown of cumulative actual inference and reconfiguration times (i.e., coping with the contribution of all chunks) under dynamic_OPT_CNVW1A1 and dynamic_OPT_CNVW2A2. One should note that, despite more chunks may be needed as the FPGA area reduces, the reconfiguration time of each chunk becomes shorter. Also, note that the actual inference time can be reduced under dynamic_OPT_CNVW1A1 and dynamic_OPT_CNVW2A2 as the largest per-layer latency and the latency of the slowest layer can become shorter by splitting the networks (since more computational resources will be available for the layers when splitting), while this may not possible under a static allocation (mainly due to the large granularity with which the folding parameters can be varied). To clarify this point, consider the case of FPGA scaling factors 0.6 and 0.7 in Figure 7(a). Note that the inference time is significantly reduced when the scaling factor is 0.6 compared to the one when it is 0.7, mainly due to the fact that the network has been split into two chunks. Nevertheless, the total execution time is still large because of the reconfiguration overhead (two chunks must be reconfigured instead of just one). In all the tested cases, the time required to solve the MILP formulation was never larger than 30 seconds.

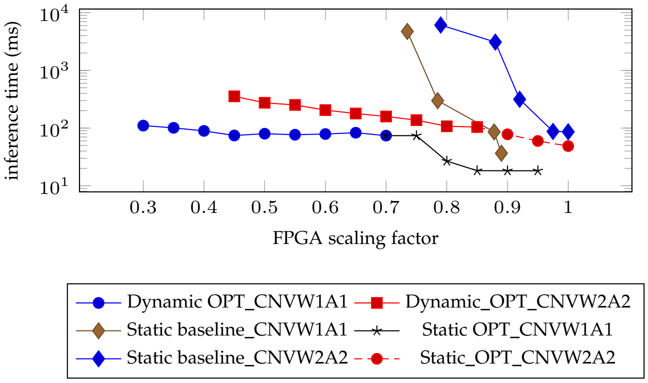
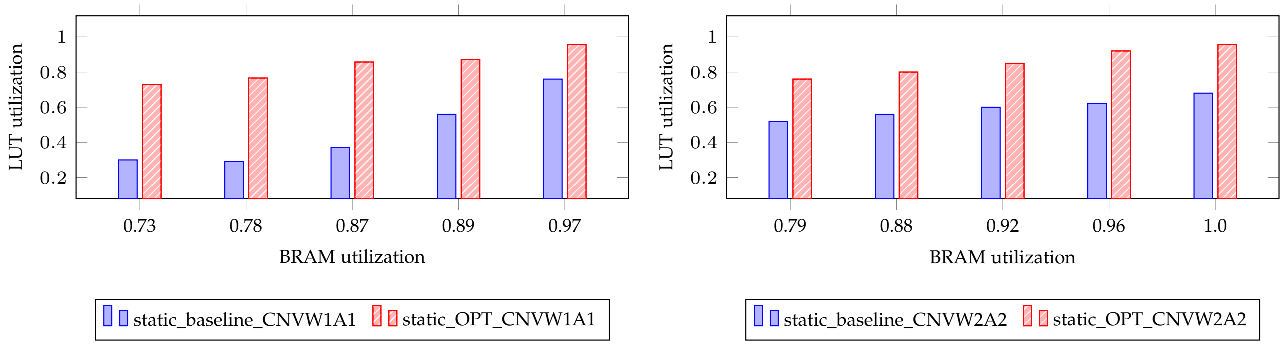


Fig. 5. Comparison of inference times (which imply throughput) as a function of the maximum percentage of FPGA resources for different configurations of the accelerators obtained by the three approaches studied in the paper.

7 Related Work

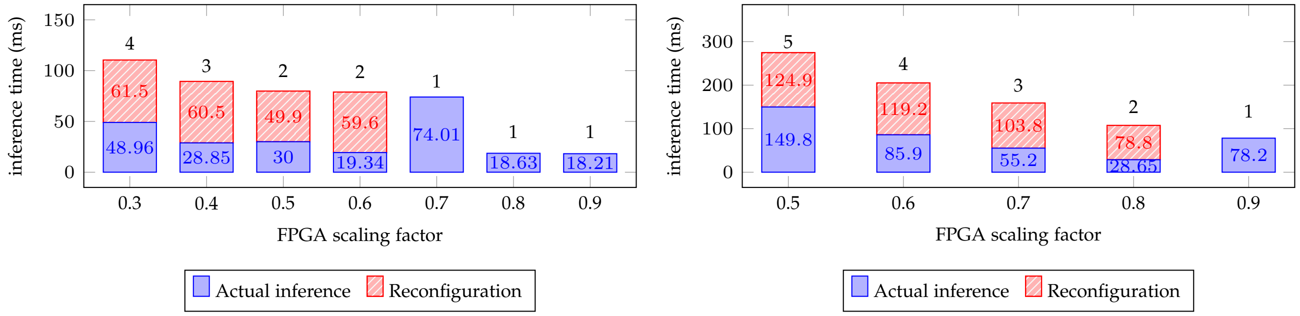
In recent years, deep learning has become a dominant approach in many domains, outperforming traditional algorithmic methods in several tasks, such as image classification, speech recognition, and control. However, due to the large computational demand of DNNs, hardware



(a)

(b)

Fig. 6. Comparison of LUT utilizations as a function of BRAM utilization for the configurations generated by (a) static_baseline_CNVW1A1 and (b) static_baseline_CNVW2A2.



(a)

(b)

Fig. 7. Breakdown of the execution times for the configurations generated by dynamically optimized into cumulative actual inference and reconfiguration times (a) dynamic_OPT_CNVW1A1 and (b) dynamic_OPT_CNVW2A2. The numbers above each bar indicate the number of chunks the accelerator is split into.

accelerators are often employed to speed up their execution. In this regard, FPGA-based hardware accelerators present an attractive solution for embedded systems as they combine the flexibility of programmable logic and partial reconfiguration while achieving high energy efficiency with respect to other forms of hardware accelerations (e.g., GPGPUs [21]). Very efficient implementations of accelerators for DNNs on FPGAs have been proposed by employing fine-grained optimizations to accelerate computational-intensive convolutional layers. Most relevant to us, Zhang et al. [22] leveraged the roofline model for optimizing HLS-based implementations of DNNs, whereas Ma et al. [23] performed a fine-grained characterization of loop optimization techniques for implementing convolutional layers. However, none of them relied on mathematical optimization methods that guarantee optimality.

Efforts [24], [25], [26] have also been spent on improving the performance of FPGA accelerators by exploring throughput vs. memory bandwidth trade-offs. The closer work to ours is the one by Suda et al. [24], which proposed a genetic algorithm (GA) based on a design space exploration methodology for optimizing DNN accelerators with 8- and 16-bit fixed-point precision. Similarly to our work, in [24] the latencies were analytically modeled, while the resource consumption of the layers was empirically modeled as a function of the network parameters. Then, a GA was used to find the parameters of each layer that minimize the total latency. Our approach differs from the one used in [24] in three major ways. First, our work is based on mathematical optimization that can guarantee the optimality of the results. Second, our work explores the possibility of deploying DNNs under resource-constrained FPGA fabrics by chunking the accelerator pipeline and leveraging DPR. Third, our approach allows balancing the resource utilization of DNN accelerators (i.e., it tries to avoid that a single type of resource causes a bottleneck for deployment), while in [24] the authors reported the rapid exhaustion of DSP resources while the other resources were under-utilized.

FPGAs are especially well suited for the acceleration of quantized neural networks (QNNs) as fixed-point computations can be efficiently implemented using the primitive logic available on FPGA fabrics, while the reduction in precision has a limited impact on accuracy [27], [28]. Even in the extreme case of BNNs, it is possible to achieve adequate precision at the benefit of a largely reduced resource utilization [12], [29]. In the context of FPGA-based hardware acceleration of BNNs, some approaches rely on mapping of the entire network on the programmable fabric [10], [14], [30] while others implement FPGA accelerators only for the most computationally-intensive operations [31]. Several researchers have also proposed different solutions to address the throughput vs. area trade-off of QNN accelerators under resource-constrained FPGAs. The most common approaches leverage partial reconfiguration. For instance, Farhadi et al. [32] proposed an adaptive and hierarchical structure for QNNs that takes advantage of partial reconfiguration to address the limitation of resources on FPGA. Kästner et al. [33] presented a co-design methodology enabling the exploitation of DPR to accelerate DNNs. However, none of the proposed approaches formulates the optimal decomposition of hardware accelerators as an optimization problem as done in this work. Finally, differently to other design space explorations proposed by other authors [10], [34], [35], our work adds a new dimension to the design space exploration: partial reconfiguration.

8 Conclusions

This work proposed an optimization-based technique to improve the timing performance of BNNs under constrained FPGA resources. Accurate modeling of latency and resource consumption of BNN layers has been employed. Then, a MILP formulation has been proposed to both compute the optimal configuration of each layer and decide how to split the network into chunks that are dynamically programmed via partial reconfiguration. The major outcome of this work is that the proposed technique allows deploying a BNN in cases in which it would not be possible with standard approaches. For instance, by leveraging partial reconfiguration, it is capable of producing solutions for an FPGA area as low as 30% of the total one available on a Zynq-7020, while static approaches are not able to produce solutions when the available area is less than about the 70% of the area. As a future work we plan to extend this approach to other frameworks for deploying neural networks and integrate it with tools that automate the floorplanning of the accelerators (e.g., as done in [36]).

References

[1] M. Z. Alom, T. M. Taha, C. Yakopcic, S. Westberg, P. Sidike, M. S. Nasrin, B. C. Van Esesn, A. A. S. Awwal, and V. K. Asari, "The history began from alexnet: A comprehensive survey on deep learning approaches," *[online] Available: https://arxiv.org/abs/1803.01164*, 2018.

[2] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue *et al.*, "An empirical evaluation of deep learning on highway driving," *[online] Availble: https://arxiv.org/abs/1504.01716*, 2015.

[3] H. Gao, B. Cheng, J. Wang, K. Li, J. Zhao, and D. Li, "Object classification using cnn-based fusion of vision and lidar in autonomous vehicle environment," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 9, pp. 4224–4231, 2018.

[4] T. Wang, Y. Chen, M. Qiao, and H. Snoussi, "A fast and robust convolutional neural network-based defect detection model in product quality control," *The International Journal of Advanced Manufacturing Technology*, vol. 94, no. 9-12, pp. 3465–3471, 2018.

[5] K. Židek, A. Hosovsky, J. Pitel', and S. Bednár, "Recognition of assembly parts by convolutional neural networks," in *Advances in Manufacturing Engineering and Materials*. Springer, 2019, pp. 281–289.

[6] J. Redmon and A. Angelova, "Real-time grasp detection using convolutional neural networks," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 1316–1322.

[7] J. Watson, J. Hughes, and F. Iida, "Real-world, real-time robotic grasping with convolutional neural networks," in *Annual Conference Towards Autonomous Robotic Systems*. Springer, 2017, pp. 617–626.

[8] M. Tsukada, M. Kondo, and H. Matsutani, "A neural network-based on-device learning anomaly detector for edge devices," *IEEE Transactions on Computers*, pp. 1–1, 2020.

[9] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "[dl] a survey of fpga-based neural network inference accelerators," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 12, no. 1, pp. 2:1–2:26, Mar. 2019. [Online]. Available: http://doi.acm.org/10.1145/3289185

[10] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. ACM, 2017, pp. 65–74.

[11] Y. Guo, "A survey on methods and theories of quantized neural networks," *CoRR*, vol. abs/1808.04752, 2018. [Online]. Available: http://arxiv.org/abs/1808.04752

[12] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," *CoRR*, vol. abs/1602.02830, 2016. [Online]. Available: http://arxiv.org/abs/1602.02830

[13] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European conference on computer vision*. Springer, 2016, pp. 525–542.

[14] P. Guo, H. Ma, R. Chen, P. Li, S. Xie, and D. Wang, "Fbna: A fully binarized neural network accelerator," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2018, pp. 51–513.

[15] B.-P. project, ",," https://github.com/Xilinx/BNN-PYNQ, 2020, [Online; accessed 11-January-2020].

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

[17] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[18] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 12, pp. 2481–2495, 2017.

[19] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[21] 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 *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–4.

[22] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 161–170.

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

[24] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 16–25.

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

[26] J. Zhang and J. Li, "Improving the performance of opencl-based fpga accelerator for convolutional neural network," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 25–34.

[27] W. Sung, S. Shin, and K. Hwang, "Resiliency of deep neural networks under quantization," *[online] Available: https://arxiv.org/abs/1511.06488*, 2015.

[28] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," *[online] Available: https://arxiv.org/abs/1702.03044*, 2017.

[29] M. Kim and P. Smaragdis, "Bitwise neural networks," *[online] Available: https://arxiv.org/abs/1601.06071*, 2016.

[30] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional

neural networks with software-programmable fpgas," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 15–24.

[31] D. J. M. Moss, E. Nurvitadhi, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. W. Leong, "High performance binary neural networks on the xeon+fpga platform," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–4.

[32] M. Farhadi, M. Ghasemi, and Y. Yang, "A novel design of adaptive and hierarchical convolutional neural networks using partial reconfiguration on fpga," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–7.

[33] F. Kästner, B. Janßen, F. Kautz, M. Hübner, and G. Corradi, "Hardware/software codesign for convolutional neural networks exploiting dynamic partial reconfiguration on pynq," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 154–161.

[34] S. I. Venieris and C.-S. Bouganis, "fpgaconvnet: Automated mapping of convolutional neural networks on fpgas (abstract only)," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 291–292. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021791

[35] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, "Design space exploration of fpga-based deep convolutional neural networks," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 575–580.

[36] B. B. Seyoum, A. Biondi, and G. C. Buttazzo, "Flora: Floorplan optimizer for reconfigurable areas in fpgas," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3358202

Biruk Seyoum Biruk Seyoum is a Ph.D candidate at the Real-Time Systems (ReTiS) Laboratory of Scuola Superiore Sant'Anna. He graduated in Electrical and Computer Engineering from Addis Ababa Institute of Technology and received M.S degree in Telecommunication and Electronics from the University of Trento in Italy. His research interests include building design tools and applications for Dynamic Partially Reconfigurable platforms, FPGA-based Deep Neural Networks acceleration, tools and frameworks for DNN auto-implementation on FPGAs.

Marco Pagani Marco Pagani is a Ph.D. candidate at the Real-Time Systems (ReTiS) Laboratory of the Scuola Superiore Sant'Anna of Pisa and at the Embedded Real-Time Adaptative System Design and Execution (meraude) team from the Centre de Recherche en Informatique, Signal et Automatique (CRIStAL) based in Lille. He received his M.Sc. degree in Embedded Computing Systems cum Laude in 2016 jointly from Scuola Superiore Sant'Anna and the University of Pisa. His research interests include predictable hardware acceleration for real-time applications on heterogeneous computing platforms and real-time operating systems for embedded platforms.

Alessandro Biondi Alessandro Biondi is Assistant Professor at the Real-Time Systems (ReTiS) Laboratory of the Scuola Superiore Sant'Anna. He graduated (cum laude) in Computer Engineering at the University of Pisa, Italy, within the excellence program, and received a Ph.D. in computer engineering at the Scuola Superiore Sant'Anna under the supervision of Prof. Giorgio Buttazzo and Prof. Marco Di Natale. In 2016, he has been visiting scholar at the Max Planck Institute for Software Systems (Germany). His research interests include design and implementation of real-time operating systems and hypervisors, schedulability analysis, cyber-physical systems, synchronization protocols, and component-based design for real-time multiprocessor systems. He was recipient of six Best Paper Awards, one Outstanding Paper Award, the ACM SIGBED Early Career Award 2019, and the EDAA Dissertation Award 2017.

Sara Balleri Sara Balleri received the graduate degree in Computing Engineering at the University of Pisa, and received the master's degree (cum laude) in Embedded Computing Systems jointly offered by the Scuola Superiore Sant'Anna and University of Pisa. Now, she works in a company that operates in information technology and provides IT solutions in the sectors of embedded and real-time software, software design and development for civil and military radar system, and industrial automation.

Giorgio Buttazzo Giorgio Buttazzo is full professor of computer engineering at the Scuola Superiore Sant'Anna of Pisa. He graduated in Electronic Engineering at the University of Pisa in 1985, received a M.S. degree in Computer Science at the University of Pennsylvania in 1987, and a Ph.D. in Computer Engineering at the Scuola Superiore Sant'Anna of Pisa in 1991. He is Editor-in-Chief of Real-Time Systems, Associate Editor of the ACM Transactions on Cyber-Physical Systems, and IEEE Fellow since 2012. He has authored 7 books on real-time systems and over 200 papers in the field of real-time systems, robotics, and neural networks.