UNIVERSITA' DELLA CALABRIA

Dipartimento di Ingegneria Informatica, Modellistica, Elettronica e Sistemistica

Dottorato di Ricerca in

Information and Communication Technologies

Con il contributo di

POR Calabria FSE/FESR 2014 – 2020

CICLO XXXV

Design Methodologies for FPGA-based Deep Learning Accelerators and Their Characterization

Settore Scientifico Disciplinare ING-INF/01

Coordinatore: Ch.mo Prof. Giancarlo Fortino

[redacted]

Firma oscurata in base alle linee guida del Garante della privacy

Supervisore: Ch.ma Prof.ssa Stefania Perri

Firma oscurata in base alle linee guida del Garante della privacy

Co-Supervisore: Ch.mo Prof. Pasquale Corsonello

Firma oscurata in base alle linee guida del Garante della privacy

Dottorando: Dott. Cristian Sestito

Firma oscurata in base alle linee guida del Garante della privacy

“La borsa di dottorato è stata cofinanziata con risorse del Programma Operativo Regionale Calabria FSE/FESR 2014 – 2020 (CCI 2014IT16M2OP006)”

To my parents

ABSTRACT

Deep Neural Networks (DNNs) are widespread in many applications, including computer vision, speech recognition and robotics, thanks to the ability of such models to extract information by building a hierarchical representation of knowledge.

Image processing benefits from the latter behavior by using Convolutional Neural Networks (CNNs), which consist of several Convolutional (CONV) layers to extract features from inputs at different levels of abstraction. However, CNNs usually require billions of computations to reach high accuracy levels. In order to sustain such computational load, proper hardware acceleration is needed.

Field Programmable Gate Arrays (FPGAs) have been shown as promising candidates, because they are able to achieve high throughput at limited power dissipation. In addition, FPGAs are flexible architectures to accommodate several CNNs' workloads. While the hardware acceleration of conventional CNN models has been widely investigated, the interest about more sophisticated tasks is still emerging. The latter includes CNNs based on Dilated Convolutions (DCONVs) and Transposed Convolutions (TCONVs), which deal with filter and image dilations, respectively. Accordingly, higher computational complexity is exhibited by these architectures, thus requiring careful hardware management.

This PhD dissertation deals with the FPGA acceleration of CNNs for Image Processing based on DCONVs and TCONVs. Specifically, several designs using both the Very High-Speed Integrated Circuits Hardware Description Language (VHDL) and the High-Level Synthesis (HLS) are presented. Detailed characterization is discussed, based on the evaluation of resources occupation, throughput, power dissipation, as well as the impact of data quantization. Overall, the proposed circuits show noticeable energy-efficiency when compared to several state-of-the-art counterparts. For instance, hardware

acceleration of run-time reconfigurable CONVs and TCONVs for super-resolution imaging has shown an energy-efficiency of up to 518.5 GOPS/W, by outperforming state-of-the-art competitors by up to 2.3 times.

ABSTRACT (ITALIANO)

Le reti neurali profonde sono diffuse in diverse applicazioni, tra cui la computer vision, il riconoscimento vocale e la robotica, grazie alla loro capacità di estrarre informazioni tramite la costruzione di una rappresentazione gerarchica di conoscenza.

Il processamento di immagini sfrutta questo comportamento attraverso reti neurali convoluzionali, che consistono di diversi layers di convoluzione per estrarre caratteristiche dagli ingressi a diversi livelli di astrazione. Tuttavia, tali modelli richiedono miliardi di calcoli per raggiungere alti livelli di accuratezza. Per sostenere questo carico computazionale, un'opportuna accelerazione hardware è necessaria.

I Field Programmable Gate Array (FPGA) si sono dimostrati candidati promettenti, in quanto sono capaci di raggiungere alte prestazioni con limitato consumo di potenza. In aggiunta, i FPGAs sono architetture flessibili e permettono di ospitare diverse configurazioni di reti neurali convoluzionali. Mentre l'accelerazione hardware di modelli convenzionali è stata ampiamente esaminata, l'interesse verso applicazioni più sofisticate è ancora emergente. Queste ultime includono le reti neurali convoluzionali basate su convoluzioni dilatate e convoluzioni trasposte, che trattano rispettivamente filtri e immagini dilatati. Di conseguenza, esse esibiscono una maggiore complessità computazionale, così da richiedere un'attenta gestione dell'hardware.

Questa tesi di Dottorato di Ricerca tratta l'accelerazione su FPGA di reti neurali convoluzionali per il processamento di immagini e basate su convoluzioni dilatate e trasposte. Nello specifico, vengono esaminati diversi design, che sfruttano sia il VHDL e la sintesi ad alto livello per la descrizione dell'hardware. Una dettagliata caratterizzazione è presentata, basata sulla valutazione dell'occupazione delle risorse, delle performance e dei consumi di potenza, così come dell'impatto della quantizzazione dei dati. In generale,

i circuiti proposti mostrano una considerevole efficienza energetica quando sono confrontati con lo stato dell'arte. Per esempio, l'accelerazione hardware di un circuito adattabile in tempo reale sia a convoluzioni che a convoluzioni trasposte, per il processamento di immagini a super risoluzione, ha mostrato un'efficienza energetica fino a 518.5 GOPS/W, performando fino a 2.3 volte meglio rispetto ad altri lavori in letteratura.

ACKNOWLEDGEMENTS

At the end of this PhD journey, I would like to express my gratitude to people who supported me, both directly and indirectly.

My heartfelt thanks to Prof. Stefania Perri, my PhD supervisor. She constantly guided me towards the right way, by providing the proper help to improve my weaknesses. Her academic expertise allowed me to conclude this thesis successfully. Special thanks to Prof. Pasquale Corsonello, my PhD co-supervisor, who critically helped me to strengthen my scientific analytical skills.

Many thanks to the staff of the Electronics group at DIMES department. Particularly, I would like to mention Dr. Fanny Spagnolo, Prof. Fabio Frustaci, Eng. Giovanni Staino, who provided generous support and nice working days. Fanny, your advices and your academic contributions were really helpful to effectively face this PhD.

Many thanks to my PhD colleagues for everyday chatting and for academic advices and discussions. Warm thanks to Massimo, dear friend, who always believed in me.

During my PhD, I had the pleasure to spend several months in Edinburgh as Visiting Scholar at Heriot-Watt University. I met a lot of people who helped me to spend the most meaningful experience of my life. I am very grateful to Dr. Rob Stewart, my external advisor, who welcomed me in the best way. He provided me with all the support, proper advices, and friendly discussions to carry out an important piece of work for my PhD. In addition, I am also grateful to the staff of the Lab for AI Verification (LAIV), excellently led by Prof. Katya Komendantskaya, to have got me involved in their activities.

Special thanks to my new (and old) friends in Edinburgh. Just to cite some of them: Simona, Marco, Luca, Birthe, Natalia, Matthew, Martina, the Greek team (Penny, Kostas,

Spiros, Yannis). Simona, you are always present and helpful; I am very grateful to you for all. You deserve the best. Marco, you have proven yourself to be a worthy friend; we had a lot of nice time in Edinburgh!

Finally, the most important acknowledgements to my family, for the invaluable support and for letting me be who I am today.

Cristian

RINGRAZIAMENTI

Al termine di questo percorso di Dottorato di Ricerca, vorrei esprimere la mia gratitudine alle persone che mi hanno supportato, sia direttamente che indirettamente.

Grazie di cuore alla Prof.ssa Stefania Perri, mio supervisore. Lei mi ha guidato costantemente verso la strada giusta, fornendomi il giusto aiuto per superare le mie debolezze. La sua competenza accademica mi ha permesso di portare a termine questo lavoro di tesi con successo. Un grazie speciale al Prof. Pasquale Corsonello, mio co-supervisore, che mi ha aiutato criticamente a migliorare le mie capacità di analisi scientifiche.

Molte grazie allo staff del gruppo di Elettronica del dipartimento DIMES. In particolare, vorrei menzionare la Dott.ssa Fanny Spagnolo, il Prof. Fabio Frustaci e l'Ing. Giovanni Staino, i quali mi hanno fornito supporto prezioso e giornate lavorative piacevoli. Fanny, i tuoi consigli e i tuoi contributi accademici sono stati di grande aiuto per affrontare con efficacia questo Dottorato.

Molte grazie ai colleghi, sia per le chiacchiere quotidiane sia per i consigli e le discussioni accademiche. Un sentito ringraziamento a Massimo, caro amico, che ha sempre creduto in me.

Durante questo Dottorato, ho avuto il piacere di trascorrere diversi mesi ad Edimburgo come studente in visita presso la Heriot-Watt University. Ho conosciuto molte persone che mi hanno permesso di vivere l'esperienza più significativa della mia vita. Sono molto grato al Dott. Rob Stewart, mio supervisore esterno, che mi ha accolto nel miglior modo possibile. Mi ha fornito tutto l'aiuto necessario, ottimi consigli e discussioni amichevoli per condurre un'importante parte della mia ricerca. Inoltre, sono molto grato

allo staff del gruppo LAIV, guidato ottimamente dalla Prof.ssa Katya Komendantskaya, che mi ha coinvolto nelle loro attività.

Un grazie speciale ai nuovi (e vecchi) amici di Edimburgo. Giusto per citarne alcuni: Simona, Marco, Luca, Birthe, Natalia, Matthew, il gruppo greco (Penny, Kostas, Spiros, Yannis). Simona, tu sei sempre presente e di grande supporto. Ti sono grato per tutto. Meriti il meglio. Marco, ti sei dimostrato un grande amico; abbiamo trascorso momenti piacevoli ad Edimburgo!

Infine, il ringraziamento più importante alla mia famiglia, per il supporto impagabile e per avermi permesso di essere chi sono.

Cristian

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1. CONTEXT AND MOTIVATIONS

Nowadays, Deep Learning (DL) is being imposed in many scenarios, including computer vision [1], speech recognition [2], game play [3] and robotics [4]. Such pervasiveness is motivated by the ability of Deep Neural Networks (DNNs) to mimic the brain functionality by building a hierarchical representation of knowledge. This is accomplished by the cooperation of several computing modules, named *layers*, that extract features from raw inputs at different levels of abstraction [5].

Convolutional Neural Networks (CNNs) are powerful DL models that have found ground in the above scenarios, especially image processing for computer vision [6]. Indeed, they make use of Convolutional (CONV) layers to emulate the human visual cortex activity effectively. However, the accuracy power of CNNs usually results in high computational complexity: for instance, ImageNet classification over the Visual Geometry Group (VGG) network requires up to 15.5 Giga Multiply-Accumulations (GMACs), distributed among 16 layers, to reach a 7.4 Top-5 error [7]. Typically, these architectures must fit high-performance applications, thus asking for powerful platforms, such as Graphics Processing Units (GPUs). The latter are well-recognized to meet low latency per single inference, thanks to the intrinsic attitude to parallelize computations. Unfortunately, GPUs demand high power dissipation, a not-negligible challenge when the aforementioned models are conceived to be run off-the-grid (e.g., Edge Computing [8]).

As a result, dedicated hardware acceleration of CNNs is gaining increasing interest in research [9]. Among different devices, Field Programmable Gate Arrays (FPGAs) are showing effective to trade-off speed performance and power dissipation.

FPGAs offer reconfigurable silicon to accommodate different CNN models using the same circuit. Moreover, they benefit of long life cycles (even decades), thus making them suitable for critical environments like defense and medicine [10]. As a further nice characteristic, since 2011 the FPGA market has witnessed the integration of such programmable architectures within complex embedded systems. There, general-purpose processors have also been included to enable heterogeneous tasks, by splitting computations among straightforward software routines and dedicated acceleration circuitries.

1.2. SCOPE OF THIS WORK AND CONTRIBUTIONS

Motivated by the above considerations, this work examines proper design methodologies to implement CNNs into FPGA architectures. As previously highlighted, these models exploit multi-dimensional CONVs that emulate the behavior of the human visual system to get information from data. Specifically, CONVs apply learned filters to inputs to extract features of interest, by means of Multiply-Accumulations (MACs). Such mechanism well works in computer vision, thus endorsing CNNs as a promising paradigm to process both images and videos [11].

Computer vision highly relies on FPGAs, given the availability of efficient logic and memory resources to manage computations and buffering, respectively. While the FPGA acceleration of conventional CNNs has been widely investigated [12], the interest about non-conventional models, referring to Dilated Convolutions (DCONVs) and Transposed Convolutions (TCONVs), is still emerging. In contrast to CONVs, which directly act over the given inputs, the latter ask for preliminary pre-processing. Typically, the latter provide wider representation spaces by means of zeros insertion. As a results, redundant computations are introduced. By the hardware perspective, this means higher resources requirements, longer execution times and lower power efficiency. Consequently, proper optimization policies are mandatory.

This dissertation strives to explore alternative ways to face the aforementioned challenges. To this aim, two key FPGA design methodologies have been taken into account:

(a) Low-level design for CNNs, by using the Very High Speed Integrated Circuits Hardware Description Language (VHDL).
(b) High-level synthesis (HLS) for CNNs, by means of C++ routines.

1.2.1. LOW-LEVEL DESIGN FOR CNNS

Hardware Description Languages, such as VHDL and Verilog, infer the Register-Transfer Level (RTL) abstraction to a circuit model. Programmable logic can be carefully configured to ensure implemented modules being close each other. This results in higher speed performance and lower power dissipation, where design constraints are very strict (e.g., real-time image/video processing).

An efficient hardware design of DCONVs was proposed in [13]. We conceived a novel buffer architecture to manage multiple DCONV, with different dilation factors, in parallel. This module was included into a complete neural network able to perform the À-trous Spatial Pyramid Pooling for biomedical image processing purposes. Post-implementation results, related to the Xilinx Zynq-7020 System-on-Chip (SoC), showed limited logic and memory occupation (i.e., 7.3% and 21%, respectively), by dissipating only 265 mW at the 181 MHz clock frequency.

The challenge of efficient implementation of TCONVs was extensively examined in [14, 15, 16]. First, we proposed a scalable engine suitable for both low-end and high-end FPGA-based SoCs, by exploiting both data- and circuit-level parallelism [14]. The effectiveness of such architecture is due to a careful use of the on-chip Digital Signal Processors (DSPs) and the management of memory transactions to allow the raster-order transfer policy for both input and output images. When the Zynq-7045 SoC was taken into account, the proposed system was made able to outperform the state-of-the-art computational capabilities of ~20%, saving more than 60% of power consumption.

Considering that TCONVs usually work jointly with conventional CONVs, further investigations were carried out to conceive a reconfigurable architecture suitable for both of them. As a beginning goal, we proposed a run-time adaptive convolution architecture [15]. The latter, able to maximize the utilization of needed resources for each provided

configuration, was made capable of supporting a peak throughput of 217.2 Giga Operations per Second (GOPS), and reaching the most favorable throughput vs parallelism ratio among the competitors. Afterwards, the circuit was also equipped with run-time TCONVs support, by conceiving a proper image buffering on-chip [16]. Unlike the state-of-the-art counterparts, the referred design avoided complex filters reorganization to nullify redundant MACs. As a case-study, the hardware model was used to perform the Super-Resolution imaging task within the Xilinx Kintex-410T FPGA. Comparisons with previous works showed that the proposed accelerator can save up to ~63% and ~48% of logic and power, respectively, without compromising the overall accuracy.

1.2.2. HIGH-LEVEL SYNTHESIS FOR CNNS

TCONV-based CNNs suitable for FPGAs were also proposed by leveraging high-level C++ routines to instruct the synthesis tool. With respect to low-level VHDL circuits, the latter design strategy makes the overall flow faster, by accelerating both verification and design space exploration [17]. Furthermore, template structures can be inferred to provide parameterizable sketches at design time, thus ensuring reusability in different applications. However, the usage of proper directives, named *pragmas*, is mandatory to guarantee an efficient hardware implementation in terms of resources occupation, latency and power dissipation.

The suitability of HLS design was examined in very low-precision DNNs, where input data were quantized to limited bit-widths [18]. This because such models, even experiencing a slight loss of accuracy, require much lower resources with respect to the full-precision counterparts. As a result, this enables the possibility of dataflow implementations, where each layer is handed over dedicated logic. In this way, low-level controls for run-time reconfigurability are no more required, and proper HLS scripts could be enough to meet the design constraints.

Specifically, the impact of bit-width over TCONV-based models was investigated [19, 20, 21]. First, we carried out a high-level study about the effects over accuracy, by referring to state-of-the-art DL architectures and considering GPUs as training platforms [19]. Specifically, a Convolution Autoencoder, the Deep Convolutional Generative Adversarial Network (DCGAN) [22] and the U-Net [23] were evaluated over proper

training datasets. Experiments showed that data quantization slightly impacts over accuracy in the range 8–5 bit integer, when compared to the 32-bit floating point references. This result suggested that the deployment of such models onto FPGAs could be promising.

Indeed, a HLS model of a TCONV engine was presented in [20]. When synthesized within a Xilinx Virtex-7 FPGA, the architecture was made capable to process 256×256 images experiencing a rate of 53 frames-per-second (fps). This work was further improved to fit parameterizable TCONV layers for DL [21]. As a case-study, the decoder section of the Convolution Autoencoder [19] was implemented within the Zynq-7020 SoC, by providing a design-space in terms of data precision and parallelism. Results showed a limited use of logic and on-chip memory (i.e., ~15% and ~7.5%, respectively), at the expense of only ~2.5% accuracy loss varying the precision from 8 to 4 bits. In addition, the maximum achievable parallelism provided the accelerator with 3.5× speed-up with respect to the baseline design.

1.2.3. THE MEETING POINT OF THESE CONTRIBUTIONS

While this dissertation presents the low-level design and the high-level synthesis within separate sections, the goals to which aspire both the approaches are the same. Indeed, efficient FPGA acceleration of CNNs for image processing aims at:

- Deploying adaptive architectures within the same chip. This can be accomplished by either exploiting a control unit that run-time reconfigures the same circuit to manage different CONV layers (i.e., the proposed low-level designs), or by providing dedicated areas for each layer that cooperate in a dataflow manner (i.e., the high-level synthesis investigation).
- Maximizing the throughput (or minimizing the latency), by carefully using the VHDL methodologies (e.g., sequential processes) or by adopting HLS directives to instruct the synthesis tool properly.
- Minimizing the power dissipation, as a consequence of proper design related to the previous remarks.

1.3. THESIS OUTLINE

This PhD dissertation is structured as follows:

- Chapter 2 provides a background about DL and CNNs, by emphasizing the functionality of DCONVs and TCONVs, main characters of such work, and providing examples of image processing tasks. Furthermore, the importance of FPGAs in CNN acceleration is motivated, by contextualizing them among other high-performance devices, as well as providing relevant state-of-the-art FPGA implementations of CNNs.
- Chapter 3 illustrates the novel low-level designs for DCONVs [13] and TCONVs [14, 15, 16]. For each research work, the proposed circuit, experiments and results are discussed. With respect to TCONVs, further clarifications about the adopted algorithms are also provided.
- Chapter 4 presents the high-level synthesis of TCONV engines for CNNs [20, 21]. In this section, particular attention is provided to the C++ code to make the reader aware of the careful writing style to get efficient hardware synthesis. Moreover, given that the latter engines have been made suitable for low-precision networks, a thorough survey about the quantization approach adopted [19] is provided.
- Chapter 5 draws the conclusion of this PhD research and provides insights to further the activity.

Figure 1.1 schematizes the organization of the thesis by means of a mind map, where the key research motivations and contributions are reported, as well as their position within the dissertation.

The motivations (*Why?*) of this PhD Research are motivated in Sections 1.1 and 2.4. To further confirm the suitability of FPGAs, Section 2.5 provides a background about the main related works about CNNs acceleration. However, further works are pointed out throughout the whole thesis ("Background" sections within the Chapters 3, 4).

The contribution (*What?*) is presented in detail in Chapters 3 and 4, by highlighting architectures, designs, and setup for experiments.

The tools to effectively conceive and complete the designs answer to the "*How?*" question. The theoretical basics are covered in Sections 2.1. 2.2. and 2.3, while the

extensive characterization in terms of resources occupation, performance, power is discussed in Chapters 3 and 4.

Finally, the key messages of this Research are summarized in Section 5.

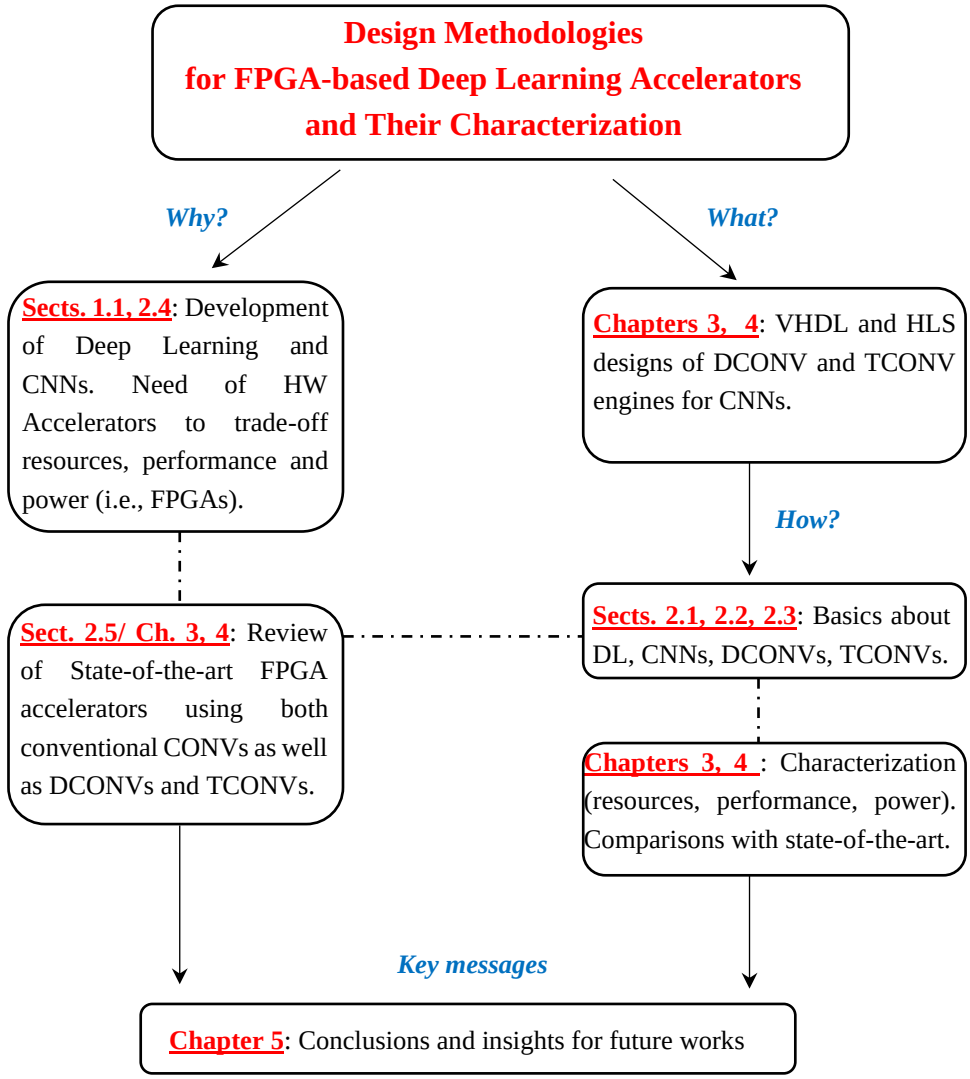


Figure 1.1 Organization of the PhD thesis.

2. BACKGROUND

This chapter firstly provides a background about Deep Learning and Convolutional Neural Networks. The role of Dilated Convolutions and Transposed Convolutions is emphasized, by illustrating some applications related to image processing. Afterwards, the main architectures to accelerate Convolutional Neural Networks are presented, by highlighting the importance of FPGAs. Finally, state-of-the-art FPGA accelerators are reviewed to present the challenges behind efficient hardware implementations.

2.1. ARTIFICIAL INTELLIGENCE AND DEEP LEARNING

Nowadays, one of the key goals of engineering is to equip systems with intelligence, in order to either assist humans in everyday activities or to replace them when these activities may be performed by machines autonomously. Artificial Intelligence (AI) meets the above tasks by using Machine Learning (ML), which provides systems with programs able to learn themselves without users' programming.

The idea of *learning something* makes us immediately think about the behavior of human brain to connect sparse information to build knowledge. Accordingly, ML inherits a sub-field, named brain-inspired computation, which emulates the brain attitude to learn and address challenges.

The human brain exploits billions of elementary computing cells, called *neurons*, to extract and propagate information either internally or towards the rest of the body. Several input connections, called *dendrites*, supply the neuron with data. The latter performs computations to produce a result, the *activation*, which is forwarded to another

neuron through a connection named *axon*. The meeting point between an axon and a dendrite is known as *synapse*.

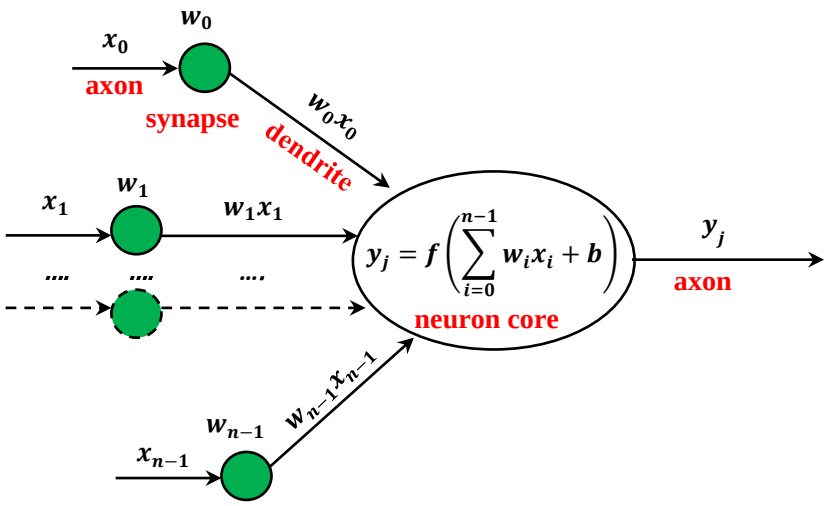


Figure 2.1 Schematization of a neuron.

Figure 2.1 schematizes the model of the neuron, following the theory of the American psychologist Frank Rosenblatt (1958). The input activations $x = (x_0, x_1, \dots, x_{n-1})$ provided by as many axons are preliminary scaled by the *weights* $w = (w_0, w_1, \dots, w_{n-1})$ associated to each synapse. The scaled data $wx = (w_0x_0, w_1x_1, \dots, w_{n-1}x_{n-1})$ are supplied to the neuron core that performs a weighted sum, by also adding an extra contribution named *bias* (b) to calibrate the working point of the neuronal cell. Finally, this computation is subjected to a non-linear function $f(\cdot)$, which represents data in a compressed domain. The output y_j is forwarded to another neuron for further processing. It is worth underlining that weights are responsible for learning and they are the only values to be adjusted by brain to improve the knowledge acquisition.

Neural Networks (NNs) are arranged as *layers* of neurons in order to extract complex information from input data. Figure 2.2 illustrates an example NN consisting of three layers. The *input layer* simply forwards the inputs $i = (i_0, i_1, \dots, i_{n-1})$ to the hidden layer, whose neurons are responsible to perform the computations reported in Figure 2.1. Finally, the *output layer* executes the last computations to provide the final results $o = (o_0, o_1, \dots, o_{n-1})$.

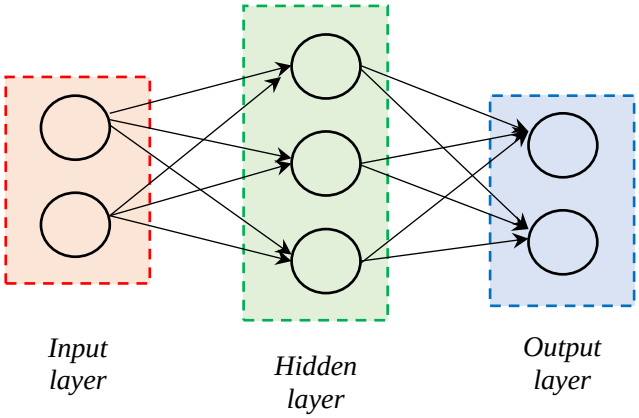


Figure 2.2 Example of neural network.

In order to carry out their task effectively, NNs usually rely on more than three layers: indeed, they may accommodate hundreds of hidden layers. When this occurs, we refer to Deep Learning (DL) and the associated NNs are known as *Deep Neural Networks* (DNNs).

DNNs work through a two-way process. During the *training* phase, the given DNN is in learning mode, where the weights are continuously adjusted within multiple iterations. During each iteration, inputs are processed by the DNN by forwarding the information across the layers. Weights are updated according to the achieved learning level, which is evaluated on the quality of the current outputs: if the results are quite good, the weights adjustment is minimal. Conversely, worse results mean higher corrections to the weights values. This process is managed through a *backpropagation* step, which can be seen as an optimization problem to manage the direction of the weights distribution. At the completion of the training, weights are final. Eventually, the *inference* phase processes new data by using that weights distribution.

State-of-the-art DNNs exploit millions of learnable weights in order to execute task at high accuracy levels. As a result, this demands many computations and high amount of storage. For instance, conventional DNNs exploit Fully-Connected (FC) layers in which each generated activation comes from the weighted sum of all the input activations of the previous layer. In order to alleviate the computational effort of the current layer, it is possible to reduce the number of weights to provide each output: the strategy of *weight sharing* allows to generate outputs by using the same group of weights. Convolutional Neural Networks (CNNs) are a meaningful example acting in this way, in which inputs

are arranged as windows of values where the same group of weights, named *filters*, are responsible to meet the desired task. This class of DNNs is the focus of this dissertation and it is described in the following Section.

2.2. CONVOLUTIONAL NEURAL NETWORKS

Convolutional Neural Networks (CNNs) are DL models able to extract informative content from multi-dimensional arrays, which can represent data sequences, images and videos. This task is carried out by convolutions that determine features from inputs, both at low-level (e.g., edges) and at high-level (e.g., the complex shape of an object).

Generally speaking, CNNs consist of two main parts: (a) the feature extractor, structured as a sequence of Convolution Layers (CONV Layers), and (b) a task-oriented section, whose architectural organization differs from model to model. In what follows, the attention is focused on CNNs for image processing. Figure 2.3 illustrate a top-level example of CNN.

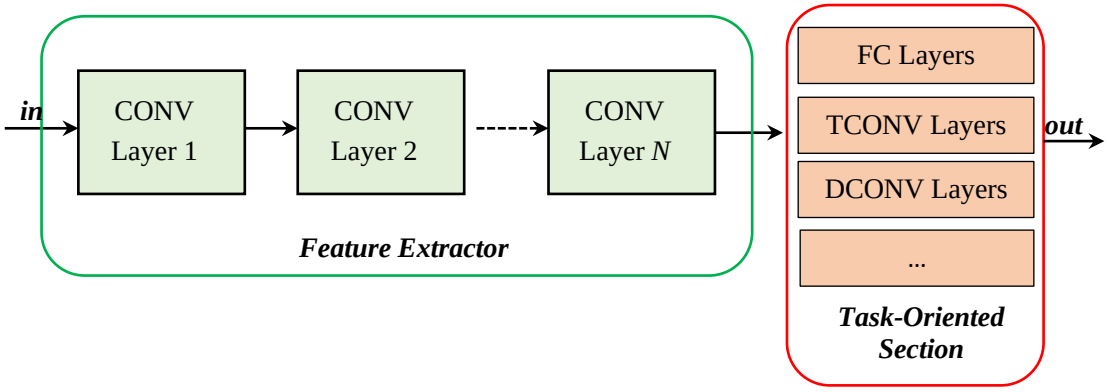


Figure 2.3 Example of Convolutional Neural Network.

2.2.1. THE FEATURE EXTRACTOR

Typically, the very first CONV Layer can receive, as input, either a gray-scale image or a RGB frame. These data are organized as I_C channels of $H_I \times W_I$ pixels matrices, where I_C=1 for the gray-scale format and I_C=3 for the RGB representation. Proper filters perform computations over these data to extract features of interest. Specifically, O_C filters consisting of I_C kernels of $K_H \times K_W$ weights are provided, each responsible to

generate a specific feature map (*fmap*). In the following, it is supposed to work with K_H=K_W=K. Each K×K weights kernel moves around the respective H_I×W_I pixels channel, in a sliding-window manner, and is subjected to Multiply-Accumulations (MACs). Accordingly, each window provides one output pixel. The provisional I_C O_H×O_W output matrices are then summed-up in a pixel-wise manner to finally generate just a single O_H×O_W output *fmap* (*ofmap*). Finally, it is worth underlining that some models add bias offsets to the *ofmaps*. This process is repeated for all the O_C filters, to furnish as many as *ofmaps*. Figure 2.4 illustrates an example when H_I=W_I=4, K=3, I_C=3, O_C=1 and the first output pixel is produced. Colors highlight both sliding windows and provisional outputs interested to.

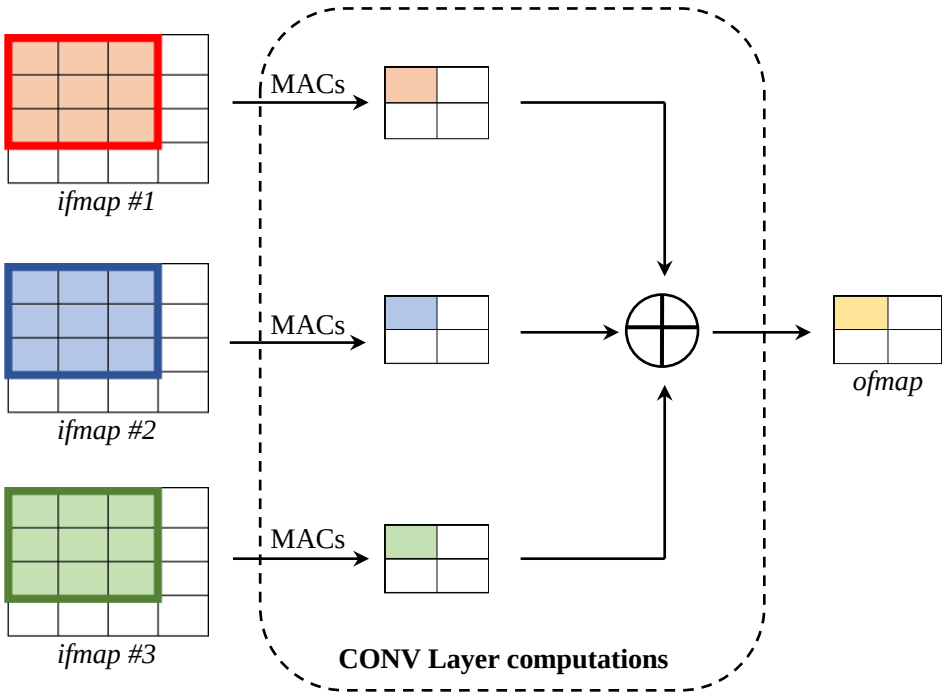


Figure 2.4 Example of CONV Layer when H_I=W_I=4, K=3, I_C=3, O_C=1.

The subsequent CONV Layers act in the same way, but they only manage *fmaps* instead of images: the main difference between the two representations is that *fmaps*' pixels contain encoded information from input images, while images represent explicit objects/scenes. CONV Layers can be followed by auxiliary units, including non-linear activations, normalization and pooling layers. Non-linear activations let CNNs to learn complex tasks by breaking linearity provided by CONV Layers. Basically, such functions change the representation domain of *ofmaps*, either by shrinking the content in a given

range (e.g., sigmoid and hyperbolic tangent [24]) or by thresholding the acceptability values (e.g., Rectified Linear Unit (ReLU) [25]). Normalization [26] is used to control the values distribution of *fmaps* in order to boost training and provide higher accuracy to the model. Finally, pooling [27] aims at reducing the spatial sizes of *fmaps* to both modulate the computational complexity of subsequent layers and infer more robust abstraction capability to the neural network. This is accomplished by splitting each *fmap* in small windows and extracting either the maximum value or the average one.

2.2.2. THE TASK-ORIENTED SECTION

After the deep feature extraction provided by CONV Layers, the provisional outputs are sent to a different section that is responsible of the specific behavior of the CNN. For instance, Fully-Connected (FC) Layers can be used for image classification: by means of computationally-intensive matrix-matrix multiplications, they progressively transform multi-dimensional *fmaps* into a 1-dimensional array. Proper non-linear activations, such as the softmax [28], represent the raw elements of the given array as a probability distribution, meaning that each element belongs in (0,1) and their sum is 1. Specifically, each value of the array indicates the probability that a given class appears within the input image.

The task-oriented section can also adopt different types of layers to comply with other applications. In this regard, Table 2.1 highlights some ways in which the referred module can be organized. Some popular CNNs are reported, by summarizing: (1) the number of CONV Layers of the feature extractor, (2) the type of task-oriented section, and (3) the specific task carried out by the model.

Table 2.1 Examples of Convolutional Neural Networks

Model	Feature extractor	Task-oriented section	Application
VGG-16 [7]	13 CONV Layers	3 FC Layers	Image classification
YOLO [29]	24 CONV Layers	2 FC Layers	Object detection
SegNet [30]	13 CONV Layers	5 Un-pooling Layers + 13 CONV Layers	Semantic Segmentation
FSRCNN [31]	7 CONV Layers	1 TCONV Layer	Super resolution Imaging
DilatedNet [32]	VGG-16 based	Multi-scale context aggregation using DCONV Layers	Semantic Segmentation

As it can be observed, while models for image classification [7] and object detection [29] usually rely on FC Layers to interpret the extracted features, more sophisticated tasks require complex architectures. For example, SegNet [30] is responsible to execute semantic segmentation by means of an encoder-decoder architecture: the feature extractor, also known as encoder, consists of 13 CONV Layers and pooling layers for down-sampling. The outputs coming from the encoder must be restored to the original sizes. This is accomplished by another module, named decoder, which includes un-pooling layers for image up-sampling and CONV Layers for learning.

Image up-sampling can also be self-learned by using Transposed Convolution (TCONV) Layers, as happens in the Fast Super-Resolution CNN (FSRCNN) [31]. There, the cooperation of an encoder consisting of 7 CONV Layers and a TCONV Layer decoder lets an input image to be represented as 2×, 3×, 4× the original sizes. Other models aim at extracting relationship between far pixels in order to understand image context properties [32]: this can be managed by using Dilated Convolution (DCONV) Layers, which dilate the $K \times K$ weights to enlarge the sliding windows. Semantic segmentation is an example task benefiting of the latter strategy.

2.3. DILATED AND TRANSPOSED CONVOLUTIONS

2.3.1. DILATED CONVOLUTIONS

Dilated Convolutions are also known as *à-trous* convolutions, since their original usage interested the *algorithme à trous* for the wavelet decomposition [33].

In contrast to conventional convolutions, the *à-trous* class preprocesses the generic $K \times K$ weights kernel to enlarge the sliding window area. This is accomplished by introducing a further parameter, named dilation factor R. The latter is responsible of inserting $R–1$ zeros between adjacent weights. In this way, the effective kernel area is $K_E \times K_E$, with $K_E = K + (K–1) \times (R–1)$. After this dilation, the convolution is performed in the straightforward way, by means of MAC computations. Figure 2.5 illustrates an example when $K=3$ and $R=2$. The square box with the red borders is the current sliding window that exhibits $K_E=5$. The orange cells represent the positions in which the actual weights

are multiplied by the respective pixels; the remaining cells refer to zeros. The yellow cell within the *ofmap* is the result coming out from the first sliding window MAC.

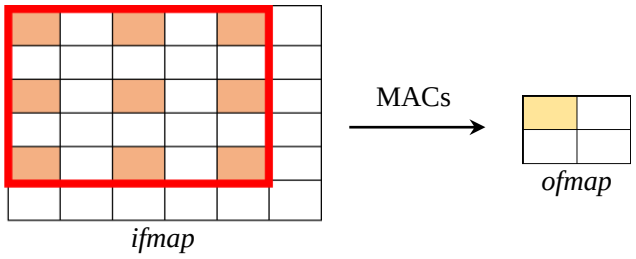


Figure 2.5 Example of DCONV when K=3, R=2.

2.3.2. TRANSPOSED CONVOLUTIONS

While conventional convolutions typically down-sample data by summarizing each sliding window with just one output pixel, transposed convolutions act in the reverse way, by providing more information with respect to the inputs. For this reason, sometimes the latter are also named *deconvolutions*[1].

Transposed Convolutions apply a $K \times K$ weights kernel over a dilated *fmap*, where the actual pixels are interleaved by S–1 zeros, with S being the stride or up-sampling factor. In addition, given P the padding size, K–P–1 further zeros can be placed at the boundaries of the *fmap*. Figure 2.6 shows a transposed convolution that provides a 5×5 *ofmap* starting from a 3×3 *ifmap*. This can be done by setting K=3, S=2 and P=1. In this way, the input pixels are spaced each other by one column/row of zeros. In addition, 1 padding row/column is added at the borders of the dilated *ifmap*. The green cells within the *ifmap* indicate the positions of the actual input pixels; the square box having red borders is the current sliding window, while the orange cell within the *ofmap* is the current output pixel.

[1] *Deconvolution* is an improper term in that mathematical deconvolutions differ significantly from transposed convolutions [64]. However, it is often used in literature to refer to Transposed Convolutions for simplicity, because the prefix '*de*' suggests the idea of the opposite process of convolutions: thus, recovering a high-resolution image starting from a low-resolution representation. In this dissertation, both Transposed Convolutions and Deconvolutions are used to indicate the same algorithm.

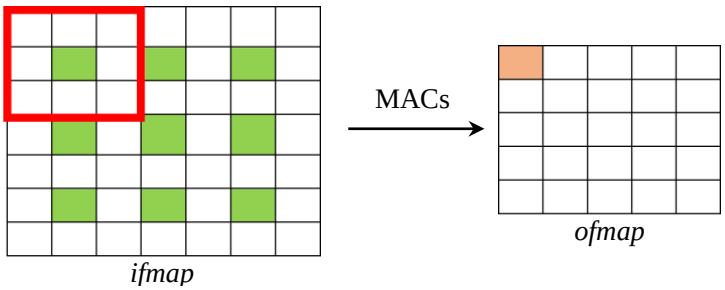


Figure 2.6 Example of TCONV when *K*=3, *S*=2, *P*=1.

2.3.3. IMAGE PROCESSING TASKS USING DCONVS AND TCONVS

DCONVs and TCONVs introduced in this Section may be exploited in different tasks related to image processing.

DCONVs positively impact semantic segmentation, due to the ability of dilated filters to strengthen wide-range relationships among pixels, other than avoiding features down-sampling through non-learnable pooling layers. For instance, the À-trous Spatial Pyramid Pooling [34] (ASPP) is a strategy in which multiple DCONV are performed on the same input image, by using ever-more dilated filters, to equip the model with accurate pixel-level classification. The Spinal Cord Gray Matter segmentation is an application that benefits from the ASPP approach through DCONVs. Figure 2.7 illustrates the top-level architecture, from [35], which performs the referred task.

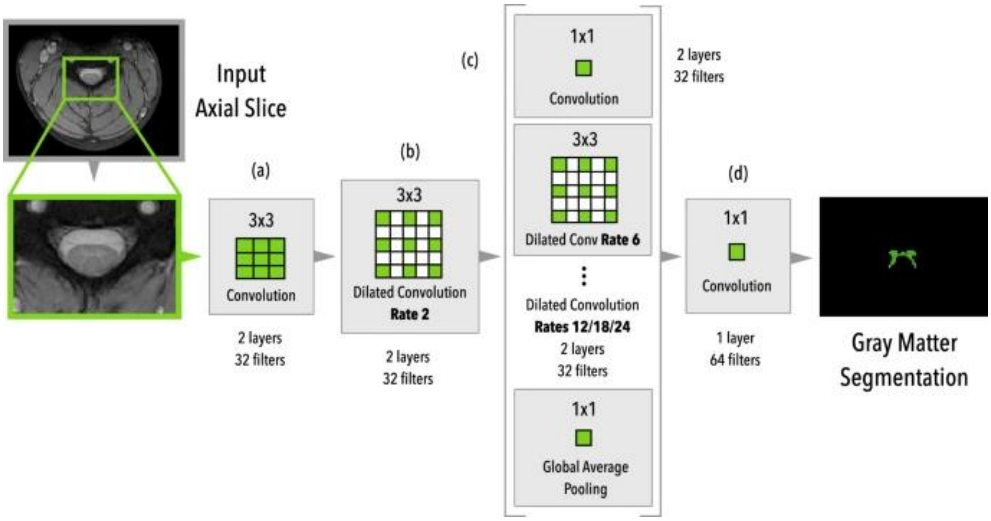


Figure 2.7 Spinal Cord Gray Matter Segmentation CNN [35].

Light Detection and Ranging (LiDAR) for drivable region segmentation is another application that may use DCONVs to improve accuracy but with computational savings. For example, the segmentation results reported in Figure 2.8 refers to a piece of work in which parallel DCONV with R=1 and R=2 were used [36].

Figure 2.8 Drivable region segmentation results related to [36].

TCONVs find ground in tasks where image up-sampling is required. Generative Adversarial Networks (GANs) exploited for synthetic image generation adopt stacked TCONV Layers to build images, starting from a latent representation of data. The generation of bedrooms images was treated in the key work of Deep Convolutional GAN (DCGAN) models [22]. Figure 2.9 illustrates some generated samples.

Figure 2.9 Generated images through the DCGAN model [22].

Super-Resolution imaging is gaining ever-increasing interest due to the development of Ultra High Definition (UHD) and Quad High Definition (QHD) imaging. The Fast Super-Resolution CNN (FSRCNN) [31] is a model consisting of an encoder of stacked CONV Layers, followed by a decoder consisting of one TCONV Layer, which up-sample images by 4×, 9×, 16× the original image area. Figure 2.10 shows an example of super-resolution, where the original image is up-sampled by adding 9× more pixels, and comparing the achieved results in terms of the Peak Signal-To-Noise Ratio (PSNR) metric with respect to other super-resolution approaches. This method provides the best quality results by the metric viewpoint.

Figure 2.10 Examples of Super-Resolution images [31].

2.4. EFFECTIVE HARDWARE IMPLEMENTATION OF CNNS

2.4.1. OVERVIEW OF ARCHITECTURES FOR CNN WORKLOADS

CNN workloads can be carried out by two classes of architectures: *temporal architectures* and *spatial architectures*.

Temporal architectures mainly rely on Central Processing Units (CPUs) and Graphics Processing Units (GPUs) to execute models through software routines. CPUs adopt Arithmetic Logic Units (ALUs) that exchange data from/to a memory support, but they are not able to move information towards homologous ALUs [37]. This severely limits the speed performances achieved by CNNs. To address this issue, GPUs exploit

parallel processing engines that are suitable for computer graphics problems, such as image processing.

However, GPUs consume hundreds of Watt to meet the performance requirements of CNN workloads. This challenge may be addressed by spatial architectures [37], implemented through dedicated hardware acceleration, in which several Processing Engines (PEs) are able to mutually exchange data, other than relying to both centralized and distributed memories for provisional buffering purposes. Solutions based on Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs) are meaningful examples.

Each ASIC circuit is designed to perform limited tasks using a very specialized circuit. Indeed, the designer properly interconnects the needed resources to minimize the area occupation, thus providing the best speed-power trade-off. Conversely, FPGAs consist of reconfigurable cells that can be hardware programmed, to comply with different tasks by using the same architecture. This flexibility ensures wider applicability with respect to ASICs, but at the cost of lower performance and higher power dissipation. Indeed, despite the attempt of design tools to proper interconnect resources to guarantee a successful speed-power compromise, several resources may be unused, thus contributing in higher static power consumption.

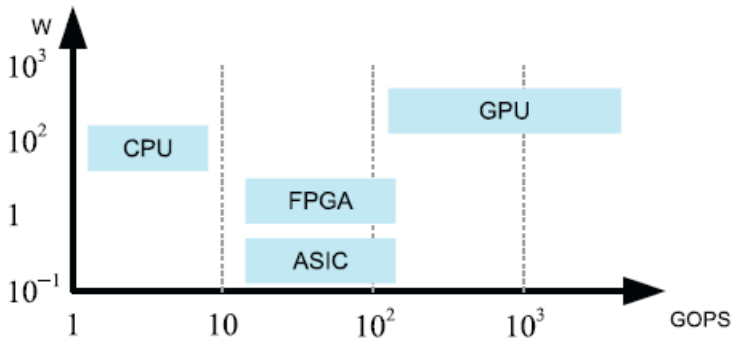


Figure 2.11 Trade-off analysis among CPUs, GPUs, ASICs and FPGAs [38].

Figure 2.11 illustrates the trade-off trend considering the above architectures [38]. The horizontal axis reports the speed performance, expressed in terms of Giga Operations Per Second (GOPS), while the vertical axis reports the power dissipation in Watt (W). It is easy to observe that hardware accelerators show the most favorable compromise.

ASICs beat FPGAs in terms of power dissipation, but if we consider the flexibility as a further design knob, FPGAs dominate the scenario.

To further investigate the suitability of FPGAs, we refer to an example scenario that evaluates their speed performance and energy efficiency with respect to GPUs. This analysis is worthy, considering that CONV Layers occupy about 90% of computation time in conventional CNNs [39]. Figure 2.12 refers to matrix multiplications for the pruned AlexNet model implemented over different Intel FPGA devices [40]. Experiments were also performed on a Titan X GPU, where the dense AlexNet counterpart was considered. For what concerns the speed performances, expressed as Tera Operations per Seconds (TOP/s), limited FPGA frequencies result in worse results with respect to the GPU execution. However, moving towards high-performance devices, the result flips over. Conversely, the energy efficiency is always better, even using more conservative frequencies.

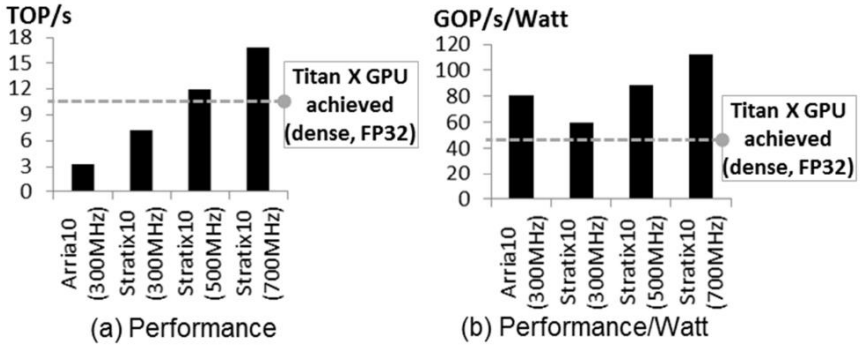


Figure 2.12 Speed performance and power efficiency comparisons between FPGAs and GPUs considering an example model [40].

Overall it can be concluded that FPGAs are the most suitable choice to implement in hardware CNNs, since they offer high flexibility, parallelism, as well as noticeable trade-off in terms of speed and power, and limited design cost. Table 2.2 compares the mentioned features by considering both temporal and spatial devices. Cells with a tick indicate that the given property is satisfactory. By a quick glance, it can be easily deduced that FPGAs positively meet all the requirements. Among the reported properties, it is worth mentioning the impact of design cost. It refers to the time frame between the design of the model and its availability on the market. Obviously, CPUs and GPUs exhibit the lowest design cost, in that it involves the realization of the software routine only. Conversely, the design effort of hardware accelerators is much higher, since it involves

the conceptualization, realization and implementation of the testing circuit. With respect to ASICs, the design cost of FPGAs is much lower, in that it requires the design, synthesis and implementation using proper tools, with no physical realization of the chipset.

Table 2.2 Comparisons of architectures for CNNs

	CPUs	GPUs	ASICs	FPGAs
Flexibility	☑	☐	☐	☑
Parallelism	☐	☑	☑	☑
Speed	☐	☑	☑	☑
Power	☐	☐	☑	☑
Design Cost	☑	☑	☐	☑

2.4.2. FIELD-PROGRAMMABLE GATE ARRAYS

FPGAs are integrated circuits consisting of reconfigurable logic, other than special resources for fast computations and on-chip memorization. In what follows, the terminology refers to the AMD Xilinx FPGAs, but with no loss of generality about the meaning of the presented resources. Figure 2.13 illustrates a top-level view of a AMD Xilinx FPGA [41].

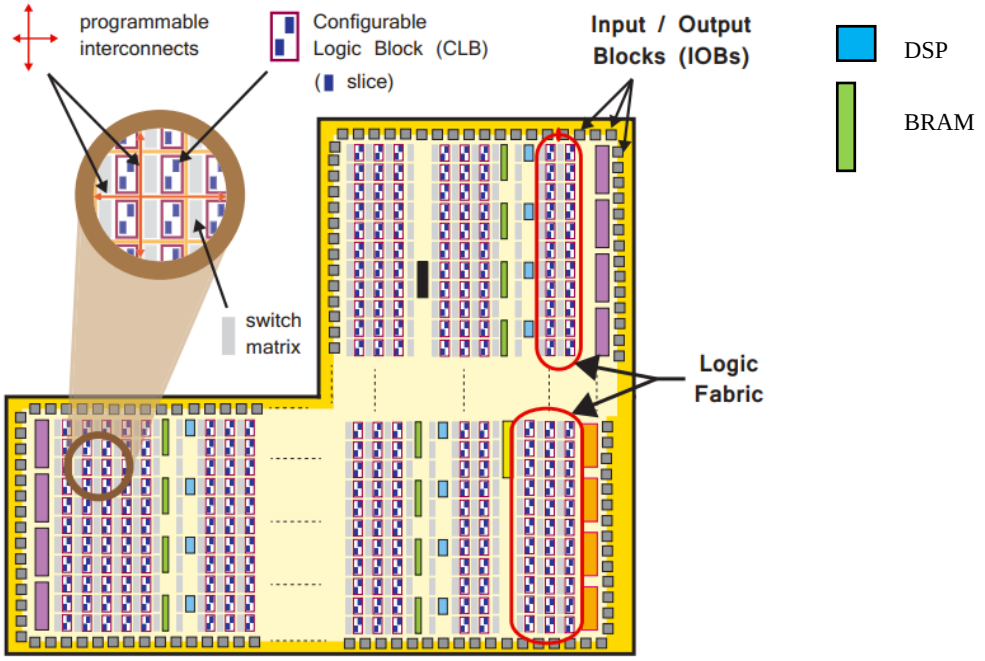


Figure 2.13 Top-level architecture of a FPGA [41].

Logic fabric mainly consists of *Configurable Logic Blocks* (CLBs) that are arranged as a 2-D matrix within the device. They are interconnected each other by means of

programmable interconnects provided by *switch matrices*. Each CLB contains *Slices* that, in turn, accommodate resources for combinatorial computations and sequential logic, namely *Look-Up Tables* (LUTs) and *Flip-Flops* (FFs), respectively. The former can be used to implement Boolean functions, as well as small Read Only Memories (ROMs) and shift-registers. The latter are 1-bit registers. Additionally, slices offer dedicated resources to implement efficient carry logic for accumulations.

Special resources, named *Digital Signal Processors* (DSPs), are also offered by modern FPGAs to perform high-performance Multiply-Accumulations, as well as to implement dedicated paradigms such as the Single-Instruction-Multiple-Data (SIMD). Furthermore, *Blocks of Random Access Memories* (BRAMs) are laid out throughout the FPGA to offer on-chip storing, as well as the possibility to emulate First-In-First-Out (FIFO) buffers. Finally, *Input/Output Blocks* (IOBs) offer an interface between the given FPGA and other peripherals that may be connected to the device to exchange data for further processing and control.

Since 2011, the flexibility of FPGAs has been associated to the simplified design effort of general-purpose CPUs to provide users with heterogeneous architectures. The latter, also known as FPGA-based Systems-on-Chip (SoCs), consist of two main parts:

(a) The Programmable Logic (PL) that is the FPGA device.

(b) The Processing System (PS) that accommodates a general-purpose processor.

FPGA SoCs may be exploited to implement complex systems, where the Hardware/Software Co-Design is mandatory. Indeed, while the PL is responsible to accelerate the time-critical section of a system, the PS may be delegated to perform low-performance sections, as well as to provide the overall control of the architecture.

2.4.3. LANGUAGES FOR HARDWARE DESIGN

As previously stated, FPGAs can be hardware programmed. This is accomplished by conceiving sketches of code using two possible ways:

(a) The adoption of Hardware Description Languages (HDL), such as the Very High-Speed Integrated Circuits HDL (VHDL). The VHDL allows the designer to build a hardware model by means of entities. Each *entity* is described through an *architecture* that explains the functionality of that entity. Different design approaches may be exploited to conceive an architecture, ranging from a dataflow style to a behavioral

model. While the former explain the computations by means of Boolean logic, the latter adopts higher-level constructs, such as *processes* with conditional statements.

(b) The use of High-Level Synthesis (HLS). The circuit is modelled by a high-level routine, typically in C or C++, which describes the behavior of the architecture. Proper directives, named *#pragmas*, may be exploited to instruct the synthesis tool about specific hardware optimizations to be applied (e.g., parallelism, insertion of sequential logic for pipelining).

Both the approaches lead to a Register-Transfer-Level (RTL) representation of the circuit, which is later synthesized and implemented by the development tool. At the completion, a bitstream is generated and forwarded to the development board for hardware testing.

2.4.4. METRICS RELATED TO FPGA ACCELERATORS

In order to easily follow the discussions about the hardware implementation of CNNs using FPGAs (especially in Chapters 3 and 4), the main metrics used for evaluation are introduced here.

- *Resource utilization* or *resource occupation*. This metric refers to the number of LUTs, FFs, BRAMs and DSPs used to implement a given circuit. These values can be absolute numbers or percentages. While the former allow comparisons even considering different FPGA devices, the latter are suitable for comparisons using the same device.
- *Latency.* This metric refers to the time needed to execute a given task. For sequential circuits, the clock period (T_{clk}) is the time unit. Accordingly, latency can be expressed either as the number of clock cycles or as the absolute time (i.e., clock cycles × T_{clk}).
- *Clock frequency.* It is related to the clock period T_{clk}, as $f_{clk} = 1/ T_{clk}$. It can be used as measure of speed performance.
- *Power dissipation* or *power consumption*. This metric refers to the power dissipated by the FPGA, and expressed in Watt. Power dissipation mainly consists of dynamic power and static power. While the former is related to the switching activity of the system at run-time, the latter provides the contribution of leakage components.

For what concerns the speed performance, some metrics related to the fields of image processing and CNNs are also reported:

- *Operations per Second* (OPS). Given a CNN, this metric indicates the number of operations that the hardware model is able to carry out in one second.
  The *theoretical OPS* or *peak OPS* are related to the computational capability of the circuit. To be more specific, if the hardware accelerator is able to perform *NOP* operations in parallel, thus the peak $OPS = NOP / T_{clk.}$
  The *effective OPS* are specific of the given CNN. Indeed the latter metric is given by the ratio between the computational complexity of the model and the execution time. Usually, OPS are reported using multiples. In CNNs acceleration, it is usually to refer to *Giga OPS* (GOPS).
- *Frame rate*. This metric indicates the number of images processed in one second. In fact, it is usually indicated as *frames-per-second* (*fps*).
- *Energy efficiency*. This metric measures the ratio between the speed performance and the power dissipation. Usually, the energy efficiency is measured in *GOPS/W* or *fps/W*.

2.4.5. SUMMARY OF FPGAS FEATURES FOR CNNS ACCELERATION

The analysis reported in this Section highlighted the suitability of FPGAs to implement efficient CNN workloads. The following remarks summarize the main findings and provide insightful comments about the challenges that will be treated throughout this thesis.

- The FPGA logic mainly consists of reconfigurable cells able to infer flexibility to a given design. Specifically, a CNN engine can be conceived to support different parameters combinations at run-time. For instance, a CONV Layer can be equipped with programmable kernel size.
- Dedicated high-performance computing resources, named DSPs, can be exploited to implement effective computing patterns close to each other to reach high clock frequencies. For example, MACs can be executed by tiles of DSPs without relying on further carry logic within conventional Slices.
- On-chip memories can be used to buffer temporary data in order to reduce the accesses to an external memory support. This reflects on faster processing time and reduced dynamic power dissipation.

- State-of-the-art FPGAs are usually integrated within heterogeneous Systems-on-Chip, where the programmable silicon cooperates with general purpose processors. In this way, while the former takes under consideration timing-critical patterns, the latter can perform low-throughput computations at the same time.

2.5. STATE-OF-THE-ART FPGA ACCELERATORS FOR CNNS

Stimulated by the previous considerations, the deployment of CNNs in FPGA is nowadays a key research focus. In the following, some relevant works are briefly reviewed.

Authors in [42] proposed a reconfigurable coprocessor to be adapted to different CNNs at run-time. Temporary inputs and outputs are cached using on-chip memories, while computations are performed through a 1024 MACs-based convolution engine. Thanks to a complex control unit, the given architecture is made able to support up to 256×256 images and 16×16 filter kernels. The programmable circuit, implemented within the Xilinx Virtex-7 XC7VX485T FPGA, exhibits a throughput of 129.7 GOPS when executing a 5-layered CNN. However, the given throughput costs 18 W of dynamic power consumption by using only the 25.8% of LUTs and the 37% of DSPs for high-performance computations. It is worth underlining that the above power comes from the dc current consumption of the VC707 board, which contains the referred FPGA part among other peripherals.

In order to boost the speed performance, but limiting the resource occupation and the power dissipation, the Single-Instruction-Multiple-Data (SIMD) paradigm may be exploited in hardware accelerators. For example, it is examined in [43], where DSPs are made able to double the number of MACs, by accommodating two multiplications in parallel. As a consequence, the buffering resources also manage this intrinsic parallelism. The designed circuit is implemented within a complete heterogenous system, where a general-purpose processor controls the overall process. 3rd parts circuitries are exploited to manage the high-throughput data movement from/to an external memory, using the Direct Memory Access (DMA) paradigm. When implemented within the Xilinx XC7Z045 part, the proposed system shows a throughput of 425.32 GOPS when

accelerating the VGG-16 model. Even using the 97.8% of the available DSPs, the overall design consumes only 3.15 W at the 167 MHz clock frequency.

While the previous pieces of work deal with the deployment of just a run-time reconfigurable circuit to implement any layer of a given CNN, the research in [44] examined the partition of the FPGA silicon into multiple processors to implement dataflow CNN layers using the High-Level Synthesis design flow. Each processor is responsible to execute one or more layers belonging to the model under test. The aim is to minimize the number of parallel processors, by guaranteeing a proper trade-off in terms of resources and speed. The suitability of the proposed strategy was evaluated over different models and FPGA parts. When the Xilinx Virtex-7 XC7VX690T FPGA and the SqueezeNet model [45] are considered, the system reaches a throughput of 909.7 GOPS, with a power dissipation of 7.2 W.

Hardware implementations dealing with TCONVs aim at avoiding redundant computations (i.e., multiplications by zero) due to the dilation of *fmaps*, either by revisiting the conventional algorithm sketched in Section 2.3.2 or by examining alternative strategies that lead to the same outputs. The FlexiGAN framework presented in [46] reorganizes processing rows to skip zeroed computations. Furthermore, it exploits the SIMD paradigm to enable high-performance computational patterns. The equivalent circuit, implemented within the Xilinx Virtex UltraScale XCVU13P, benchmarks state-of-the-art Generative Adversarial Networks (GANs) at the 190 MHz clock frequency.

Taking into account the different patterns of the actual computations through sliding windows, authors in [47] transformed the generic TCONV computations into multiple CONV computations, by reorganizing filter kernels. Indeed, wide up-sampling filters may be split into smaller sub-filters, before performing TCONVs as multiple conventional CONVs running in parallel. They implemented the equivalent hardware engine within a multi-processor architecture to execute the FSRCNN model [31] (Section 2.3.3). The proposed design choice allowed the system to reach a throughput of up to 2691 GOPS, by dissipating only 5.4 W using the Xilinx XC7K410T FPGA.

A completely different strategy was proposed in [48]. There, the Input-Oriented Mapping algorithm [49] was used to avoid redundant computations. Such an approach multiplies each input pixel by the relative $K \times K$ kernel, thus furnishing a block of $K \times K$ output products. It is worth noting that neighboring input pixels lead to overlapping output

blocks. With *S* being the supported stride, up to *K*–*S* overlapping rows and columns must be properly managed to provide the correct result. To avoid the overlapping management, the reverse looping was exploited. Instead of perform the computations considering the input space, the output space is evaluated to determine which input blocks must be used to get a specific output value. A GAN network was tested within the Xilinx XC7Z020 FPGA SoC, by exhibiting a throughput of 2.6 GOPS at the 100 MHz clock frequency and using all the available DSPs slices.

Uni-OPU [50] is a full software/hardware stack able to provide a uniform architecture to support both CONV Layers and TCONV Layers, as well as the nearest-neighboring up-sampling strategy [51]. The latter acts similarly to TCONVs, but uses actual pixels replicas instead of zeros. The referred architecture was integrated within the Xilinx XC7Z100 FPGA SoC to accelerate several models, including GANs and the FSRCNN at the 200 MHz clock frequency and using the 1987 DSPs (98.37% of the total) for high-performance computations. Specifically, when the circuit manages TCONVs, it manifests a throughput of 2350 GOPS by consuming only 2.89 W of dynamic power.

Efficient implementation of DCONV-based neural network were also examined in the last years. In [36], the ChipNet architecture was proposed to manage real-time drivable region segmentation using a CNN having R=1 and R=2. The circuit was implemented within the Xilinx Ultra Scale XCKU115 FPGA. The 350 MHz clock frequency and the utilization of the ~56% of DSPs allow the engine to complete the processing in about 12.59 ms, outperforming the Nvidia K20 GPU by 13 times, and dissipating about 9.7 W of dynamic power.

In [52], a flexible accelerator was proposed to deal with both DCONV and TCONV layers, as well as conventional CONV layers. Authors proposed the sparsity-alike processing method to handle the dilation factor of kernels effectively, without wasting redundant computations. This strategy comes from the observation that non-zero locations within DCONV and TCONV follow a regular pattern, thus being predictable. The proposed system, implemented within the Intel Arria 10 SoC, runs at 200 MHz and uses only the 36% of DSPs. When the ENet model [53] is accelerated, it exhibits a throughput of 200.3 GOPS.

Table 2.3 summarizes the main features of the state-of-the-art pieces of work presented in this Section. For each reference, the table reports the FPGA used, a short

heading about the key feature of the design, the implemented CNN, as well as performance metrics.

Table 2.3 Comparisons of performance of state-of-the-art CNN accelerators on FPGAs

Ref.	FPGA	Key Feature	CNN	f_{clk} [MHz]	GOPS	Power [W]	GOPS/W
[42]	XC7VX485T	Run-time reconfigurability	Custom	150	129.7	18	7.2
[43]	XC7Z045	MACs using SIMD	VGG-16	167	425.32	3.15	135
[44]	XC7VX690T	Multi-processors FPGA	SqueezeNet	170	909.7	7.2	126.3
[46]	XCVU13P	TCONVs through zeros skipping	GAN	190	NA[1]	NA	NA
[47]	XC7K410T	Transforming TCONVs into CONVs	FSRCNN	130	2691	5.4	500.2
[48]	XC7Z020	TCONVs using the Input-Oriented Mapping	DCGAN	100	2.6	NA	NA
[50]	XC7Z100	TCONV and Nearest-Neighboring up-sampling	GAN	200	2350[2]	2.89	813.1[2]
[36]	XCKU115	DCONV with R=1,2	ChipNet	350	NA	9.7	NA
[52]	Arria 10	Adaptive architecture for CONV, DCONV, TCONV	ENet	200	200.3	NA	NA

[1] NA = Not Available. [2] Related to TCONV layer only.

The background presented in this Section presented relevant state-of-the-art FPGA solutions to provide a basic understanding of the challenges related to the hardware acceleration of CNNs. Chapters 3 and 4 will provide further background, referring also to other architectures for fair comparisons in terms of resource occupation, speed and power.

2.6. SUMMARY

In this chapter, basic concepts about Deep Learning and Convolutional Neural Networks were introduced. Specifically, Dilated Convolutions and Transposed Convolutions were briefly presented to provide the proper background for the comprehension of the following chapters. Examples of applications for image processing were also provided. Then, the main architectures to accelerate CNNs' workloads were discussed, by highlighting the advantages of FPGAs with respect to CPUs, GPUs, and ASICs. FPGAs, indeed, show the most favorable trade-off in terms of resource

occupation, speed, power and flexibility to manage effectively different CNNs' configurations using the same circuit.

The hardware design methodologies, as well as the metrics used for evaluation were discussed, in order to facilitate the understanding of Chapters 3 and 4. Finally, some state-of-the-art FPGA accelerators were reviewed, by summarizing the design strategies and the key experimental results.

3. LOW-LEVEL DESIGN FOR CONVOLUTIONAL NEURAL NETWORKS

This chapter presents the design of several FPGA-based accelerators for CNNs, made compliant with DCONVs and TCONVs, and using the VHDL as Hardware Description Language. The integration of the referred engines within heterogeneous embedded systems is also discussed, in order to analyze real-life use cases. For each novel architecture, the characterization in terms of resources utilization, speed performance and power dissipation is provided, as well as proper comparisons with state-of-the-art FPGA accelerators.

3.1. HARDWARE ACCELERATION OF MULTI-RATE DILATED CONVOLUTIONS

3.1.1. BACKGROUND

As stated in Chapter 2, conventional CNNs combine both CONV and FC Layers to meet image classification. Specifically, the spatial resolution of the intermediate *fmaps* produced over the network is progressively reduced and a one-dimensional array is produced as the final output. However this computational flow does not comply with pixel-level classification (i.e., semantic segmentation). In fact, the latter requires robust spatial information and lower abstraction to perform dense predictions. To address this issue, the FC Layers are replaced with CONV Layers only [54]. The networks obtained in this way, known as Fully Convolutional Networks (FCNs), are able to output prediction maps instead of one-dimensional arrays. To strengthen the localization accuracy of different Regions-of-Interest (ROI), DCONV Layers can be exploited.

The latter, also known as *à-trous* convolutions, limit the abstraction of features by using up-sampled filters that keep the data resolution unchanged over consecutive layers. Moreover, DCONVs at different dilation factors R can be performed jointly to build relationship with ever more far pixels, as stated by the À-trous Spatial Pyramid Pooling (ASPP) approach [34] (introduced in Section 2.3.3). However, the *à-trous* convolutions required by the latter constitute a speed bottleneck that makes the use of efficient hardware accelerators desirable to reduce the execution time. Unfortunately, although the number of operations required to perform dilated convolutions with a $K \times K$ kernel does not vary with R, the hardware accelerators validated for traditional CNNs cannot be trivially adapted to the ASPP. Indeed, other than the management of computations, data buffering must be properly treated. For this reason, new designs have been recently proposed [36, 55, 56].

Drivable region segmentation using the Light Detection And Ranging (LiDAR) strategy is performed by the accelerator presented in [36]. The latter executes both conventional and dilated convolutions, with R=2, and takes advantage of high parallelism to achieve real-time performances. Autoregressive deep CNNs are executed using the architecture [55], whereas 1D DCONVs are equipped with data reuse and batching to manage temporal correlations in [56]. Considering the peculiarities of the referred proposals, it can be concluded that none of them is suitable to completely support the ASPP approach for image segmentation. Indeed, on the one hand, the architectures proposed in [55, 56] are conceived to perform 1D dilated convolutions and do not provide either the adequate computational capabilities or the appropriate data buffering architectures to move towards the 2D scenario. On the other hand, the design demonstrated in [36] supports only the dilation rate R=2 that is quite low for certain applications, such as biomedical imaging, that could require dilation rates up to R=24 [35].

3.1.2. THE PROPOSED ARCHITECTURE

The top-level architecture of our proposed *Dilated Convolution Engine* (DCE) [13] is illustrated in Figure 3.1. The *à-trous Buffer* reads data belonging to the generic *ifmap* as inputs and arranges different dilated convolution windows in parallel. Meanwhile, the *Weights Buffer* furnishes as many as kernel weights. The *Convolution Core* is fed by the

referred data and executes the required MACs. In order to produce the generic *ofmap*, the results obtained for the current *ifmap* must be accumulated to those obtained for the previous ones. The unit *BRAMs for Accumulations* exploits on-chip memories to both store the current results and to resume them for subsequent accumulations among the different *fmaps'* channels. All these operations are orchestrated by the *Control Unit* and repeated until the last *ifmap* is processed. When this event occurs, the module *ReLUs* infer the homonymous non-linearity to the provisional outputs, thus providing the final *Rofmaps* values. According to the ASPP method, *ifmaps* are also subjected to the Global Average Pooling [57] through the homonymous *Pooler*.

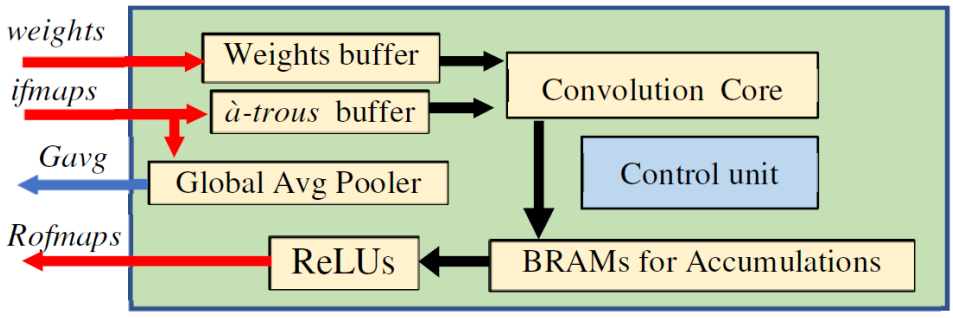
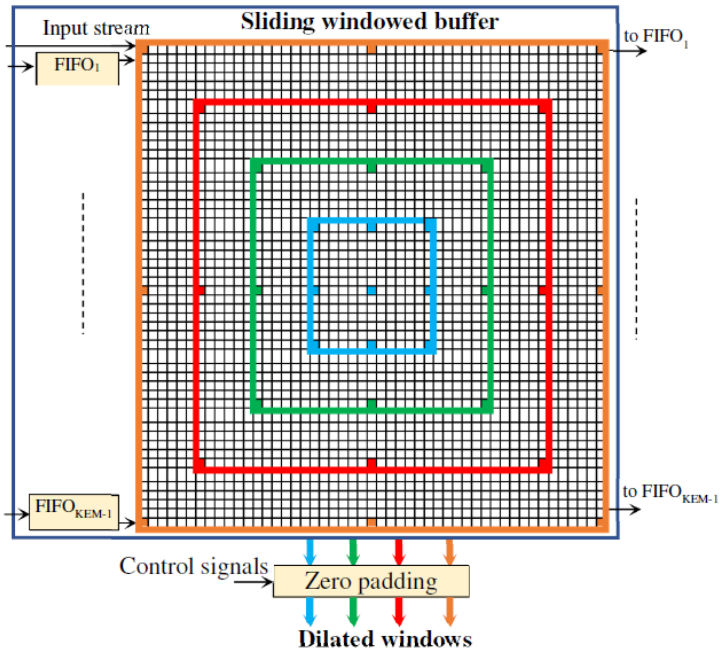


Figure 3.1 The proposed Dilated Convolution Engine (DCE).

The *à-trous Buffer* arranges inputs into M separate $K \times K$ dilated windows, each supporting a specific dilation rate, with M set at design time. Figure 3.2(a) schematize the latter when M=4. There, K_{EM} is the actual size of the maximum window given by $K_{EM} = K+(K-1)\times(R_M-1)$, with R_M being the maximum supported dilation factor. Specifically, the referred figure refers to K_{EM} = 49 considering K=3 and R_M=24. Inputs directly supply the *Sliding Windowed Buffer* as a stream of data. The latter consists of a register file, which accommodates up to $K_{EM} \times K_{EM}$ pixels, and K_{EM}–1 First-In First-Out memories (FIFOs) that locally store $W-K_{EM}$ data, with W being the width of each *ifmap*. Finally, the *Zero-Padding* module properly manages the border values taking into account that each dilated window may require a different padding size. Figure 3.2(b) details the VHDL description of the generic dilated window using parametric constructs, which ensures the correct arrangement of multiple windows of pixels through few code lines, other than the possibility to reuse this piece of code within other designs having different requirements in terms of kernel size and/or dilation factors. Specifically, this piece of code consists of

a *for generate* loop, through which the body is replicated *M* times (i.e., the *RATE_WIDTH* parameter), by providing as many windows of inputs for processing. The multi-dimensional array *win_int* collects the actual inputs provided by the dilated *win_in* array. The array parameter *R* contains all the supported dilation factors. The cells taken into account are expressed by the instances on the right, and following the patterns reported in Figure 3.2(a).



(a)

```
WI: for i in 0 to RATE_WIDTH-1 generate
    win_int(i)(0) <= win_in((KE*KE-1)/2-R(i)*(1+KE));
    win_int(i)(1) <= win_in((KE*KE-1)/2-R(i)*KE);
    win_int(i)(2) <= win_in((KE*KE-1)/2+R(i)*(1-KE));
    win_int(i)(3) <= win_in((KE*KE-1)/2-R(i));
    win_int(i)(4) <= win_in((KE*KE-1)/2);
    win_int(i)(5) <= win_in((KE*KE-1)/2+R(i));
    win_int(i)(6) <= win_in((KE*KE-1)/2-R(i)*(1-KE));
    win_int(i)(7) <= win_in((KE*KE-1)/2+R(i)*KE);
    win_int(i)(8) <= win_in((KE*KE-1)/2+R(i)*(1+KE));
end generate WI;
```

(b)

Figure 3.2 (a) The *à-trous* Buffer for *M*=4; **(b)** The VHDL code of the generic dilated window using parametric constructs.

The *Convolution Core* processes *M* sliding windows in parallel using as many as MAC_{KS} units through cascaded DSP slices. For instance, Figure 3.3 illustrates the MAC_{KS} engine when K=3, $KS=K^2$=9. It is worth underlining that the module MAC_9 is designed to efficiently exploit also the fast dedicated interconnections available on-chip to meet high-performance requirements [58].

With the input data and the kernel coefficients being *x*- and *y*-bit wide, respectively, the generic convolution produces a *z*-bit result, with $z = x + y + \lceil \log_2(K \times K) \rceil$. To limit the on-chip memory usage, each result is quantized to *x*-bit. Homologous results coming from the different dilated convolutions are packed within a single ($M \times x$)-bit word and locally stored within the *BRAMs for Accumulations*. Finally, the module *ReLUs* rectifies the *M* values by using simple multiplexing logic.

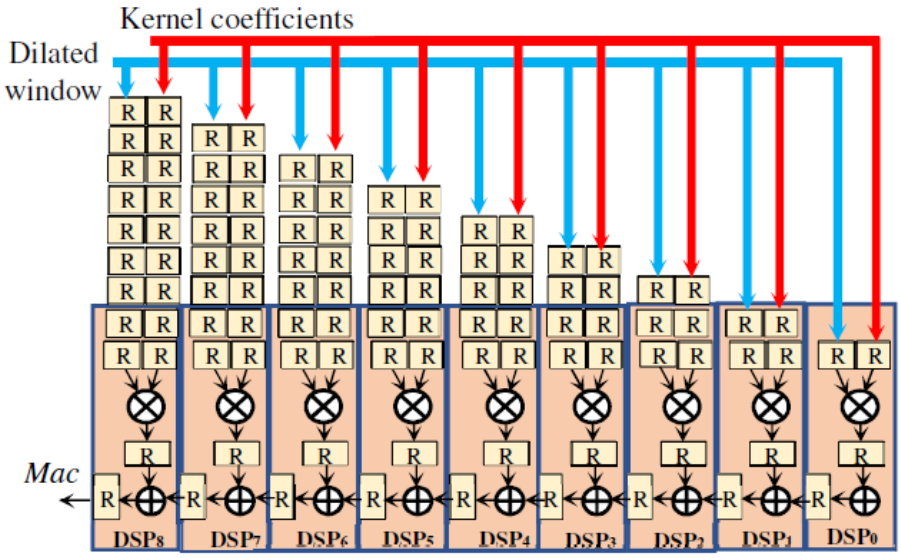


Figure 3.3 The MAC_9 module for K=3.

The novel DCE was purposely designed to comply with the fourth generation of the Advanced eXtensible Interface (AXI4) protocol [59]. The latter makes the circuit able to manage inputs and outputs as streams, other than being equipped with configuration registers that can be accessed by the general-purpose processor to configure the unit at run-time (e.g., start/stop of computations, beginning of new inputs to be processed). This capability is provided through the *Control Unit* visible in Figure 3.1 that manages the data transfers and orchestrates all the auxiliary operations required during the overall computation. While the *ifmaps*, the kernel coefficients, and the final *Rofmaps* are

transferred by AXI4-Stream transactions, the *Gavg* is managed by the AXI4-Lite interface.

3.1.3. EXPERIMENTAL RESULTS

As a case study, the proposed DCE was specialized to accelerate a model for biomedical image segmentation [35] reported in Figure 3.4. This neural network processes 32 200×200 *ifmaps*. The dashed box highlights the ASPP Layer, where all the units run in parallel. To support it, the DCE was configured with x=y=8, M=4 and R=6, 12, 18, 24. Validation tests were performed on the FPGA-based SoC shown in Figure 3.5 and implemented within the Xilinx XC7Z020 device [58]. There, the DMA units (third party cores) were used to upload both the *ifmaps* and the kernel coefficients from the external DDR memory and to stream them towards the DCE.

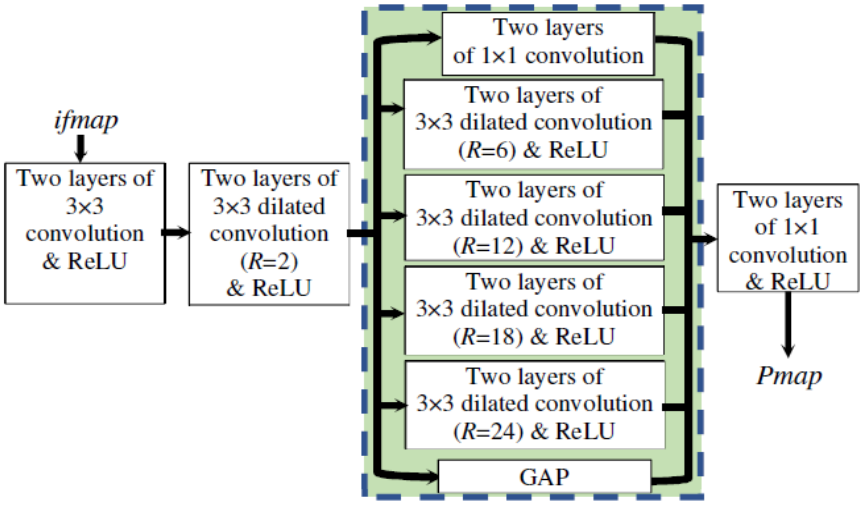


Figure 3.4 The deep CNN model referenced as a case study.

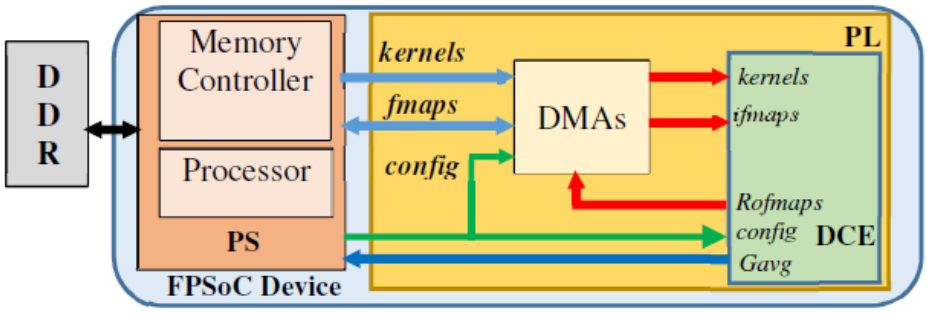


Figure 3.5 The referred embedded system architecture.

In turn, the latter moves the computed *Rofmap* towards the DMAs, which store the received results within the DDR memory. The software routine executed by the Processing System (PS) configures the DMAs by means of AXI4-Lite transactions.

Table 3.1 reports the post-implementation characterization in terms of (a) the model configuration (i.e., *ifmap* sizes, number of input channels N_C, number of *ifmaps* processed in parallel *Pf*, *M* and *R* as previously defined); (b) the design type (i.e., characterization of the single DCE as a standalone (SA) unit, characterization of the complete embedded system (ES)); (c) input arrays format (i.e., one-dimensional or bi-dimensional); (d) resource occupation (i.e., number of LUTs, FFs, BRAMs and DSPs used); (e) achieved clock frequency; (f) power dissipation in Watt. Figure 3.6 illustrates the percentage changes with respect to the counterparts. Even though a direct comparison with [55] and [56] is not feasible, since they process 1D data, the referred table would provide the reader with the "big picture". It is worth underlining that [36] and [55] are characterized as SA. Furthermore, the highly parallel architecture demonstrated in [36] was developed using the high-end Xilinx UltraScale platform [60], running at 350 MHz running with a latency of 0.036ms to provide couples of *ofmaps*. However, the maximum dilation rate supported is *R*=2 and the auxiliary modules responsible for the communication protocol are not taken into account either in terms of resources requirements or in terms of speed performances. The amounts of LUTs, FFs, BRAMs and DSPs required by the novel DCE are ~9.9×, ~14×, ~42×, and ~73× lower than [36]. This is motivated by the high input parallelism of [36] (i.e., 64 inputs in parallel), as well as the management of padding which requires several memory locations pre-loaded with zeros to emulate the behavior of borders. When compared to [55], the proposed DCE uses ~70× and ~13× less BRAMs and DSPs, but ~2.3× and ~5.6× more LUTs and FFs, respectively. Indeed, since [55] deals with 1-D data, it heavily exploits buffers for queue caches management to meet the data reuse requirement.

Despite the usage of the low-end Xilinx Zynq-7000 platform [58], our DCE exhibits a remarkable computational capability. Indeed, it runs at 181 MHz and processes the generic *ifmap* with the dilation rates 6, 12, 18 and 24 in only 0.25ms. Moreover, the very limited use of the available resources limits the power consumption to just ~265 mW. When compared to its all-software counterpart that runs on the 666.67 MHz dual-core

ARM processor within the SoC, the novel DCE exhibits a 648× speed-up. Indeed, 324 ms are needed to run the whole ASPP core by software.

Table 3.1 Characterization of the proposed DCONV accelerator and state-of-the-art comparisons.

Ref.	Device	Data structure, Parallelism Dilation Rates	Design	Data	Resources Utilization LUTs	FFs	BRAMs	DSPs	Frequency [MHz]	Power [W]
New	XC7Z020	*ifmap* size 200×200 N_C=32, Pf=1, M=4, R=6, 12, 18, 24	ES	2D	8745	8264	38.5	42	150	1.768[1]
New	XC7Z020	*ifmap* size 200×200 N_C=32, Pf=1, M=4, R=6, 12, 18, 24	SA	2D	3854	2363	36.5	42	181	0.265
[36]	XCKU115	*ifmap* size 64×180 N_C=64, Pf=64, M=2, R=1, 2	SA	2D	38082	33530	1543	3072	350	12.594
[55]	XCVU13P	-	SA	1D	1669	425	2580	540	150	23
[56]	XC7Z020	-	ES	1D	47489	26942	120	192	80	-

[1]Also the contribution of the Processing System is included.

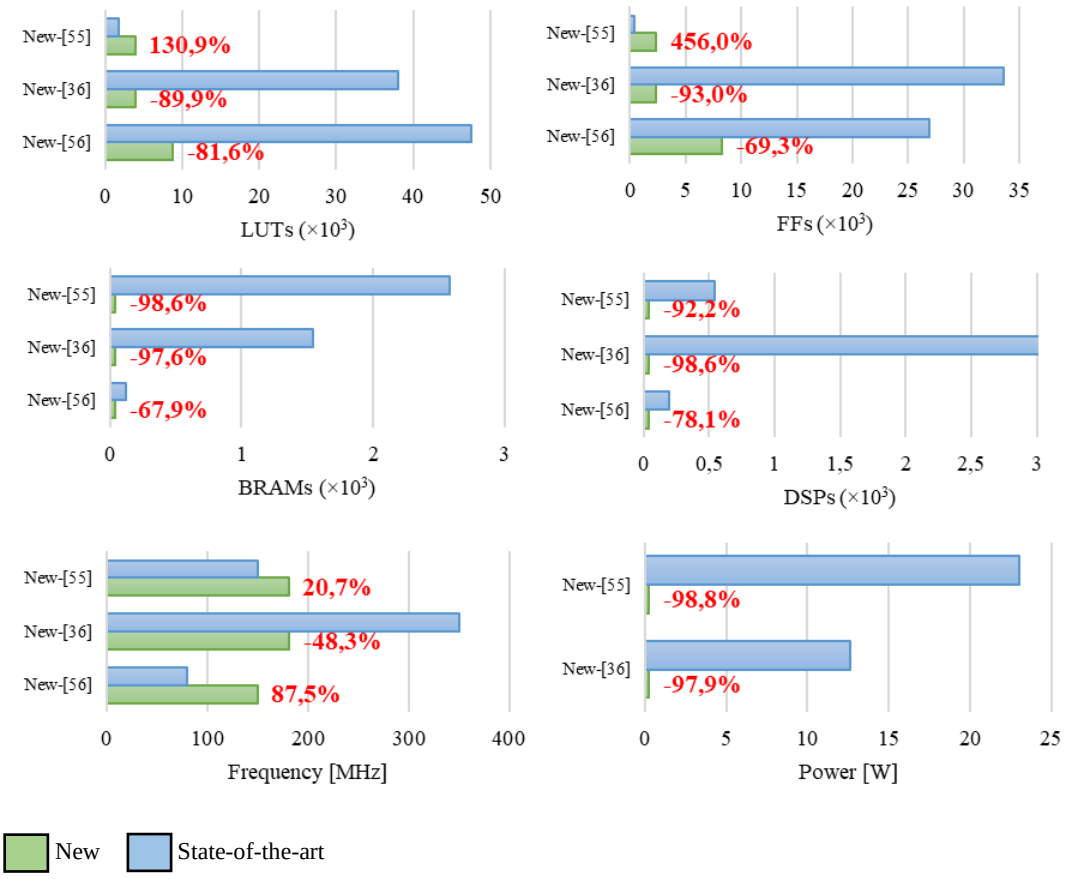


Figure 3.6 Percentage change comparisons: resources, frequency, power.

Table 3.1 also shows that the contribution of the auxiliary circuitry responsible for the communication protocol does not significantly affect the overall performances of the

whole ES. In fact, the maximum running frequency is reduced by only ~17% with respect to the SA implementation. In comparison with [56], apart the ~2.2× higher running frequency, the proposed ES exhibits much lower resources utilization.

Finally, it was estimated that, without significantly increasing the resources requirement reported in Table 3.1, the proposed ES is able to execute the entire biomedical imaging CNN model within a time ~244× lower than the pure software routine run by the ARM processor. Table 3.2 illustrates this performance estimation. The case Full HW reports the estimation when all the layers of the model [35] are performed by the FPGA. The case HW+SW refers to the case in which the ASPP only is accelerated by the FPGA. Finally, the case Full SW reports the estimation related to the full model executed on the general-purpose processor.

Table 3.2 Estimation of the execution time of the model [35].

Layers	Full HW [ms]	HW+SW [ms]	Full SW [ms]
CONV 3×3, *R*=1 (×2)	0.50	80.00 (*SW*)	80.00
CONV 3×3, *R*=2 (×2)	0.50	80.00 (*SW*)	80.00
ASPP (×2)	0.50	0.50 (*HW*)	324.00
CONV 1×1 (×2)	0.50	4.00 (*SW*)	4.00
Total	2.00	164.50	488.00

In order to evaluate the scalability of the proposed solution, the resources trend, varying both the *ifmaps* and the kernel sizes, was evaluated considering the XC7Z030 device [58] as benchmarking platform. Figure 3.7 shows that the number of DSPs, logic LUTs and FFs mainly depends on the kernel size. This is because the wider the kernel size, the wider the size of the *à-trous* Buffer, therefore the higher the number of registers as FFs. In addition, wider kernels mean more MACs to be performed by DSPs.

The amount of occupied BRAMs depends only on the *ifmap* size. Indeed, the module BRAMs for Accumulations adopts these on-chip storage resources to temporarily buffer provisional results, which depend in turn on the sizes of input images. Finally, the number of LUTs used as memory varies almost equally versus both the *ifmap* and the kernel sizes.

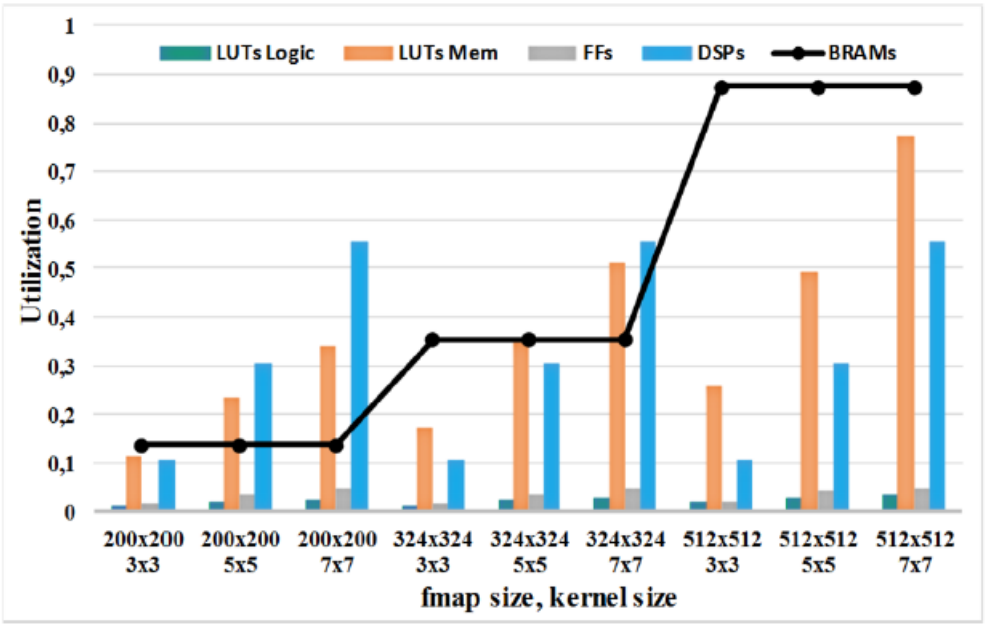


Figure 3.7 Analysis of resource utilization.

3.2. DESIGN OF AN EFFICIENT DECONVOLUTION ENGINE FOR FPGA-BASED SYSTEMS-ON-CHIP

3.2.1. BACKGROUND

Image segmentation [61], as well as object generation [62] and high-resolution imaging [63], can be interested by a cooperation between conventional CONV Layers and TCONV Layers (or Deconvolution Layers). In such a scenario, while conventional CONVs act like a down-sampler to compact the most relevant features, deconvolutions aim at predicting new values in order to furnish up-sampled outputs. Despite this difference, the key computations are performed similarly. Indeed, deconvolutions are nothing but CONVs executed on padded and strided inputs [64].

In the last years, the hardware acceleration of deconvolutional neural networks has gained ever more attention with the aim to minimize the number of redundant computations carried out by padding and striding (i.e., multiplications by zero). As an example, the conventional algorithm introduced in Chapter 2 was exploited in the FlexiGAN framework [46] to execute Generative Adversarial Networks (GANs) [65]. To

avoid redundant computations, data and filters are properly reorganized. As a drawback, this leads to unbalanced workloads, thus making additional control logic necessary and severely limiting the achievable overall performances.

The efficient design strategy presented in [50] overcame the above issues by performing a kernel conversion to calculate all the pre-addable weight combinations. Thus, a new set of filters is applied to the *ifmaps* to perform conventional convolutions. Such a strategy drastically reduces the computational complexity and introduces noticeable speed-up either over other FPGA accelerators or over GPU platforms.

Authors in [48] presented a design based on a completely different technique [49]. Specifically, each input pixel is multiplied by the respective $K \times K$ kernel, thus furnishing a window of $K \times K$ output products. It is worth underlining that neighboring input pixels can lead to overlapping output windows. In fact, considering the stride S, up to $K–S$ overlapping rows and columns must be properly managed to provide the final results. Unfortunately, the referred architecture does not manage the latter effectively. Indeed, in order to recognize no-overlapping blocks, it applies the reverse looping strategy that requires the computation of input coordinates at each filtering step, with penalties in terms of computational complexity and speed performance.

The accelerator proposed in [66] was purposely conceived to handle semantic segmentation, by means of a complex architecture to manage convolutions and deconvolutions separately. However, due to its hardware resources requirements, it is not suitable for low-end FPGA-based SoCs. In addition, while multiplications are performed efficiently, by only using DSPs, the additions required for row/columns overlaps are executed by means of configurable logic resources, thus limiting the clock frequency. This structure was further improved in [67], where both convolutions and deconvolutions are joined in just one engine to perform remote segmentation. The computing core consists of a MAC unit that operates in a serial manner, and supporting high level of parallelism for both *fmaps* and filters. Unfortunately, due to its high hardware resources requirements, this accelerator can be actually exploited only within high-end FPGA-based SoCs. Moreover, it is not a suitable solution to achieve the highest speed. Indeed, it requires up to 4 clock cycles to furnish each output pixel from a deconvolution, depending on how many overlapping pixels must be managed.

The solution presented in [68] deals with both 2D and 3D deconvolutions. A compression scheme was introduced to manage sparse activations and filter in order to reduce the computational redundancy. However, this strategy requires data to be encoded in coordinate format. This task involves the calculation of the output coordinates of each pixel, thus limiting the feasibility in continuous streaming-based heterogeneous embedded systems.

The proposals of [69, 70] exploited the effective Winograd algorithm to deploy Deconvolution Layers for GANs. Indeed, this method transforms CONVs into element-wise multiplications implemented by additions and shift operations. Despite the remarkable speed performances, also thanks to high parallelism, the Winograd algorithm introduces area and power overheads due to the pre- and post-processing operations to transform data back and forth the Winograd domain.

3.2.2. THE PROPOSED ARCHITECTURE

Among the several approaches previously discussed, we refer to the Input-Oriented Mapping (IOM) [49]. This algorithmic strategy considers a $H \times W$ *ifmap* and a $K \times K$ filter kernel, and takes into account the stride S as the up-sampling factor, which provides the ratio between the sizes of the final output image with respect to those of the original input image. The following conceptual steps are performed:

- the generic input pixel $I(i,j)$ is multiplied by the kernel and the resulting block of $K \times K$ products is properly arranged within the *ofmap* space by occupying the $K \times K$ area starting at the position $(i \times S, j \times S)$.
- Neighboring blocks, obtained from adjacent input pixels, have $K–S$ overlapping rows/columns, which must be summed up.
- Finally, P pixels on the *ofmap* borders are cropped, thus generating a $H_O \times W_O$ *ofmap*, with $H_O = S \times (H–1) + K – 2P$ and $W_O = S \times (W–1) + K – 2P$.

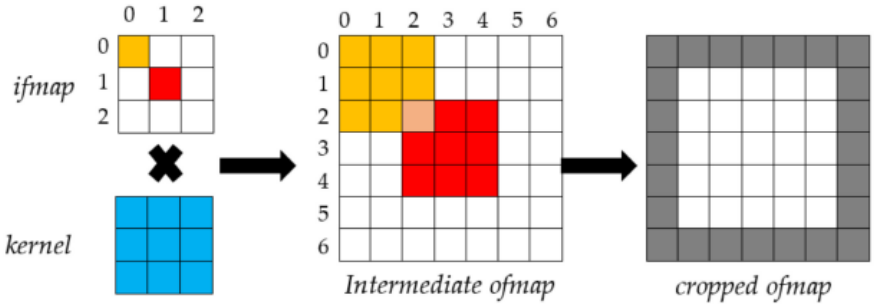


Figure 3.8 The adopted deconvolution approach.

The example depicted in Figure 3.8 shows an example that considers a 3×3 *ifmap* by using K=3, S=2 and P=1. For instance, the orange pixel *I(0,0)* within the *ifmap* leads to the 3×3 orange block of pixels starting at the location *(0,0)* within the intermediate *ofmap*. Similarly, by the red pixel *I(1,1)* within the *ifmap*, the 3×3 red block starting at the location *(2,2)* in the intermediate *ofmap* is obtained. And so on for all the other pixels. The overlapping pixels within neighboring blocks in the intermediate *ofmap* are summed up, as happens to the pink pixel in the location *(2,2)*. Finally, a 1-wide border (i.e., the grey border in the figure) is cropped to provide the final *ofmap*.

The top-level architecture of the novel accelerator [14], named *Deconvolution Layer Processing Element* (DLPE), is depicted in Figure 3.9. It is able to process T_N *ifmaps* (if_0, …, if_{TN-1}) and T_M filter kernels in parallel, thus providing T_M *ofmaps* (of_0, …, of_{TM-1}) at the same time. The engine also meets pixel-level parallelism. Indeed, the DLPE can receive P_N pixels from each *ifmap* and can furnish P_M results belonging to each *ofmap* in parallel, with $P_M=S\times S\times P_N$. It is worth underlining that T_N, T_M, P_N, P_M are set at design time.

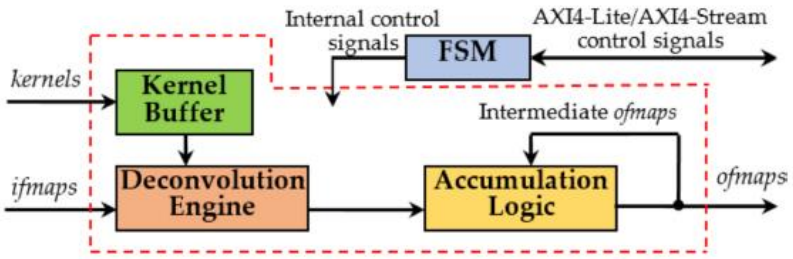


Figure 3.9 The top-level architecture of the Deconvolution Layer Processing Element.

The novel FPGA accelerator was conceived supposing that both *ifmaps* and kernels are stored within an external memory. These data can be uploaded and streamed towards the DLPE by means of auxiliary circuitries, such as the *Direct Memory Access* (DMA) and the *Video DMA* (VDMA) modules. While the F-bit pixels of the *ifmaps* are streamed-

in directly to the *Deconvolution Engine* (DE), the *N*-bit kernels coefficients are preliminarily locally stored within the *Kernel Buffer* and then provided to the DE, which is the core unit of the design. The *Accumulation Logic* (AL) adopts fast adder trees to accumulate the provisional results produced at the various computational steps and collected within on-chip memory resources until the last step is performed and the final *ofmaps* are generated. Indeed, the data parallelism parameters T_N and T_M strictly depend on the resources availability within the specific device chosen as the realization platform. Accordingly, when the number N_C of *ifmaps* and/or the number N_F of kernels to be processed are greater than T_N and T_M, respectively, the overall computation is completed within multiple steps. The latter, as well as the overall operations, are properly orchestrated by the *Finite State Machine* (FSM) unit, which also makes the accelerator AXI4 [59] compliant. Specifically, it takes care of managing the activities related to data transfers, including the AXI4-Stream transactions through which the kernels coefficients and the *ifmaps* are moved.

The *Kernel Buffer*, depicted in Figure 3.10, consists of a register file that stores $K \times K \times T_M \times T_N$ *N*-bit coefficients. During each clock cycle, the buffer is fed by the homologous coefficients (i.e., located at the same spatial positions) related to the T_N ifmaps $if_0, \ldots, if_{TN-1}$ and packed within one $T_N \times N$-bit word. Therefore, all the coefficients are provided to the DE within just $K \times K \times T_M$ clock cycles. The *Separate & Route* logic properly dispatches the coefficients to the DE. The latter is the computational element of the DLPE and, as shown in Figure 3.11, it consists of $T_M \times T_N$ *Deconvolution Units* (DUs) that work in parallel. At each clock cycle, the generic DU_{out}^{in} receives P_N adjacent input pixels $I(i,j), I(i,j+1), \ldots, I(i,j+P_N-1)$ from the *ifmap* if_{in}, with $in=0, \ldots, T_N-1$, and performs the deconvolution with the relative $K \times K$ kernel C_{out}^{in} as required to compute the *ofmap* of_{out}, with $out=0, \ldots, T_M-1$. The input pixels are multiplied in parallel by the coefficients of the kernel and P_N blocks of $K \times K$ products are computed at the same time.

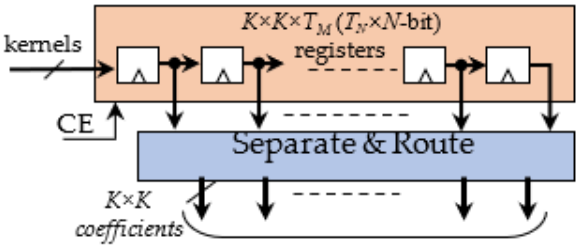


Figure 3.10 The Kernel Buffer.

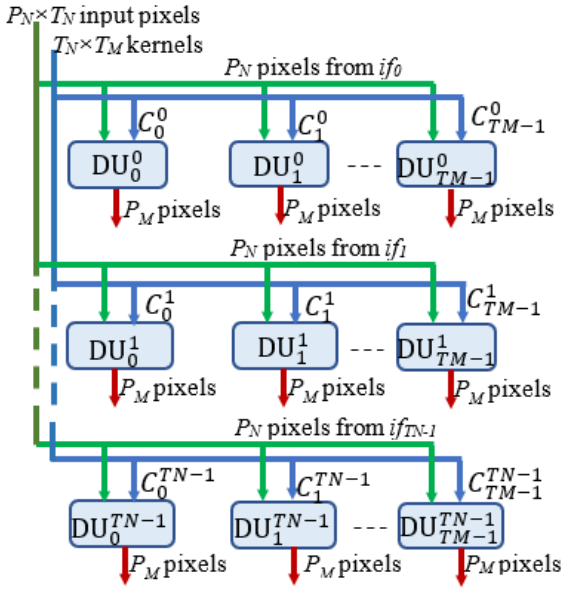


Figure 3.11 The architecture of the novel Deconvolution Engine.

Each DU was structured, as depicted in Figure 3.12, to manage efficiently the overlapping rows/columns between the neighboring blocks of products. Indeed, each DU consists of the *K* modules *Rowx*, with *x=0, …, K-1*, each using an appropriate number of DSPs, depending on the supported parallelism. In addition, to guarantee the proper time alignment of the overlapping products, First-In-First-Out (FIFO) *Buffers* are exploited.

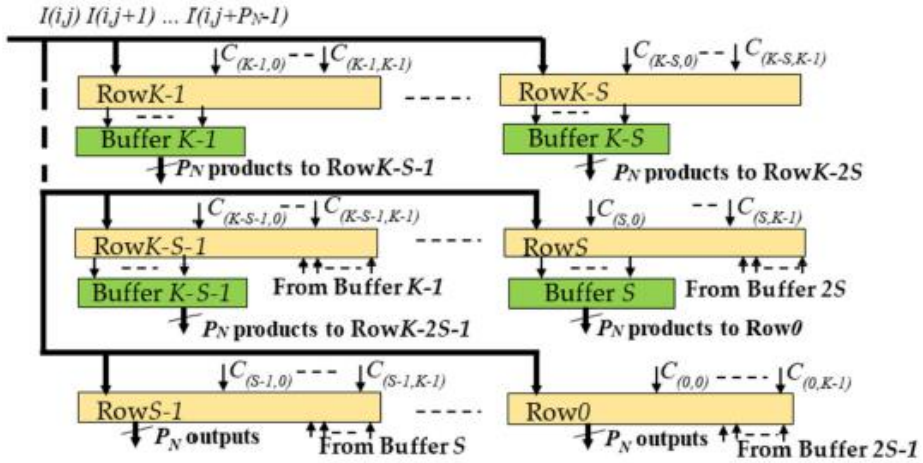


Figure 3.12 The structure of the generic DU.

To better explain how the generic DU performs deconvolutions, let us consider, as an example, the kernel size *K*=5, the stride *S*=2 and P_N=4. In this case, *x* ranges from 0 to 4 and five modules *Rowx* are required, each one, as reported in Figure 3.13, consisting of 20 DSPs. The latter are named *dy*, with *d=0, …, 3* and *y* ranging from 0 to 4, to indicate that they multiply the input pixel *I(i,j+d)* by the kernel coefficient $C_{(x,y)}$. The additional

DSPs *x0, …, x7* are required only within the modules *Row0*, *Row1* and *Row2* to manage the overlapping rows. Conversely, the $S \times P_N$ results computed by *Row3* and *Row4* are directly provided by the DSPs 00, 01, 10, 11, 20, 21, 30 and 31. All the multiplications and the additions performed by the generic DU are summarized in Figure 3.14 that also shows, for each entry, the related row and column indices within the intermediate *ofmap*. Since K–S=3, as highlighted by colored entries, each block of products computed by the DU has three columns and three rows overlapped with neighboring blocks.

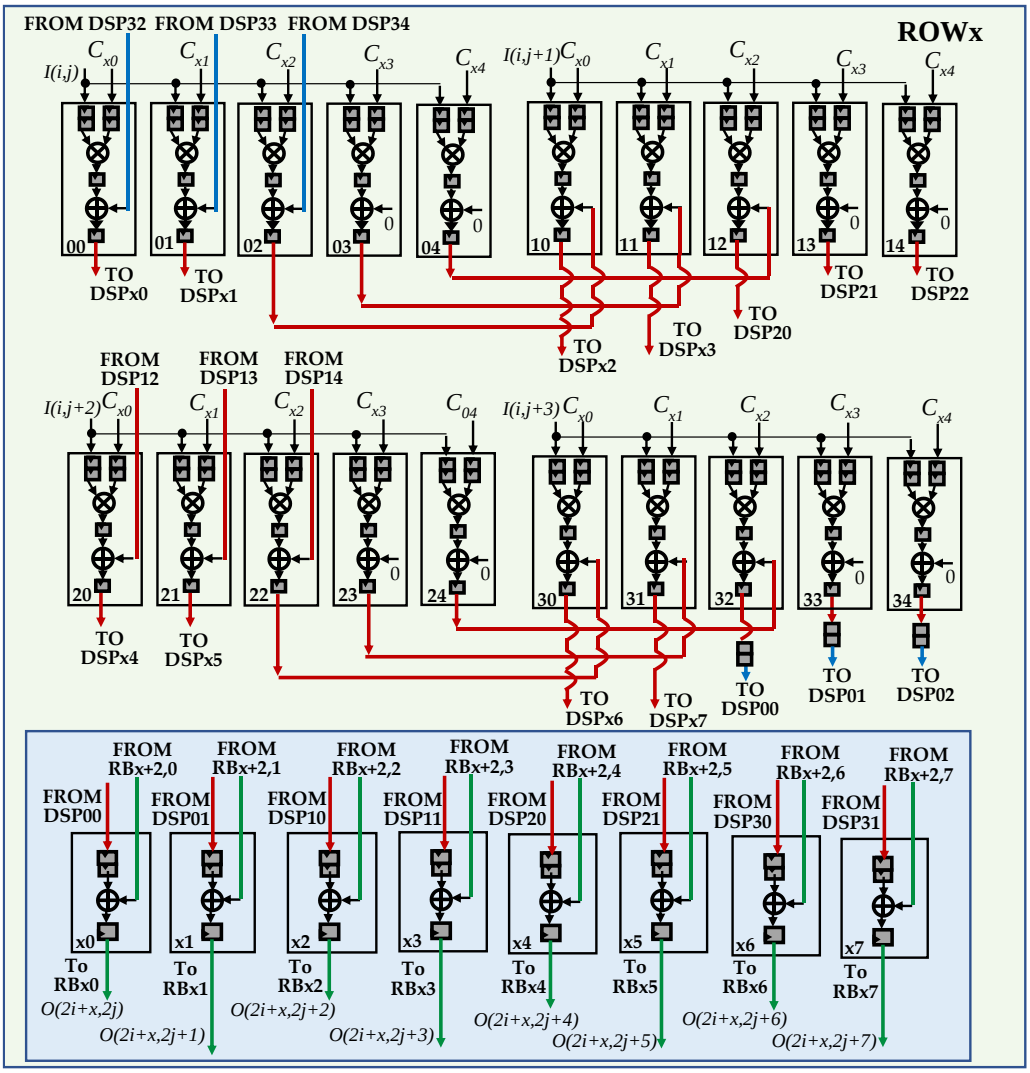


Figure 3.13 The architecture of the module *Rowx* for P_N=4, K=5, S=2.

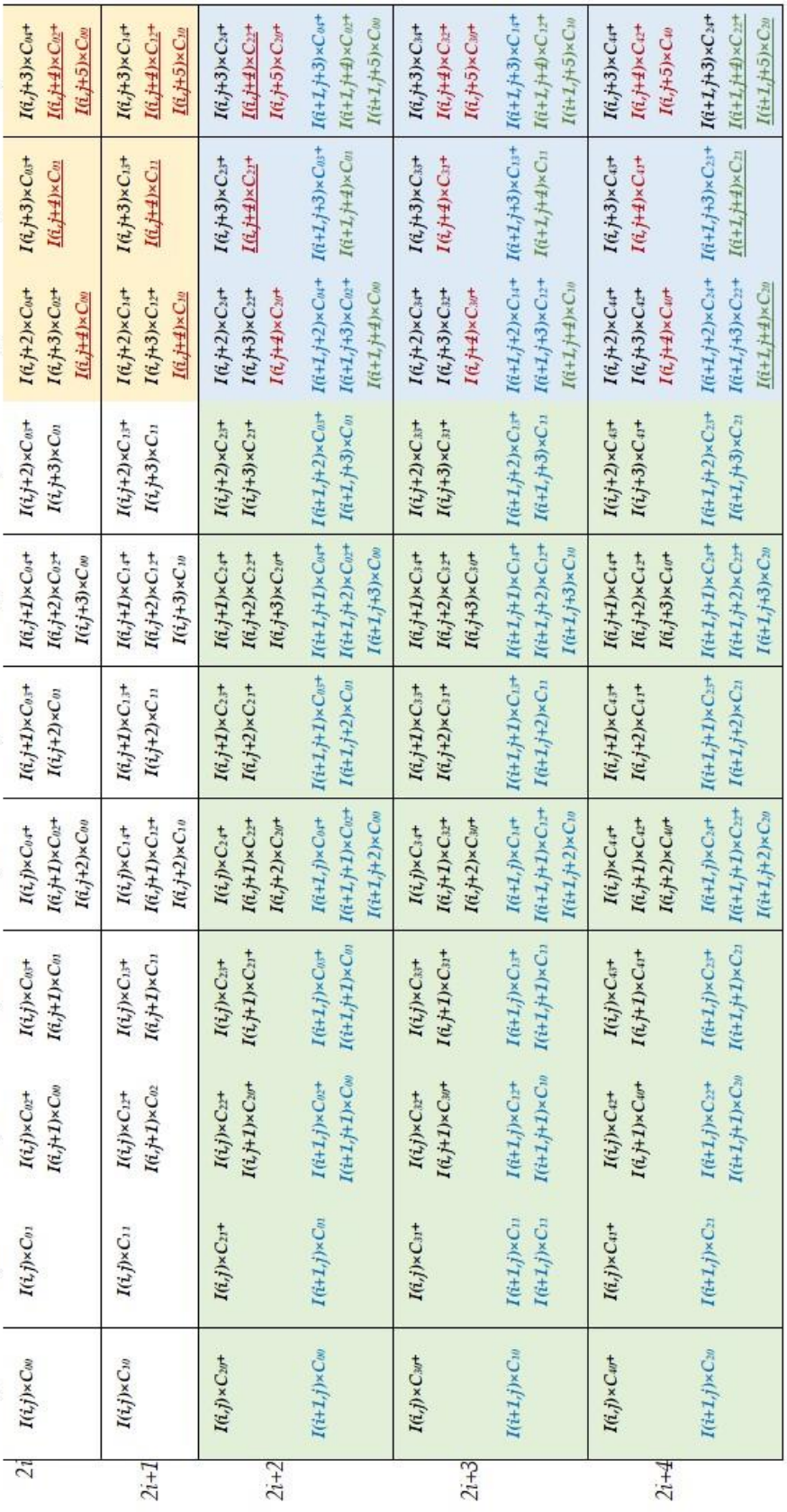
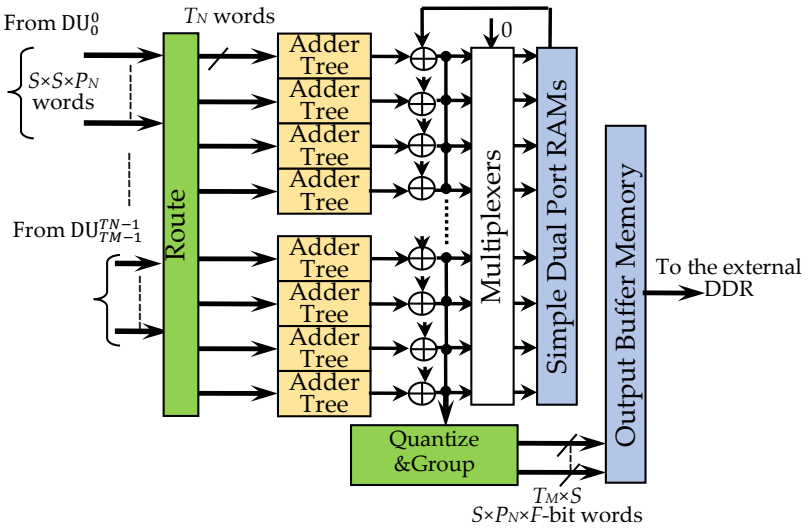


Figure 3.14 The operations performed by the generic DU when P_N=4, K=5, S=2.

It can be noticed that several kinds of overlapping products must be managed. The products related to the P_N adjacent input pixels, currently received by the DU, have the row index equal to *i* and the column index ranging between *j* and *j+3*. These products are reported in Figure 3.14 with black characters and their overlaps are managed through the red interconnections visible in Figure 3.13. Conversely, the products with red characters are computed at the next clock cycle, when the DU is receiving the next P_N adjacent pixels, i.e. *I(i,j+4), …, I(i,j+7)*, as inputs. The column overlaps related to these products are managed through the blue interconnections in Figure 3.13 to transfer the delayed outputs produced by the DSPs 32, 33 and 34 towards the DSPs 00, 01 and 02, respectively. Finally, the products with blue characters involve the pixels *I(i+1,j), …, I(i+1,j+3)*, which belong to the *(i+1)*-th row of the current *ifmap*. To receive these pixels as input, the DU waits until all the pixels of the *i*-th row are processed. As previously stated, appropriate *Buffers* are exploited to guarantee the proper time alignment of these overlapping products. In the referred example, they are required at the output of the three modules *Row2, Row3* and *Row4*. These overlapping products are managed within the modules *Row0, Row1* and *Row2* through the DSPs *x0, …, x7* and the green interconnections depicted in Figure 3.13. Thanks to the fully pipelined architecture, after the initial latency, each DU furnishes $S\times S\times P_N$ deconvolved pixels at every clock cycle. These deconvolved pixels are reported in the white entries of Figure 3.14 as provided by the modules *Row0* and *Row1*.

Resources requirements, latency and throughput rate of the DE depend on the *ifmaps* size $H\times W$, as well as on *K, S,* T_N, T_M and P_N. In the generic scenario, each DU needs $[K\times K+S\times(K-S)]\times P_N$ DSP slices to perform multiplications and to sum the overlapping neighboring products that are time aligned through $S\times(K-S)\times P_N$ *Row Buffers*, each being $\left\lceil\frac{S\times(W-1)+K}{S\times P_N}\right\rceil-2$ wide. The novel DE was designed also considering the management of border pixels. This is a key aspect, since it affects the data flow of the input streams. In fact, each of the $T_M\times T_N$ DUs, which operate in parallel, receives its own *ifmap* in the raster order. At the end of each row, the DE stops the incoming stream of pixels for $\left\lceil\frac{S\times(W-1)+K}{S\times P_N}\right\rceil-\frac{W}{P_N}$ clock cycles. During this time, the zero padding is applied through a proper multiplexing logic directly controlled by the FSM that also manages the AXI4 protocol signals. At the end of the current step, the DE provides $\left\lceil\frac{S\times(H-1)+K}{S}\right\rceil-H$

padding rows, before acquiring the next group of *ifmaps* to perform the subsequent computational step.



(a)

(b)

Figure 3.15 The Accumulation Logic **(a)** the architecture; **(b)** the *ofmap* arrangement for P_N=4 and S=2.

As previously stated, each DU furnishes $T_M\times T_N$ blocks of $S\times S\times P_N$ pixels. The homologous pixels within these blocks are accumulated to compose an intermediate *ofmap*. In turn, the intermediate *ofmaps* are accumulated step-by-step to each other until the DLPE provides the final result. The *Accumulation Logic* (AL) acts in this way and it is represented in Figure 3.15(a). The *Route* module receives blocks of deconvolved pixels from the DUs and sends them to $S\times S\times T_M\times P_N$ adder trees, considering that each group of T_N homologous data must supply the same *Adder Tree*. The latter adopts DSPs to execute accumulations. The intermediate *ofmaps* are temporarily stored within local *Simple Dual Port RAMs* (SDPRAMs). They are resumed later to be accumulated with the intermediate *ofmaps* produced at the next step. During the last computational step, the final

deconvolved pixels are generated. The *Quantize&Group* module quantizes the final deconvolved pixels to *F*-bit values and properly arranges them into packed words to be processed by the next layer, as required by the referenced model. These words are stored within the *Output Buffer Memory* to be then moved to the external DDR memory. In the meantime, a bank of *Multiplexers* drives SDPRAMs with zeros. In this way, the SDPRAMs are prepared for the next deconvolution task without wasting additional initialization time.

The example depicted in Figure 3.15(b) shows one 6×24 *ofmap* produced with S=2 and P_N=4. Different colors are used to highlight the pixels furnished in parallel at a certain clock cycle, as well as the different numbers refer to the clock cycle in which those pixels are provided. So that, as an example, all the pixels located at the magenta entries are furnished at the 4th clock cycle. To ensure that the final *ofmap* is stored within the external DDR in the raster order, each block of pixels can be arranged in two words, each containing the pixels within the same row. Hence, in the example, the *Quantize&Group* module would furnish two $2\times P_N\times F$-bit words at every clock cycle. In general, this module packs the final deconvolved pixels within $T_M\times S$ words, each being $S\times P_N\times F$-bit wide. Taking into account the example of Figure 3.15(b), and supposing T_M=2 and F=16, this means that the *Quantize&Group* unit packs the output pixels within 4 words, each being 128-bit wide.

3.2.3. EXPERIMENTAL RESULTS

Parametric VHDL constructs allowed the novel hardware accelerator to be customized to different operating conditions and high computational speeds to be achieved by carefully using the available resources. The circuit was designed and characterized using the Vivado Design Suite (v.2019.2) [71]. For purposes of comparison with existing competitors, the DLPE accelerated the DCGAN neural network [22]. To this aim, it was accommodated with the FPGA-based embedded system of Figure 3.16. Even though only implementations within Xilinx devices are detailed in the following, virtually any other devices family can be used for purposes of prototyping. The Programmable Logic (PL) accommodates the DLPE and all the auxiliary circuitry required to manage the data transfers from/to the external DDR memory, as ruled by the AXI4 communication protocols [59]. As detailed in the legend of Figure 3.16, different

colors are used to distinguish connections supporting memory-mapped transactions from data streams.

The supported parallelism level is dictated by T_M, T_N, P_M and P_N, which are properly set in accordance with the amount of resources available within the specific device chosen as the target implementation platform. As an example, using the low-end XC7Z020 Zynq device [58], with T_M and T_N being set to 2 and 3, respectively, P_M=4 and P_N=1 can be used. This means that $\frac{N_F}{T_M}$ pairs of *ofmaps* are computed, each within $\frac{N_C}{T_N}+1$ computational steps. For what concerns the other modules within the PL: 1) the DMA [72] is responsible for uploading the kernels coefficients; 2) the VDMAs [73] are responsible for resuming and storing the *ifmaps* and the *ofmaps*; 3) the AXIS Combiner [74] synchronizes the parallel input data within a single data stream, then fed to the DLPE; 4) eventually, the AXIS Broadcaster [74] separates the output pixels received in parallel from the DLPE depending on the *ofmap* they belong to. It is worth noting that the adopted data transfer policy allows the *ofmaps* to be directly arranged within the DDR memory in the raster order. Therefore, subsequent cascaded deconvolutional layers can process them, without requiring either complex management of the memory address space or expensive data reorganization.

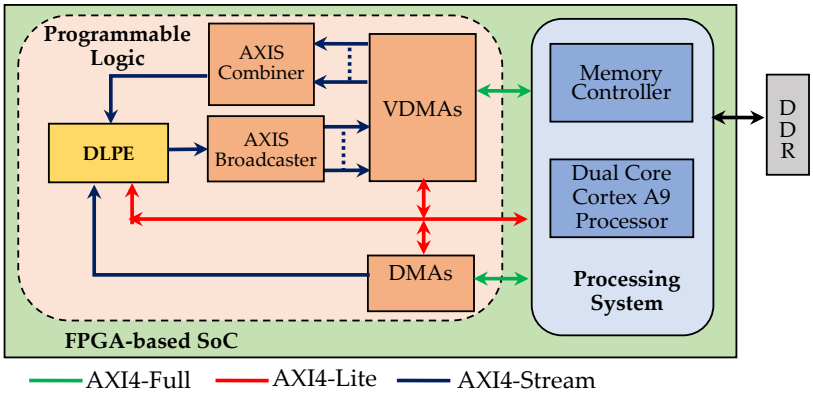


Figure 3.16 The referred embedded system architecture.

Figure 3.17 clarifies the computational flow of the DLPE with regards to the whole embedded system. During the first step, the processor configures the DMA to specify which off-chip memory area must be accessed to read a block of $K\times K\times T_M$ kernel coefficients. These coefficients are then streamed-in towards the DLPE to be stored within the *Kernel Buffer*. Meanwhile, the processor instructs the VDMAs to transfer

$H \times W \times T_N$ *ifmaps* from the off-chip memory into the DLPE. After a starting latency, the latter will produce the intermediate T_M *ofmaps* to be temporarily stored on-chip for further accumulations. The above operations are repeated for all the subsequent steps, until the last one is executed. Thus, the VDMAs are configured to transfer the final quantized T_M *ofmaps* to the external memory.

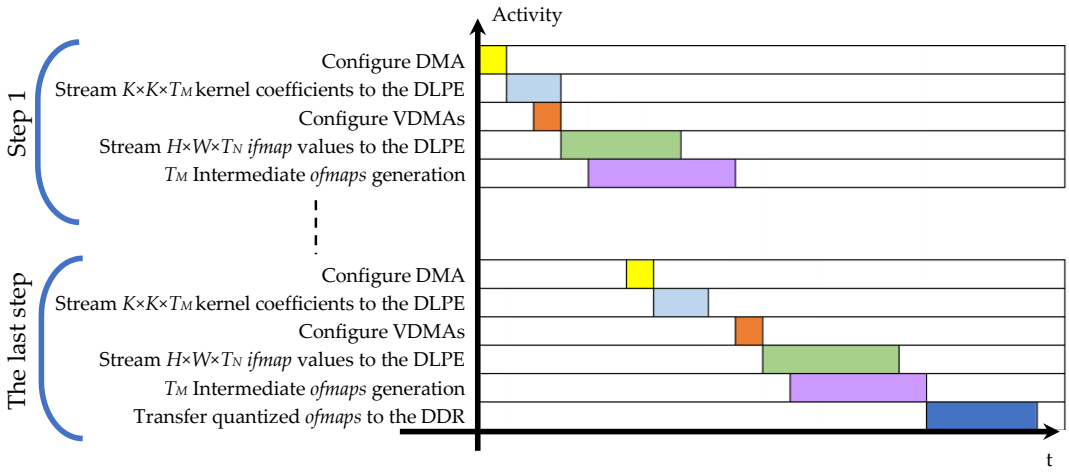


Figure 3.17 The computational flow of the whole architecture.

Table 3.3 presents the characterization of the proposed architecture using both low- and high-end devices for fair comparisons with state-of-the-art counterparts. The analysis reports: the supported parallelism (T_M, T_N, P_M and P_N), the kernel size (K) and the stride (S); the resources requirements; the running frequency; the number of giga operations performed per second (GOPS); and, finally, the dynamic power consumption. In addition, while the designs presented in [68, 69, 50] are standalone units (SUs), those demonstrated in [48, 75, 70] are embedded systems (ESs). For this reason, several SU and ES versions of the design here presented were implemented. Obviously, in comparison with the SU implementations, due to the auxiliary modules used to manage data transfers from/to the external DDR memory, the ES implementations occupy more LUTs, FFs and on-chip BRAMs, other than an increased dynamic power dissipation. Figures 3.18-3.19 report the percentage changes with the counterparts to easily follow the discussion.

The SU architectures presented in [68, 50] use K=3 and S=1 at high parallelism. Nevertheless, the design presented here, though it exploits a lower parallelism and operates with K=5 and S=2, which are more complex to manage, at a parity of the device used, reduces the amount of occupied LUTs, FFs, and DSPs by ~86%, ~90% and ~43%

with respect to [50]. Furthermore, it occupies 20× less BRAMs and reaches a 1.5× higher running frequency. When operating with $T_M=2$, $T_N=4$, $P_M=16$, $P_N=4$, $K=3$ and $S=1$, the resources requirements are further reduced and the consumed dynamic power is more than 45% lower than [50]. Similarly, when implemented within the high-end XC7VX690T device [76], the proposed design saves a significant amount of occupied resources with respect to [68].

Table 3.3 Characterization of the proposed architecture for deconvolutions and state-of-the-art comparisons.

	Device/ (Design, Precision)	T_M, T_N P_M, P_N K, S	**LUTs**	**FFs**	**BRAMs [Mb]**	**DSPs**	**Freq. [MHz]**	**GOPS**	**Dyn. Power [W]**
New	XC7Z020 (SU, 16b fix-p)	2, 3 4, 1 5, 2	2.9k (5.5%)	4.3k (4.1%)	0.84 (17.1%)	210 (95.5%)	200	72	0.42
New	XC7Z045 (SU, 16b fix-p)	2, 4 8, 2 5, 2	6.4k (2.9%)	9.6k (2.2%)	0.84 (4.4%)	560 (62.2%)	250	240	1.14
New	XC7Z100 (SU, 16b fix-p)	2, 4 16, 4 5, 2	15.5k (5.6%)	22.9k (4.1%)	0.84 (3.2%)	1120 (55.5%)	300	576	2.62
New	XC7VX690T (SU, 16b fix-p)	2, 3 32, 8 5, 2	23.2k (5.4%)	34.4k (4%)	0.84 (1.6%)	1680 (46.7%)	320	921.6	4.1
[69]	XC7VX485T (SU, 32b float-p)	4, 128 1, 1 5, 2	142.7k (47%)	151.4k (24.9%)	9.14 (25.2%)	2560 (91.4%)	100	NA	NA
[68]	XC7VX690T (SU, 16b fix-p)	2, 64 8, 1 3, 1	304.2k (70.2%)	602.7k (69.6%)	25.03 (48.4%)	2304 (64%)	200	1578	NA
[50]	XC7Z100 (SU, 16b fix-p)	64, 64 1, 1 3, 1	117.9k (42.5%)	247.2k (44.5%)	17.4 (65.5%)	1987 (98.4%)	200	NA	2.89
New	XC7Z020 (ES, 16b fix-p)	2, 3 4, 1 5, 2	12.8k (24.6%)	17.7k (17.1%)	1.49 (30.4%)	210 (95.5%)	150	54	1.73
New	XC7Z045 (ES, 16b fix-p)	2, 4 8, 2 5, 2	16.3k (7.5%)	23k (5.3%)	1.86 (9.7%)	560 (62.2%)	167	160.3	2.3
[48]	XC7Z020 (ES, 12b fix-p)	NA, NA 1, 1 NA, NA	25.5k (48%)	30.9k (29%)	2.35 (48%)	220 (100%)	100	2.6	NA
[75]	XC7Z045 (ES, 16b fix-p)	NA, NA 1, 1 5, 2	161.8k (74%)	148.6k (34%)	15.3 (80%)	810 (90%)	150	NA	NA
[70]	XC7Z045 (ES, 16b fix-p)	2, 2 4, 4 5, 2	196.7k (90%)	NA	10.9 (57%)	603 (67%)	167	133.8	5.8

[1] NA = Not Available.

Another consideration is about the high parallelism exploited in [68, 69, 50] at the *ifmaps* parallelism T_N. Indeed, ad-hoc memory managements are necessary to allow either 64 or 128 homologous pixels belonging to as many *ifmaps* to be accessed at the same time. To sustain this data access strategy, the designs presented in [68, 50] need

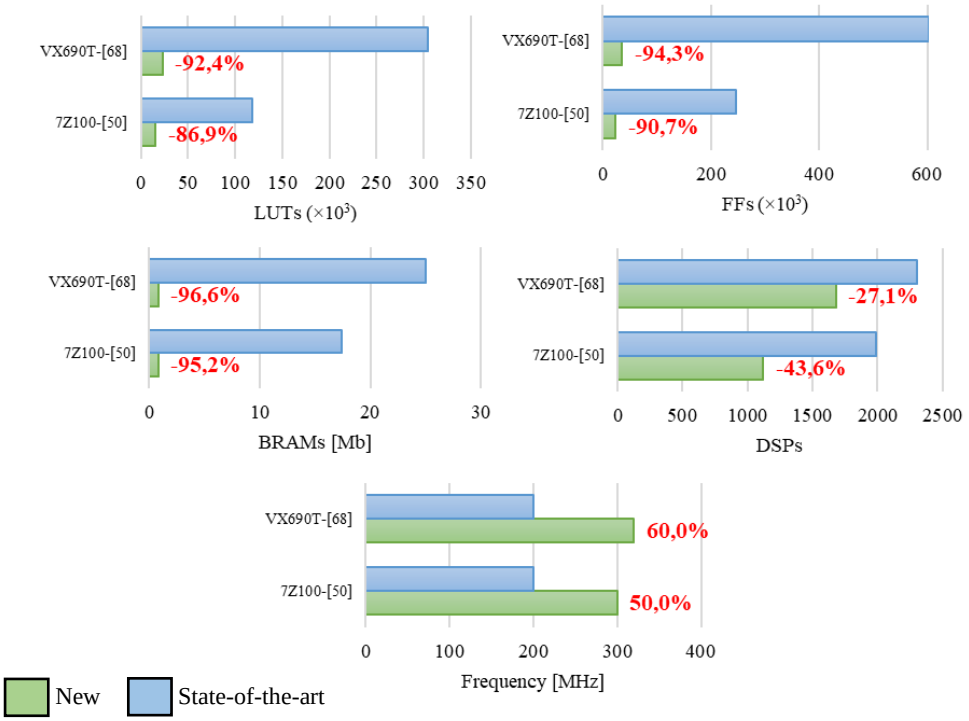


Figure 3.18 Percentage change comparisons (SU): resources, frequency.

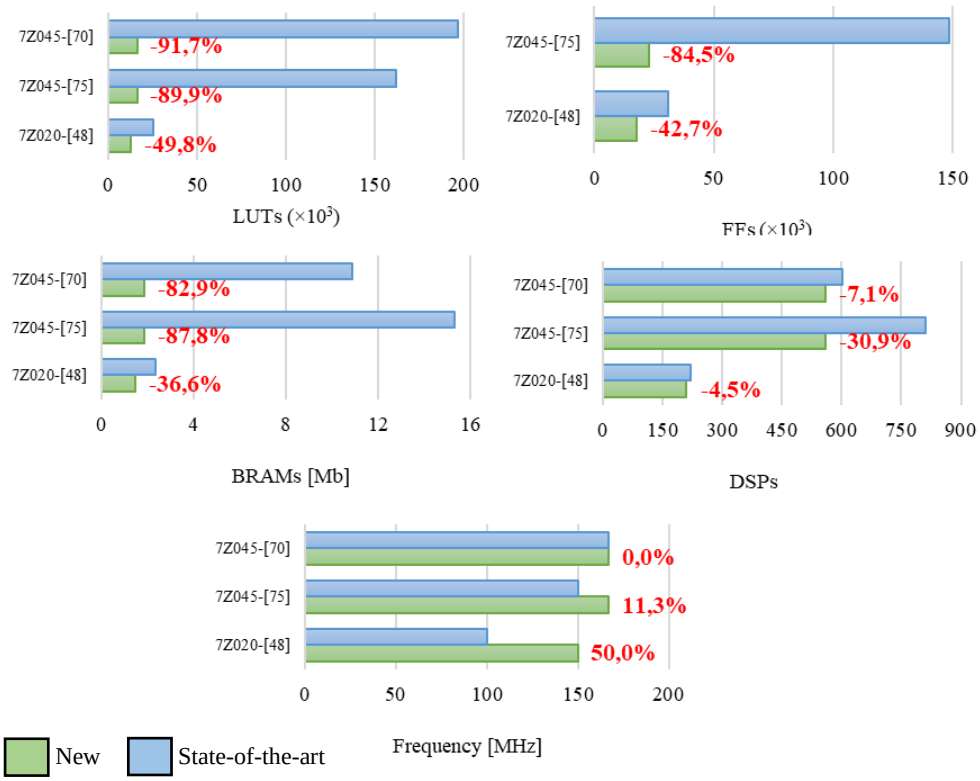


Figure 3.19 Percentage change comparisons (ES): resources, frequency.

considerable usage of on-chip BRAMs. However, this approach limits the scalability towards low-end devices at reasonable speed performance. Conversely, in order to keep data transfer to/from the external memory regular, as happens with the simple raster scan order, the novel accelerator exploits pixel-level parallelism. This is a key feature to strengthen the feasibility towards low-end chips. Similar considerations arise for the accelerator demonstrated in [69]. However, the latter supports the 32-bit floating-point representation, which certainly leads to an overall quality higher than all the other solutions.

Among the ES implementations, that based on the reverse looping approach [48] is the slowest one due to the addressing of pixels. At the parity of the implementation device platform, the ES presented here occupies ~49.8% less LUTs, ~42.7% less FFs, 1.6× less BRAMs and ~5% less DSPs. In addition, it is ~20.7× faster and achieves a density efficiency, evaluated as the ratio GOPS/DSPs, ~21.7× higher.

The architecture here discussed also exhibits advantages with respect to [75, 70]. The significant reduction of occupied resources is due to the more efficient architecture of the generic DU. In fact, the separate analysis, purposely performed varying K and S, demonstrated that the proposed DU always minimizes the amount of occupied LUTs and FFs. This happens because, unlike the accelerator in [75], DSPs are exploited to perform both multiplications and additions. Finally, among the ES implementations, the design presented in [70] is certainly the most competitive in terms of speed performance. However, the novel ES exhibits a computational capability ~19.5% higher, it occupies ~12× less LUTs, ~5.8× less BRAMs and ~7% less DSPs, and consumes ~60% less power.

As previously stated, the computational capability supportable by the proposed accelerator depends on the specific realization platform. In fact, it is mainly dictated by the amount of required DSPs that, in turn, depends on the kernel size and the stride. However, the same amount of DSPs can be exploited differently to implement different configurations of the novel DLPE, depending on the parameters T_M, T_N, S, P_M and P_N. Establishing which configuration is the most appropriate for a specific operating environment is essential to implement the resources in the most efficient way. To this aim, different design spaces can be explored by varying the above parameters. For instance, the design space exploration reported in Figure 3.20 was carried out by considering the XC7Z020 device [58] as the target, thus setting the maximum number of

available DSPs to 220. The behavior of the proposed accelerator has been examined for various kernel sizes K and parallelism factors T_M and T_N with S=2, P_M=4 and P_N=1. In this condition, two scenarios were analyzed: in the *Case1*, T_M=2 and T_N=3 are maintained unchanged to establish the maximum supportable K; conversely in *Case2*, also T_N varies between 24 and 6, while T_M=1. In addition, for each K the maximum T_N was considered (e.g., with K=2, thus T_N=24). The number of DSPs used in the two referred cases by the DE and the AL versus K are indicated. As expected, in the first case the wider the kernel size, the higher the number of DSP slices required by the DE. Conversely, the red line shows that the number of DSPs used to implement the fast adder trees within the AL module is maintained constant to 24, since this value only depends on the parallelism and the stride. The above results show that, in such a case, the maximum kernel size supportable with 220 DSPs is K=5. This is the solution previously referenced in Table 3.3 for both the SU and ES designs implemented within the XC7Z020 device [58].

Results collected for the second analyzed scenario prove that, in order to comply with the amount of DSPs on-chip available, as the kernel size increases, the parallelism must decline. Obviously, as clearly shown by the blue line, the lower the parallelism, the lower the number of DSPs used for accumulations.

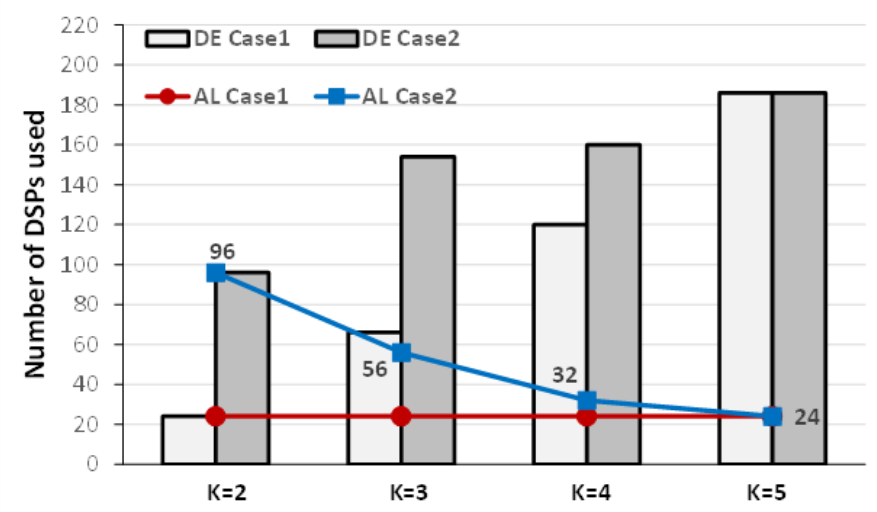


Figure 3.20 Design space exploration within the XC7Z020 device at S=2.

Finally, referring to the XC7Z020 device [58], the execution time of the ES implementation here proposed was compared to a pure software design run by the 666 MHz ARM-Cortex Processor available on-chip. When executing the most complex deconvolution layer involved in the selected DCGAN model [22], the embedded system

is more than 1000× faster than the all-software implementation. Indeed, the software execution requires ~620 ms, while the FPGA execution takes ~0.6 ms.

Overall, the main benefits of the proposed design can be summarized as follows:

- The FPGA accelerator is scalable, thus being suitable to accelerate deconvolutions within different CNNs.
- The novel architecture exploits both data- and circuit-level parallelism. As a results, both low-end and high-end FPGAs may be used to meet power- and performance-constrained environments, respectively.
- DSPs are fully exploited to perform efficient MACs. Accordingly, competitive speed performances with respect to the state-of-the art are achieved with reduced logic resources requirements and power consumption.
- The proposed hardware accelerator complies with the AXI4 protocol [59] and, therefore, it can be easily accommodated within SoCs based on FPGAs. Furthermore, input and output streams are read and written from/to an external memory through the raster-order transfer policy; the latter allows data packets to be moved concurrently, thus positively impacting the latency.

3.3. RUN-TIME RECONFIGURABILITY OF CONVOLUTION ENGINES FOR FPGAS.

3.3.1. BACKGROUND

In the last years, relevant endeavors were undertaken to equip FPGA-based accelerators for CNNs with run-time adaptability in terms of the supported kernel size, stride and image size [77, 42, 78, 79, 80, 44, 81]. Considering that state-of-the-art models frequently use 3×3 kernels, the convolution engine proposed in [77] combines multiple 3×3 basic blocks to manage different kernel sizes. Unfortunately, this approach causes a non-negligible resource underutilization. Just to cite an example, four 3×3 blocks are used to support 5×5 CONVs, with ~30.5% of redundant computing engines.

Authors in [42] proposed an uniform 4×256 MAC array proposed to address the above cited issue. In particular, it was made able to support up to 16×16 CONVs and 256×256 *fmaps*' size with a noticeable scalability. However, the management of both

ifmaps and coefficients buffering requires a quite complex auxiliary control logic, thus affecting the resources requirements significantly.

The reconfigurable architecture presented in [78] allows to execute 3×3, 5×5 and 7×7 convolutions (i.e., varying the kernel size *K*) with a very high utilization of the implemented resources. However, since this design was mainly tailored for high-end powerful platforms, it allows high speed performance at the cost of high power dissipation.

Reconfigurable structures based on systolic arrays [79] reach low execution times, but they suffer of buffering requirements to accommodate data batches, thus showing a relatively high energy dissipation.

The circuit adopted in [80] to accelerate the YOLO model [82] implements only 1×1 and 3×3 kernels at the stride *S*=1. Moreover, even though the adopted parallelism makes this strategy devoted to high-end devices, it achieves limited peak throughput.

In [44], Multiple Convolutional Layer Processors (M-CLP) are used to perform cascaded CONVs in parallel. Given that faster CLPs stay in idle until slower ones end their computations, this approach cannot utilize the resources as efficiently as possible.

Finally, the Adaptive and Hierarchical CNN proposed in [81] uses resources much more efficiently by exploiting the partial reconfiguration of FPGAs, but the reconfiguration time (i.e., tens of milliseconds) significantly reduces the actual throughput.

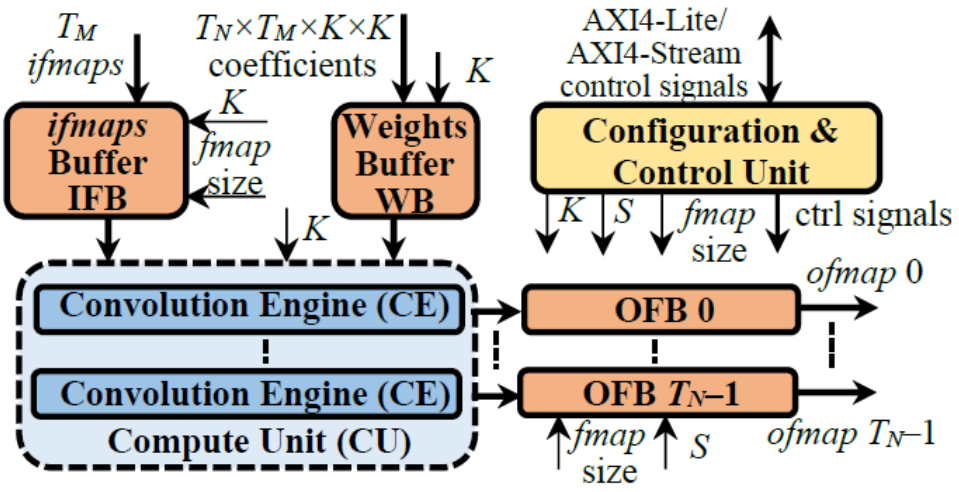
The work presented in this Section introduces an innovative efficient adaptive convolution architecture [15] that, in contrast with its direct competitors [77, 42, 78, 79, 80, 44, 81], can support both odd and even kernel sizes by combining non-uniform computational blocks. Thanks to this approach, for a given CNN model, the proposed hardware accelerator guarantees either a higher utilization of the implemented resources to be achieved or a cheaper implementation platform to be used. Furthermore, the possibility of also managing even-sized kernels makes the approach proposed here suitable for CNN models that use TCONV Layers. Specifically, the proposed Reconfigurable Convolution Processing Element (RCPE) is made able to be run-time reconfigured to support the following set of parameters:

- The kernel size *K*, which can assume the values *K* = 1, 3, 4, 5, 7, 9.
- Square *ifmaps*, consisting of up to 224×224 pixels.

- Stride *S*=1, 2.

3.3.2. THE PROPOSED ARCHITECTURE

The top-level architecture of the RCPE is depicted in Figure 3.21. The included table reports the run-time adaptive parameters, with the respective ranges. The unit complies with the Advanced eXtensible Interface (AXI4) [59] to be integrated within FPGA-based embedded systems. The *Configuration & Control Unit* (CCU) receives the protocol signals and proper control signals that allow self-adapting the computation to different *fmaps'* size, kernel sizes and strides. According to the actual set of reconfigurable parameters, the *ifmap Buffer* (IFB) is supplied by multiple *ifmaps* and arranges proper convolution windows. Meanwhile, the *Weights Buffer* (WB) stores the needed filters. Then, the *Compute Unit* (CU) performs several convolutions in parallel by using multiple *Convolution Engines* (CEs). Eventually, the *ofmap Buffers* (OFBs) store these intermediate results and perform accumulations to compose the final *ofmaps*.



Run-time adaptive parameters	Range
Kernel size *K*	1, 3, 4, 5, 7, 9
Stride *S*	1, 2
Input size *fmap size*	Up to 224×224

Figure 3.21 The top-level architecture of the proposed Reconfigurable Convolution Processing Element.

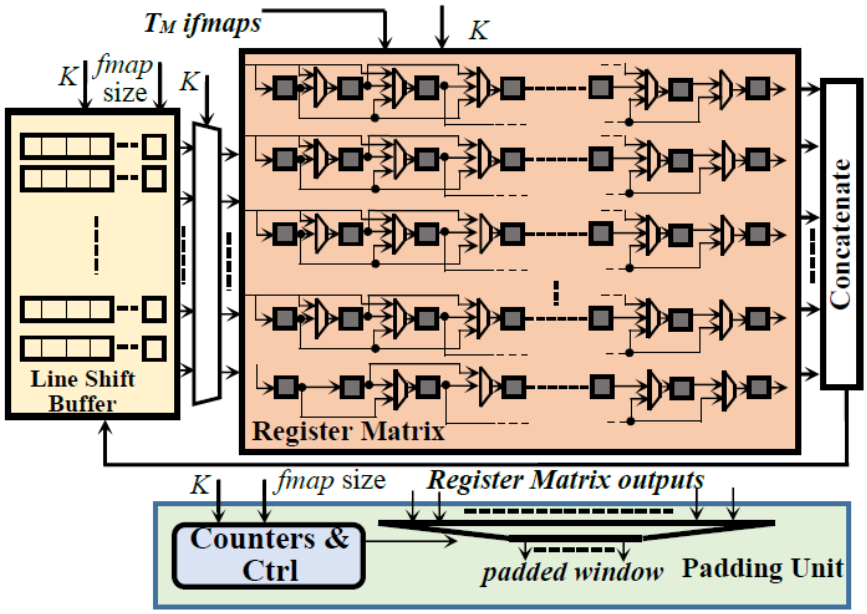
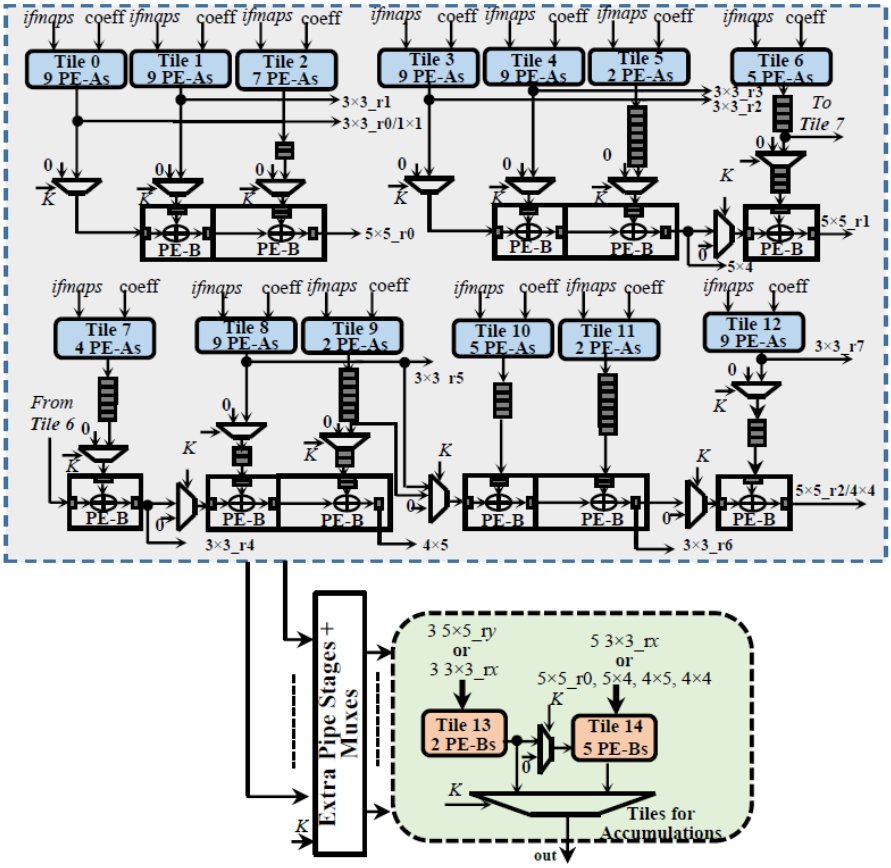


Figure 3.22 Example schematic of the *ifmap Buffer* when K_M=10.

Figure 3.22 illustrates the IFB unit. It consists of three parts: the $K_M \times K_M$ *Register Matrix*, the *Line Shift Buffer* and the *Padding Unit*. It is worth underlining that K_M depends on the maximum supported kernel size. Specifically, we adopted K_M=10. Due to the multiplexing logic interleaving the $K_M \times K_M$ registers of the *Register Matrix*, the IFB can arrange input pixels within different convolution windows: (a) one 9×9 window, (b) one 7×7 window, (c) three 5×5 windows, and (d) eight 3×3 windows. The top leftmost registers are responsible for storing T_M input pixels, each of them belonging to a different channel, whereas the others shift the incoming data and make them available for the subsequent processing. The rightmost registers are also responsible for transmitting pixels to the *Line Shift Buffer*. The latter is made of embedded Block RAMs, working as K_M–*1* First-In-First-Out memories *fmap_size–K* wide, which temporarily stores pixels, thus providing expected input data to the leftmost registers of the *Register Matrix*. Finally, the *Padding Unit* performs proper zero-padding over the composed windows. Such a module consists of $K_M \times K_M$ banks of multiplexers, which furnish either the current pixels or zero in the case of padding. Two counters, that indicate the position of the current anchor point within the *fmap*, manage the selectors of the multiplexers.

The CU includes T_N CEs, with T_N being the number of *ofmaps* provided in parallel. According to the windows provided by the IFB, each CE is able to execute up to T_M convolutions and accumulates the homologous results to provide a provisional *ofmap*

value. Among the possible non-uniformly sized tiles solutions, we chose the CE architecture illustrated in Figure 3.23(a), in that it well fits the Zynq target platforms [58] because uses almost completely the available DSPs.



(a)

```
process(clk)
begin
    if rising_edge(clk) then
        if (rst = '1') then
            mults <= (others => (others => '0'));
            pint <= (others => (others => '0'));
        elsif (ce = '1') then
            for i in 0 to L-1 loop
                mults(i) <= ops_a_reg(i,L-1)*ops_b_reg(i,L-1);
            end loop;
            pint(0) <= cin_reg+mults(0);
            for i in 1 to L-1 loop
                pint(i) <= pint(i-1)+mults(i);
            end loop;
        end if;
    end if;
end process;

sum_out <= std_logic_vector(pint(L-1));
```

(b)

Figure 3.23 (a) The reconfigurable Convolution Engine. **(b)** VHDL code for DSPs belonging to Type-A PEs.

The CE consists of two types of *Processing Elements* (PEs): the MAC units, indicated as *type-A PEs* (PE-As), and the simple adders, named *type-B PEs* (PE-Bs). Both PE types are carefully implemented by means of the embedded DSP slices, assuring that critical signals are routed through fast dedicated interconnections and adaptive paths between PEs are properly activated by the current *K*. Figure 3.23(b) reports a sketch of VHDL code dealing with the interconnection of DSP slices within the generic PE-A. The parametric constructs allows the given code to be reused for any PE-A configuration. The DSPs' computations are reported into the main *for loop*, where the array *mults* refers to the multiplication stage of each DSP, while the array *pint* refers to the accumulation stage. The VHDL code allows the DSPs to use the internal pipeline stages. This is accomplished by using a process that is sensitive to the clock signal *clk*. Each register is equipped with reset *rst* and clock enable *ce*.

Overall, the referred CE is made of 81 PE-As and 18 PE-Bs that allow performing different types of convolutions: (a) eight 3×3 convolutions, (b) three 5×5 convolutions, (c) 5×4, 4×5, 4×4 convolutions, (d) eight 1×1 convolutions. Each of the configurations (a)-(d) refers to computations performed in parallel. Pipeline stages, depicted in grey, and the *Extra Pipe Stages+Muxes* unit ensure the parallel results provided in the various configurations by the PE-As/PE-Bs tiles to be correctly time aligned. The latter unit also selects data for the subsequent accumulations. In fact, the last 7 PE-Bs employed in the *Tiles for Accumulations* module sum either the eight 3×3 results, indicated as *3×3_rx* (with *x=0,…,7*), or the three 5×5 results, indicated as *5×5_ ry* (with *y=0,…,2*). Furthermore, this set of accumulators can be exploited to provide other types of convolution results: both the *5×5_r0* and the *5×5_r1* can be accumulated by the *Tile 13* to comply with a 7×7 convolution. Alternatively, the *5×5_r0*, the *5×4* and *4×5* outputs, and the *4×4* result can be supplied to the *Tile 14* to compose a 9×9 convolution output.

Finally, the T_N OFBs receive the results from the T_N CEs and accumulate them to the homologous results produced at the previous steps. These provisional results are stored within simple dual-port BRAMs adapted to the current *ofmaps* size. The OFBs also take care of managing the stride *S*. Indeed, when *S*=1, all the incoming results are stored within the BRAMs. Conversely, when *S*=2, BRAMs store ¼ of the outputs.

3.3.3. EXPERIMENTAL RESULTS

The proposed RCPE was integrated within the heterogeneous Embedded System (ES) depicted in Figure 3.24. The Processing System (PS) configures and controls the custom circuits implemented within the Programmable Logic (PL) that hosts the RCPE and the Direct Memory Access (DMA) units.

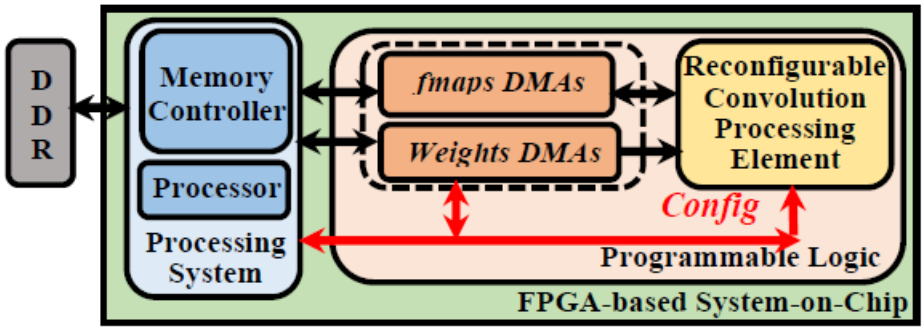


Figure 3.24 The referred embedded system architecture.

The latter are responsible for data transferring from/to an external DDR memory. The *fmaps DMAs* provide the RCPE with up to T_M input pixels at a time and, at the completion of the entire processing, they move the T_N output pixels towards the external storage resource. Meanwhile, the *Weights DMAs* provide the needed filters for each convolutional step. The DMA units and the RCPE receive configuration signals from the PS by means of AXI4-Lite transactions [59].

The Xilinx Zynq 7020 and 7045 SoCs [58] were used to implemented the referred system. Table 3.4 summarizes the results obtained by the comparison with several state-of-the-art counterparts. The resource utilization, the clock frequency and the power consumption, as well as the supported kernel sizes and strides, the maximum achievable parallelism ($T_{Mmax} \times T_{Nmax}$) and the data precision are reported. In addition, the throughput, expressed in terms of both the peak and the effective Giga Operations per Second (GOPS), and the CONVs execution time related to different CNN models are highlighted. In this regard, it must be noted that, while the effective GOPS depend on the specific CNN model, the peak GOPS rely on the computational capability of the implemented architecture. Figures 3.25-3.26 illustrate the main percentage changes with competitors.

The cheapest version of the proposed design, implemented within the low-end Zynq-7020 device [58], uses 8-bit data and performs up to 8×2 parallel convolutions at the 118 MHz running frequency. At a parity of the supported data precision, this design

shows appreciable advantages in terms of the resource utilization and the power consumption over the competitors [77, 78, 79]. In fact, the new reconfigurable architecture, purposely designed to maximize the portion of the global computational load sustained by DSPs slices, saves up to ~87%, ~84.1% and ~91.3% of Look-Up Tables (LUTs), Flip-Flops (FFs) and Block RAMs (BRAMs), respectively. Among the compared 8-bit designs, at the parity of the neural network model, those described in [77, 78] exhibit better effective-throughput/parallelism ratio. This is because, while the design in [77] significantly relies on a clock frequency ~1.8× higher, that proposed in [78] is based on a more powerful platform that, in turn, offers more DSPs and consumes more energy. However, [77, 78] support much less kernel sizes and strides configuration, thus limiting their flexibility.

Table 3.4 Characterization of the proposed architecture for run-time adaptive convolutions and state-of-the-art comparisons.

Ref.	[77]	[78]	[79]	New			
Device	7Z020	ZU9EG	7Z035	7Z020			
LUTs	29867	49022	75241	9772			
FFs	35489	85340	91444	14544			
BRAMs [Mb]	3	13	10.97	1.13			
DSPs	190	1048	758	200			
Freq. [MHz]	214	195	100	118			
Peak GOPS	-	350.4	-	42.7			
Precision	8-bit	8-bit	8-bit	8-bit			
***K* (*S*)**	1,3,5(1)	3,5,7(1)	1,3,5,7 (1,2)	1,3,4,5,7,9 (1,2)			
Par. T_{Mmax}×T_{Nmax}	16×2	8×8	-	8×2			
Power [W]	3.50	4.80	3.44	1.98			
CNN	VGG16	VGG16	TinyYOLO[1]	VGG16	YOLO[2]	TinyYOLO[2]	VGGS
Eff. GOPS	84.3	214	125	32	17.1	19.7	28.6
Eff.GOPS/Par.	2.63	3.34	-	2.00	1.07	1.23	1.79
GOPS/DSP	0.444	0.204	0.165	0.160	0.086	0.099	0.143
Time [ms]	364	140	53	958.3	325.5	79.5	241.9

Ref.	[77]	[42]	[80]	[44]	New			
Device	7Z045	7VX485T	ZU9EG	7VX690T	7Z045			
LUTs	182616	78318	95136	133854	28743			
FFs	127653	96929	90589	161411	57775			
BRAMs [Mb]	17.09	12.5	8.63	19.48	7.49			
DSPs	780	1034	609	3494	800			
Freq. [MHz]	150	150	300	170	150			
Peak GOPS	-	300.0	289	-	217.2			
Precision	16-bit	18-bit	16-bit	16-bit	16-bit			
***K* (*S*)**	1,3,5(1)	≤16(1)	1,3 (1)	1,3,7 (1,2)	1,3,4,5,7,9 (1,2)			
Par.T_{Mmax}×T_{Nmax}	64×2	-	8×64	5×256	8×8			
Power [W]	9.63	18	11.80	7.20	2.75			
CNN	VGG16	Custom	YOLO[1]	SqueezeNet	VGG16	YOLO[2]	TinyYOLO[2]	VGGS
Eff. GOPS	187.8	129.7	102	909.7	153.6	82.3	91.9	140.7
Eff.GOPS/Par.	1.47	-	0.19	0.71	2.40	1.29	1.44	2.19
GOPS/DSP	0.241	0.125	0.167	0.260	0.192	0.103	0.115	0.176
Time [ms]	163	0.3	288	0.8	199.8	67.8	17	49.2

[1]416×416 input images [2]224×224 input images.

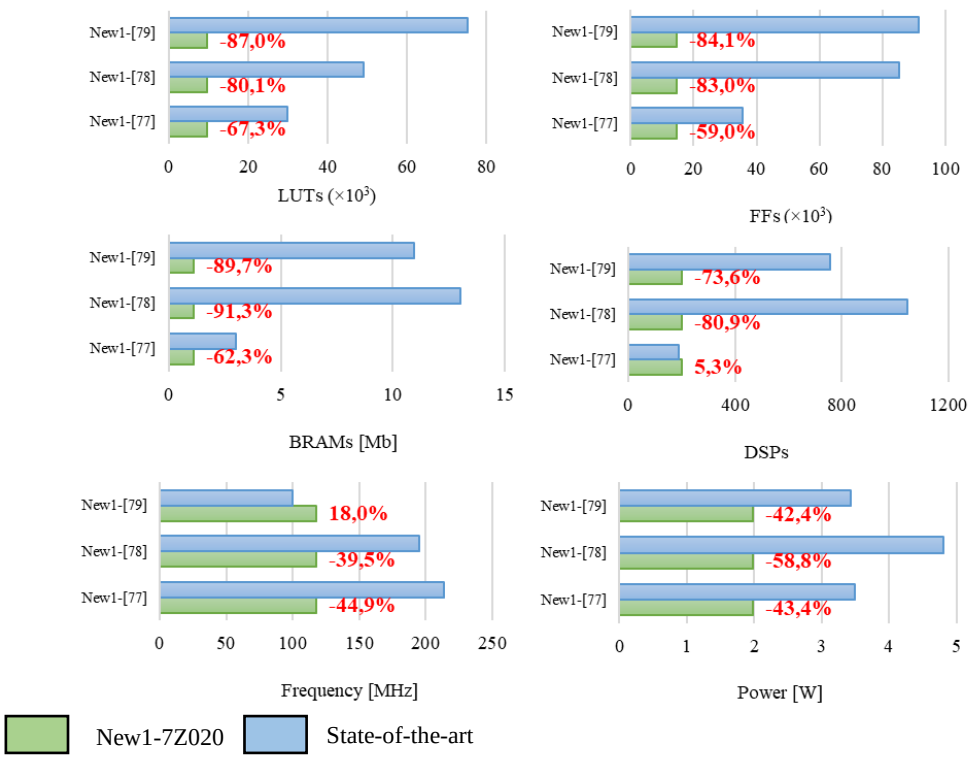


Figure 3.25 Percentage change comparisons (7Z020 impl.): resources, frequency, power.

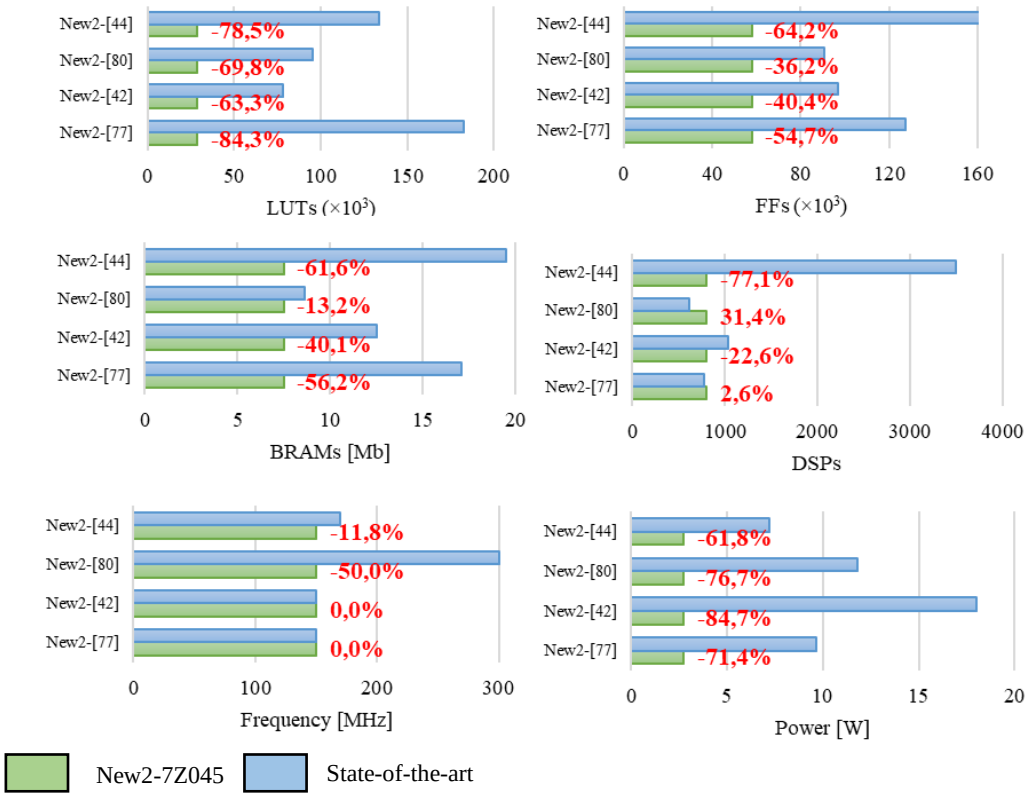


Figure 3.26 Percentage change comparisons (7Z045 impl.): resources, frequency, power.

Moving towards the most powerful implementation within the Zynq-7045 SoC [58], the novel accelerator performs up to 8×8 parallel convolutions and, compared to the 16- and 18-bit counterparts [77, 42, 80, 44], it exhibits the most favorable effective-throughput/parallelism ratio. This behavior is further confirmed if the above ratio is correlated to the dissipated power. When the VGG-16 [7] and the YOLO [82] models are taken into account, the proposed architecture achieves an effective throughput only ~1.2× lower than that reported by [77] and [80], despite the halved parallelism and clock frequency, respectively.

The architectures introduced in [42, 44] reach, in turn, the highest flexibility and the most relevant parallelism. However, such results require a noticeable amount of DSPs, actually among the highest among the competitors.

Finally, it is worth underlining the behavior of the proposed 16-bit architecture when the VGG-S model [83] is inferred to. The latter consists of CONVs at different *K*, ranging from 3 to 7, and *S*, ranging from 1 to 2. In such a case, the effective throughput reaches 140.7 GOPS, thus confirming the performance efficiency of the proposed adaptive scheme.

3.4. TOWARDS A RECONFIGURABLE ARCHITECTURE FOR CONVOLUTIONS AND TRANSPOSED CONVOLUTIONS

3.4.1. BACKGROUND

Typically, neural networks involved in image up-sampling tasks follow an encoder-decoder structure. The encoder consists of a stack of CONV Layers, which progressively down-sample the *fmaps* to abstract the features' extraction. Proper up-sampling layers constitute the decoder in order to recover the original sizes of features to provide the output images.

Super Resolution (SR) imaging (introduced in Section 2.3.3) is a well-recognized example that has become crucial in several applications, such as video surveillance, medical diagnosis, remote sensing. CNN-based SR algorithms typically adopt TCONV layers to up-sample images [31] that, with their computational complexity up to 6.75× higher than CONV Layers, represent the most critical component of the CNNs [47]. In addition, in comparison with CONVs, TCONV Layers require more complex strategies

to access the data memory, and make skipping operations necessary to manage the incoming pixels properly [14]. In order to overcome the aforementioned issues, besides the alternatives discussed in Section 3.2, other solutions were proposed to transform TCONV into CONV layers by pre-processing either the input data or the filters coefficients [47]. Starting from an analysis of the IOM method [49], and with the objective of avoiding overlapping on input activations, the computational scheme proposed in [47, 84, 85, 86] performs an inverse mapping on the filter coefficients. More specifically, the proposed *Transform Deconvolution into Convolution* (TDC) approach converts each filter of a TCONV into $S_D \times S_D$ smaller sub-filters according to the relative position of the original input activations within the up-sampled *ifmap*, where S_D is the stride of the TCONV. Due to this splitting strategy, several locations within the sub-filters contain zero values, thus causing unbalanced computations. Furthermore, the specific configuration (i.e., size and number of the sub-filters) depends on S_D. Therefore, the splitting process has to be performed offline and the pre-processed filters must be stored on-chip, thus limiting the possibility of reconfiguring at run-time the architecture to accelerate different CNN models.

As observed in [84], when the zero-TCONV approach is used, the filter coefficients that are being multiplied by zero activations can be removed by decomposing filters into additional sub-blocks. Also for this decomposition algorithm, the filters must be pre-processed offline. Moreover, in order to remove unbalanced computations, an overall logic more complex than [47] is required in that sub-filters are not regular in size and require proper control.

To manage both TCONV and CONV operations, hardware designs proposed in [85, 86] decompose filters into smaller sub-blocks with different dimensions, according with the values of the filter size k and S_D; then to avoid filter reversal and zero padding on the borders, they apply a variant of the Winograd algorithm. In such a case, unconventional computational modules, suitable to implement operations involved in the Winograd transformation (like inverse transformation of matrices), are required.

To overcome all the aforementioned issues, we conceived a novel hardware-oriented algorithm [16] made able to convert TCONV into CONV Layers efficiently, without the requirement of any pre-processing step. The given architecture, designed using parametric VHDL, can be adapted at design time to support different configurations

of parallelism and input sizes. Functional parameters of CONVs/TCONVs can be adapted at run-time by means of the control logic.

3.4.2. THE PROPOSED HARDWARE-ORIENTED ALGORITHM

In contrast to [47, 84, 85, 86], which manipulate the $k \times k$ filter coefficients to form smaller sub-blocks (thus introducing the necessity of offline elaborations to arrange weights in different sub-filters), the proposed algorithmic strategy applies an unconventional remapping directly on the incoming *ifmaps* values. From the hardware perspective, this means that: (1) the process occurs online and the pre-processing is not required, and (2) the result of the proposed algorithm can be provided as soon as it is produced, thus avoiding additional times and buffering/computing resources. As a further advantage, the incoming *ifmaps* are not actually up-sampled, but they are processed as if they were up-sampled with the conventional TCONV approach with zeros insertion.

In order to achieve high-speed performances and to prevent redundant multiplications by zero, the proposed method is on-purpose made able to furnish $S_D \times S_D$ results in parallel for each computed *ofmap*. The steps illustrated in Figure 3.27(a) are performed to process the $K_C \times K_C$ window of activations, with $K_C = \left\lceil \frac{k+S_D-1}{S_D} \right\rceil$. The generic sliding window received as input, with the first (i.e., the top left) activation of the window being $I_{i,j}$ (with $i=0,\ldots,H_I-1$ and $j=0,\ldots,W_I-1$), is remapped within a $k \times k$ window; then element-wise multiplications are performed between the remapped window and the $k \times k$ filter, followed by accumulations to produce $S_D \times S_D$ parallel results. The main innovation introduced with respect to the conventional approach and methods based on filter decomposition [47, 84, 85, 86] is the remapping of the $K_C \times K_C$ input activations within the sliding window *RI*. The latter is formed as illustrated in Figure 3.27(b) that also shows the local row and column indices *m* and *n*, both varying from 0 to k–1. The remapped window is obtained by applying the following basic rules:

- the first activation $I_{i,j}$ is assigned to the local position (0,0) within the up-sampled window *RI* and no more replicated;
- the activations having the row index equal to *i* are replicated S_D times horizontally;
- the activations having the column index equal to *j* are replicated S_D times vertically;

- the activations having the row and column indices varying, respectively, from $i+1$ to $i+K_C-2$ and from $j+1$ to $j+K_C-2$, are replicated S_D times vertically and S_D times horizontally, thus forming $S_D \times S_D$ sub-windows;
- if $(k-1)\ mod\ S_D = 0$, the activations having the row index equal to $K_C - 1$ are replicated S_D times horizontally (this is the case illustrated in Figure 3.27(b)), otherwise they are replicated $(k-1)\ mod\ S_D$ times;
- if $(k-1)\ mod\ S_D = 0$, the activations having the column index equal to $K_C - 1$ are replicated S_D times vertically (this is the case illustrated in Figure 3.27(b)), otherwise they are replicated $(k-1)\ mod\ S_D$ times.

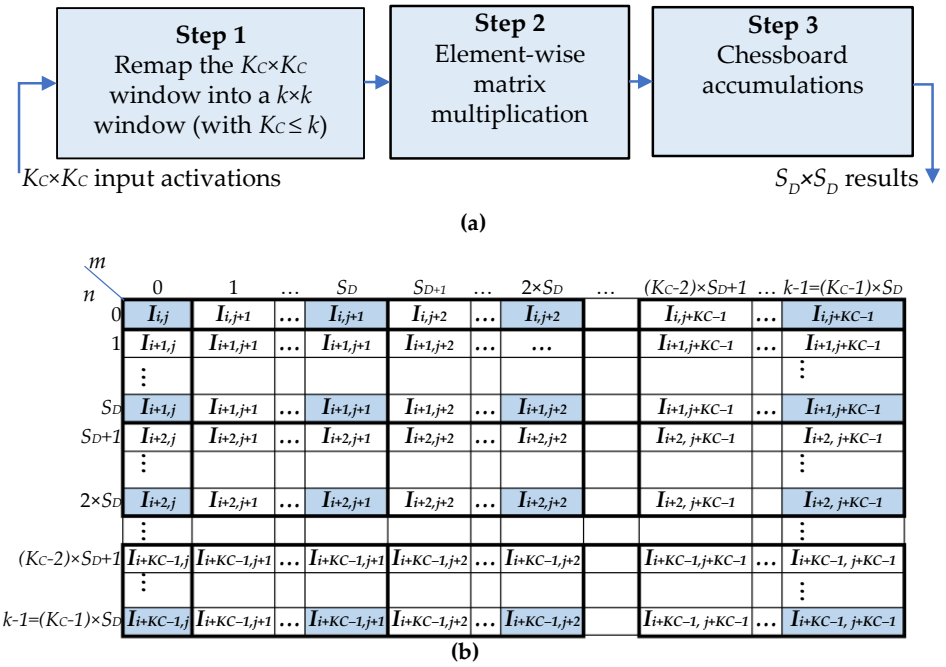


Figure 3.27 The novel algorithm **(a)** the computational steps involved, **(b)** the remapping strategy.

The elements of the remapped window, obtained as above explained, are multiplied by the homologous filter coefficients $W_{m,n}$ that do not require any type of rearrangement. Then, the computed $k \times k$ products $PP_{m,n}$ are properly accumulated to finally provide the $S_D \times S_D$ parallel results $O_{i \times S_D + p, j \times S_D + q}$, with p and q varying from 0 to S_D–1. To take into account the up-sampling factor S_D, the generic result $O_{i \times S_D + p, j \times S_D + q}$ must be computed by accumulating $K_C \times K_C$ products $PP_{mm,nn}$ picked up starting from the location $mm = i \times S_D$, $nn = j \times S_D$ and going on as in a chessboard with horizontal and vertical jumps of S_D positions (i.e., with stride S_D). However, it is worth noting that some jumps

lead to values of *mm* and/or *nn* exceeding *k*, thus indexing unavailable products. Actually, referring to the *ifmap* currently processed as if it were up-sampled with the conventional approach, it is easy to verify that these missing products correspond to multiplications by zero. Therefore, they do not contribute to the accumulate operations and can be simply ignored. As a consequence, the results computed with the proposed strategy have the same values provided by the conventional approach [64]. However, the method proposed here completely avoids multiplications by zero and filter partitioning.

It is important observing that the remapping strategy here proposed is a different point of view of the methods based on filters decomposition [47, 84, 85, 86]. Indeed, while the latter re-arrange filter coefficients to perform proper element-wise multiplications, the former re-arrange input activations. However, the proposed strategy is more efficient from the hardware perspective, because it allows online computations and does not require complex architectures to manage the remapping.

	0	1	2	3	4	5	6	7	8
0	$I_{i,j}$	$I_{i,j+1}$	$I_{i,j+1}$	$I_{i,j+2}$	$I_{i,j+2}$	$I_{i,j+3}$	$I_{i,j+3}$	$I_{i,j+4}$	$I_{i,j+4}$
1	$I_{i+1,j}$	$I_{i+1,j+1}$	$I_{i+1,j+1}$	$I_{i+1,j+2}$	$I_{i+1,j+2}$	$I_{i+1,j+3}$	$I_{i+1,j+3}$	$I_{i+1,j+4}$	$I_{i+1,j+4}$
2	$I_{i+1,j}$	$I_{i+1,j+1}$	$I_{i+1,j+1}$	$I_{i+1,j+2}$	$I_{i+1,j+2}$	$I_{i+1,j+3}$	$I_{i+1,j+3}$	$I_{i+1,j+4}$	$I_{i+1,j+4}$
3	$I_{i+2,j}$	$I_{i+2,j+1}$	$I_{i+2,j+1}$	$I_{i+2,j+2}$	$I_{i+2,j+2}$	$I_{i+2,j+3}$	$I_{i+2,j+3}$	$I_{i+2,j+4}$	$I_{i+2,j+4}$
4	$I_{i+2,j}$	$I_{i+2,j+1}$	$I_{i+2,j+1}$	$I_{i+2,j+2}$	$I_{i+2,j+2}$	I_{i+2j+3}	$I_{i+2,j+3}$	$I_{i+2,j+4}$	$I_{i+2,j+4}$
5	$I_{i+3,j}$	$I_{i+3,j+1}$	$I_{i+3,j+1}$	$I_{i+3,j+2}$	$I_{i+3,j+2}$	$I_{i+3,j+3}$	$I_{i+3,j+3}$	$I_{i+3,j+4}$	$I_{i+3,j+4}$
6	$I_{i+3,j}$	$I_{i+3,j+1}$	$I_{i+3,j+1}$	$I_{i+3,j+2}$	$I_{i+3,j+2}$	$I_{i+3,j+3}$	$I_{i+3,j+3}$	$I_{i+3,j+4}$	$I_{i+3,j+4}$
7	$I_{i+4,j}$	$I_{i+4\ j+1}$	$I_{i+4,j+1}$	$I_{i+4,j+2}$	$I_{i+4,j+2}$	$I_{i+4,j+3}$	$I_{i+4,j+3}$	$I_{i+4,j+4}$	$I_{i+4,j+1}$
8	$I_{i+4,j}$	$I_{i+4,j+1}$	$I_{i+4,j+1}$	$I_{i+4,j+2}$	$I_{i+4,j+2}$	$I_{i+4,j+3}$	$I_{i+4,j+3}$	$I_{i+4,j+4}$	$I_{i+4,j+4}$

(a)

	0	1	2	3	4	5	6	7	8
0	$W_{0,0}$	$W_{0,1}$	$W_{0,2}$	$W_{0,3}$	$W_{0,4}$	$W_{0,5}$	$W_{0,6}$	$W_{0,7}$	$W_{0,8}$
1	$W_{1,0}$	$W_{1,1}$	$W_{1,2}$	$W_{1,3}$	$W_{1,4}$	$W_{1,5}$	$W_{1,6}$	$W_{1,7}$	$W_{1,8}$
2	$W_{2,0}$	$W_{2,1}$	$W_{2,2}$	$W_{2,3}$	$W_{2,4}$	$W_{2,5}$	$W_{2,6}$	$W_{2,7}$	$W_{2,8}$
3	$W_{3,0}$	$W_{3,1}$	$W_{3,2}$	$W_{3,3}$	$W_{3,4}$	$W_{3,5}$	$W_{3,6}$	$W_{3,7}$	$W_{3,8}$
4	$W_{4,0}$	$W_{4,1}$	$W_{4,2}$	$W_{4,3}$	$W_{4,4}$	$W_{4,5}$	$W_{4,6}$	$W_{4,7}$	$W_{4,8}$
5	$W_{5,0}$	$W_{5,1}$	$W_{5,2}$	$W_{5,3}$	$W_{5,4}$	$W_{5,5}$	$W_{5,6}$	$W_{5,7}$	$W_{5,8}$
6	$W_{6,0}$	$W_{6,1}$	$W_{6,2}$	$W_{6,3}$	$W_{6,4}$	$W_{6,5}$	$W_{6,6}$	$W_{6,7}$	$W_{6,8}$
7	$W_{7,0}$	$W_{7,1}$	$W_{7,2}$	$W_{7,3}$	$W_{7,4}$	$W_{7,5}$	$W_{7,6}$	$W_{7,7}$	$W_{7,8}$
8	$W_{8,0}$	$W_{8,1}$	$W_{8,2}$	$W_{8,3}$	$W_{8,4}$	$W_{8,5}$	$W_{8,6}$	$W_{8,7}$	$W_{8,8}$

(b)

Figure 3.28 Example of computation with k = 9, S_D = 2 and K_C=5: **(a)** the remapped window RI; **(b)** the filter W.

To better explain the novel computational scheme, let us consider the example in Figure 3.28 that refers to k=9, S_D=2 and K_C=5. In this case, the local row and column indices *m* and *n* vary from 0 to 8. Therefore, for each input pixel Ii,j, the above explained basic rules lead to the remapped 9×9 window visible in Figure 3.28(a), where the 5×5 elements of the original sliding window are highlighted in blue. It can be observed that the remapped window collects all the data needed to compute the results $O_{i\times S_D+p,j\times S_D+q}$ contemporaneously, with the indices *p* and *q*, used to locate the produced results within the *ofmap*, ranging between 0 and 1. Indeed, since S_D=2, the results $O_{i\times 2,j\times 2}$, $O_{i\times 2,j\times 2+1}$, $O_{i\times 2+1,j\times 2}$ and $O_{i\times 2+1,j\times 2+1}$ are computed as given in (1).

$$
\begin{aligned}
O_{i\times 2,\, j\times 2} = {} & I_{i,j} \times W_{0,0} + I_{i,j+1} \times W_{0,2} + I_{i,j+2} \times W_{0,4} + I_{i,j+3} \times W_{0,6} + I_{i,j+4} \times W_{0,8} + \\
& + I_{i+1,j} \times W_{2,0} + I_{i+1,j+1} \times W_{2,2} + I_{i+1,j+2} \times W_{2,4} + I_{i+1,j+3} \times W_{2,6} + I_{i+1,j+4} \times W_{2,8} + \\
& + I_{i+2,j} \times W_{4,0} + I_{i+2,j+1} \times W_{4,2} + I_{i+2,j+2} \times W_{4,4} + I_{i+2,j+3} \times W_{4,6} + I_{i+2,j+4} \times W_{4,8} + \\
& + I_{i+3,j} \times W_{6,0} + I_{i+3,j+1} \times W_{6,2} + I_{i+3,j+2} \times W_{6,4} + I_{i+3,j+3} \times W_{6,6} + I_{i+3,j+4} \times W_{6,8} + \\
& + I_{i+4,j} \times W_{8,0} + I_{i+4,j+1} \times W_{8,2} + I_{i+4,j+2} \times W_{8,4} + I_{i+4,j+3} \times W_{8,6} + I_{i+4,j+4} \times W_{8,8}
\end{aligned}
$$

$$
\begin{aligned}
O_{i\times 2,\, j\times 2+1} = {} & I_{i,j+1} \times W_{0,1} + I_{i,j+2} \times W_{0,3} + I_{i,j+3} \times W_{0,5} + I_{i,j+4} \times W_{0,7} + \\
& + I_{i+1,j+1} \times W_{2,1} + I_{i+1,j+2} \times W_{2,3} + I_{i+1,j+3} \times W_{2,5} + I_{i+1,j+4} \times W_{2,7} + \\
& + I_{i+2,j+1} \times W_{4,1} + I_{i+2,j+2} \times W_{4,3} + I_{i+2,j+3} \times W_{4,5} + I_{i+2,j+4} \times W_{4,7} + \\
& + I_{i+3,j+1} \times W_{6,1} + I_{i+3,j+2} \times W_{6,3} + I_{i+3,j+3} \times W_{6,5} + I_{i+3,j+4} \times W_{6,7} + \\
& + I_{i+4,j+1} \times W_{8,1} + I_{i+4,j+2} \times W_{8,3} + I_{i+4,j+3} \times W_{8,5} + I_{i+4,j+4} \times W_{8,7}
\end{aligned}
\qquad (1)
$$

$$
\begin{aligned}
O_{i\times 2+1,\, j\times 2} = {} & I_{i+1,j} \times W_{1,0} + I_{i+1,j+1} \times W_{1,2} + I_{i+1,j+2} \times W_{1,4} + I_{i+1,j+3} \times W_{1,6} + I_{i+1,j+4} \times W_{1,8} + \\
& + I_{i+2,j} \times W_{3,0} + I_{i+2,j+1} \times W_{3,2} + I_{i+2,j+2} \times W_{3,4} + I_{i+2,j+3} \times W_{3,6} + I_{i+2,j+4} \times W_{3,8} + \\
& + I_{i+3,j} \times W_{5,0} + I_{i+3,j+1} \times W_{5,2} + I_{i+3,j+2} \times W_{5,4} + I_{i+3,j+3} \times W_{5,6} + I_{i+3,j+4} \times W_{5,8} + \\
& + I_{i+4,j} \times W_{7,0} + I_{i+4,j+1} \times W_{7,2} + I_{i+4,j+2} \times W_{7,4} + I_{i+4,j+3} \times W_{7,6} + I_{i+4,j+4} \times W_{7,8}
\end{aligned}
$$

$$
\begin{aligned}
O_{i\times 2+1,\, j\times 2+1} = {} & I_{i+1,j+1} \times W_{1,1} + I_{i+1,j+2} \times W_{1,3} + I_{i+1,j+3} \times W_{1,5} + I_{i+1,j+4} \times W_{1,7} + \\
& + I_{i+2,j+1} \times W_{3,1} + I_{i+2,j+2} \times W_{3,3} + I_{i+2,j+3} \times W_{3,5} + I_{i+2j+,4} \times W_{3,7} + \\
& + I_{i+3,j+1} \times W_{5,1} + I_{i+3,j+2} \times W_{5,3} + I_{i+3,j+3} \times W_{5,5} + I_{i+3,j+4} \times W_{5,7} + \\
& + I_{i+4,j+1} \times W_{7,1} + I_{i+4,j+2} \times W_{7,3} + I_{i+4,j+3} \times W_{7,5} + I_{i+4,j+4} \times W_{7,7}
\end{aligned}
$$

As expected, the results $O_{i\times 2+p,j\times 2+q}$, corresponding to p and/or q greater than zero, are obtained by accumulating less than $K_C \times K_C$ products and the missing products are simply ignored, since they are related to multiplications by zero.

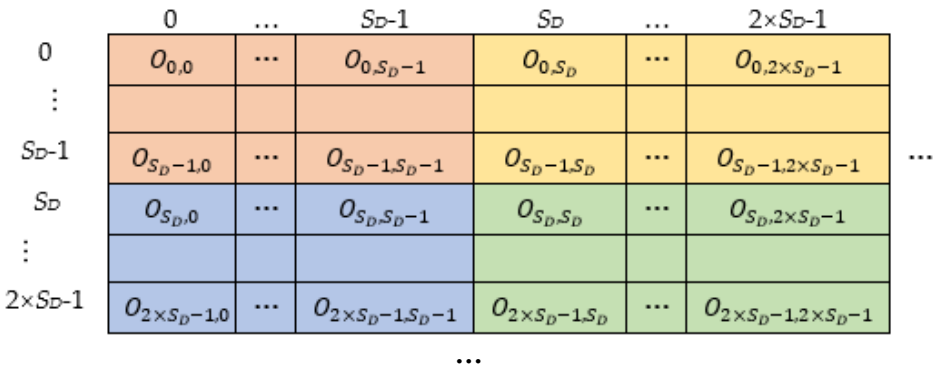


Figure 3.29 The arrangement of the computed results within the generic *ofmap*.

The computations described above are repeated for each pixel of the *ifmap* and, at the completion, the $H_i \times W_i$ groups of $S_D \times S_D$ results obtained in this way are arranged in the *ofmap* as illustrated in Figure 3.29. There, different colors are used to highlight each group of $S_D \times S_D$ results computed in parallel.

It is worth noting that, when S_D is 1, K_C is equal to k and the sliding window does not require remapping operations; in such a case, the proposed algorithm performs a standard CONV. With the input volume consisting of M *ifmaps*, all the computations

described above must be repeated *M* times. The *M* intermediate *ofmaps* computed in this way are summed up to populate the volume of the expected *N ofmaps*.

3.4.3. THE PROPOSED ARCHITECTURE

The novel method above presented to convert TCONVs into CONVs was employed within a reconfigurable hardware structure purposely designed to perform both CONVs and TCONVs by run-time adapting itself to different operating modes.

In order to achieve high computational speeds, the proposed hardware accelerator exploits a certain level of parallelism. In the following, it is shown that T_M *ifmaps* and T_N filters are processed at a time, with T_M and T_N varying at run-time in accordance with the current operation mode, the kernel size *k* and the up-sampling factor S_D. Anyway, for the operations of the generic layer to be completed, regardless of whether it is a CONV or a TCONV layer, $\left\lceil \frac{M}{T_M} \right\rceil \times \left\lceil \frac{N}{T_N} \right\rceil$ steps are required.

The top-level architecture of the proposed hardware is shown in Figure 3.30. It consists of the *Computational Module* (CM) and the *Finite State Machine* (FSM). The former receives as inputs T_M *ifmaps* and T_N filters, each consisting of T_M kernels collecting $k \times k$ coefficients, and provides T_N *ofmaps* at a time. Conversely, the FSM is supplied with the input configuration, which sets the required operating mode (indicating whether CONVs or TCONVs must be performed), the kernel size *k*, the *fmap* sizes and the window size K_C, and furnishes proper control/configuration signals to the CM. Specifically, the FSM: 1) run-time configures the proposed hardware accelerator, thus ensuring that T_M and T_N change as required by each layer; 2) scans the computational steps.

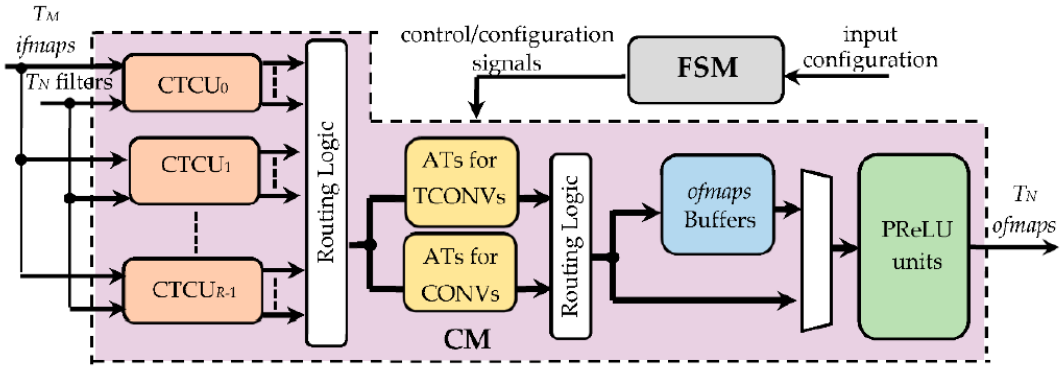


Figure 3.30 The top-level architecture of the proposed hardware accelerator.

The CM splits the incoming T_N filters into R groups and employs as many *CONV/TCONV Units* (CTCUs). Each CTCU, depending on the received control and configuration signals, arranges data in proper sliding windows and executes either CONVs or TCONVs by processing the T_M *ifmaps* and its own $\left\lceil \frac{T_N}{R} \right\rceil$ filters. The results provided by the CTCUs are then dispatched to the subsequent modules passing through the *Routing Logic* purposely designed to take into account that the supported operating modes lead to different data-flows. In fact, depending on whether CONVs or TCONVs are performed, the intermediate results related to the current T_M input channels must be accumulated by the proper *Adder Trees* (ATs). Then, data must be routed either to the *ofmaps Buffers*, which happens when the computation of the current T_N *ofmaps* is not yet completed, or, vice versa, to the *Parametric Rectified Linear Units* (PReLUs) that implement the homonymous linear rectification method [87].

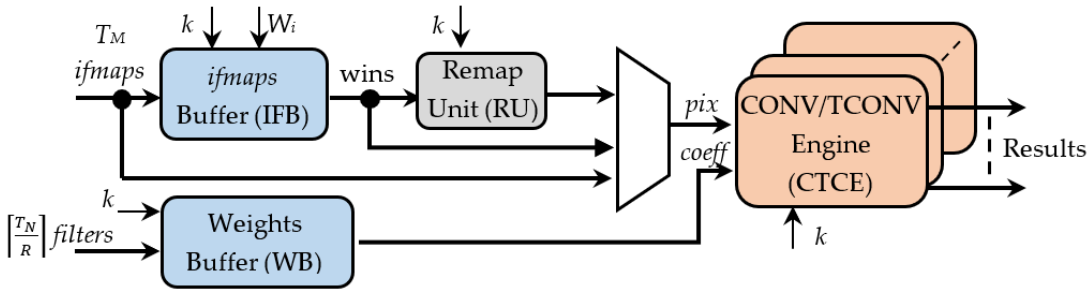


Figure 3.31 The architecture of the CONV/TCONV Unit.

The generic CTCU is structured as depicted in Figure 3.31. The *ifmaps Buffer* (IFB) and the *Weights Buffer* (WB) collect, respectively, the N_A-bit pixels of the incoming T_M *ifmaps* and the N_W-bit coefficients of the received $\left\lceil \frac{T_N}{R} \right\rceil$ filters. In particular, the IFB circuit is responsible for arranging the $K_C \times K_C$ sized sliding windows that will be processed through the proposed algorithm and is based on the architecture proposed in our previous work [15] and described in Section 3.3.2. When TCONVs are executed, the *Remap Unit* (RU) performs the first step of the proposed approach. It implements the novel logic discussed in Section 3.4.2 to remap the T_M $K_C \times K_C$ sliding windows into as many $k \times k$ windows. The $\left\lceil \frac{T_N}{R} \right\rceil$ CONV/TCONV Engines (CTCEs) execute the element-wise multiplications and the accumulations required by Steps 2 and 3 of the proposed approach; they receive the T_M remapped windows and the filters coefficients as arranged,

in the meantime, by the WB. When CONVs are executed with kernel sizes greater than 1, the RU is bypassed, thus the IFB and WB feed directly the CTCE. In the case of 1×1 CONVs, both the IFB and the RU are bypassed, thus inputting the *ifmaps* directly to the CTCE.

Within the CTCE, multiplications and accumulation are performed, respectively, through two different pipeline sub-circuits, here named *Type-A* (TA) and *Type-B* (TB), following a similar approach as that described in Section 3.3.2. Indeed, each tile consists of several *Processing Elements* (PEs). The PEs inside the TAs execute MACs, whereas the PEs within the TBs perform accumulations between two operands. Figure 3.32 illustrates examples of a TA having 4 PEs and a TB consisting of 2 PEs.

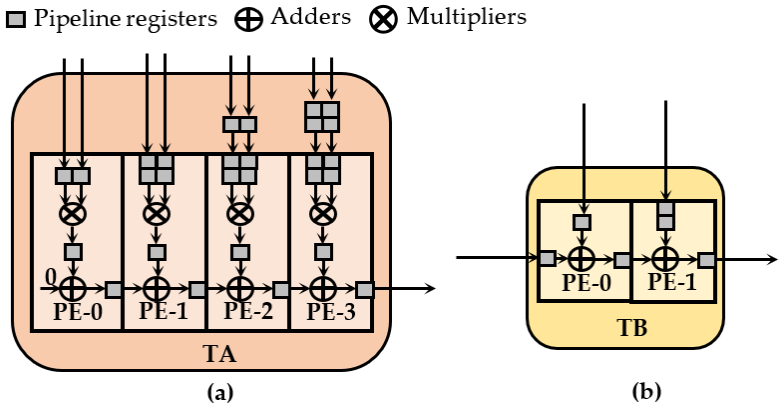


Figure 3.32 Examples of **(a)** TA with 4 PEs and **(b)** TB with 2 PEs.

In order to provide a flexible architecture, suitable to perform both CONVs and TCONVs at different operating conditions, the CTCE uses several TA and TB circuits, which are connected to each other by multiplexers. The latter allow to activate a specific path within the CTCE, depending on the currently processed kernel size. Taking into account that, at the parity of the kernel size, the TCONVs are more complex than CONVs, the employed sub-circuits TAs and TBs have been organized to meet the computational capability required by TCONVs in the worst case, thus intrinsically being able to satisfy also the computational requirements of CONVs. Figure 3.33 illustrates the design of the CTCE by reporting one of the configurations actually tested. Depending on which operation must be currently performed (i.e., CONVs or TCONVs) and based on the filter size *k*, the auxiliary multiplexing logic also depicted in Figure 3.33 coordinates the cooperation between TAs and TBs and guarantees that the different supported operations

are performed correctly. The gray boxes represent the pipeline stages that, being deep as indicated by the reported numbers, time-align the performed computations.

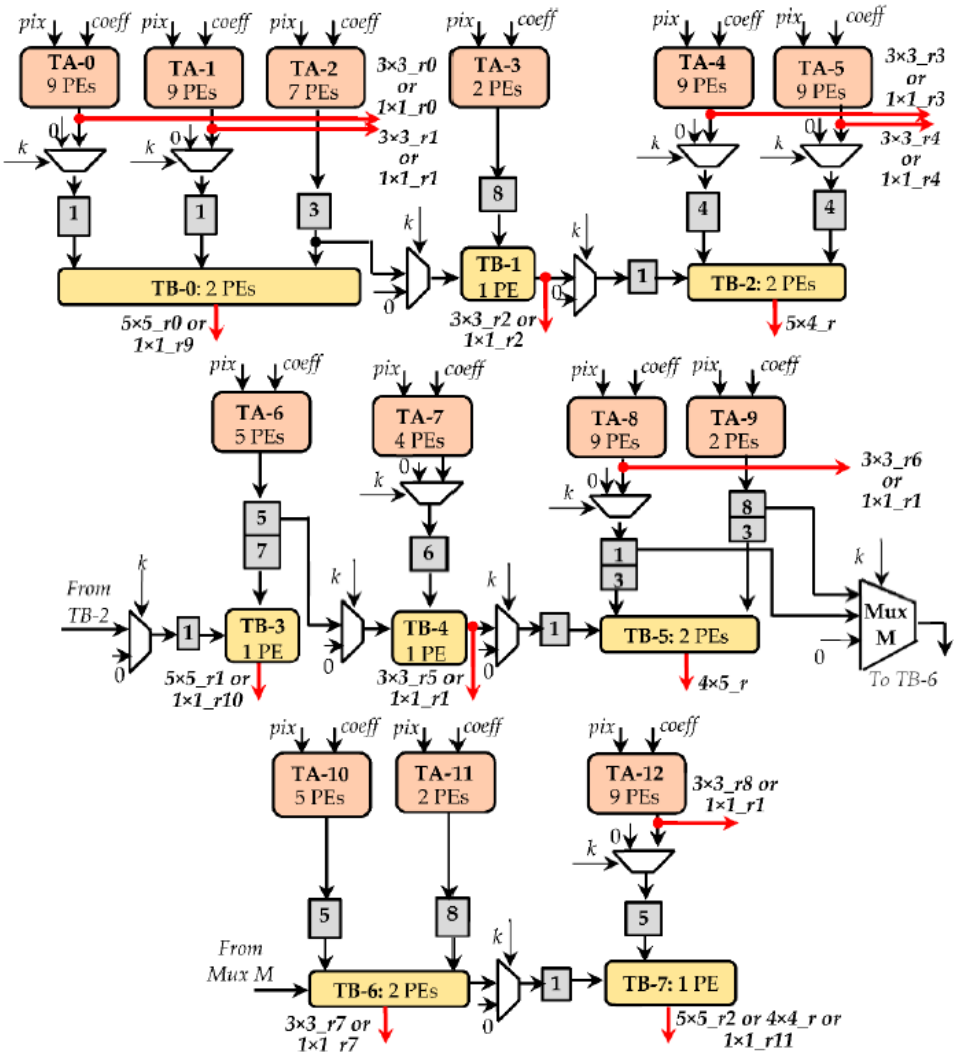


Figure 3.33 An example of the computations performed by CTCE when *k* is up to 9 and S_D = 2. The kernel size *k* can assumes the values 1, 3, 4, 5, 7, 9.

By the TCONV viewpoint, the circuit in Figure 3.33 complies with a 9×9 kernel at S_D=2. In this regard, 13 TAs and 8 TBs are properly arranged to accomplish Steps 2 and 3 of the proposed method. The TAs, consisting of 81 PEs, exploit as many as multipliers to execute the element-wise matrix multiplication (step 2). Accumulators internal to the TAs, in conjunction with the 12 PEs provided by the TBs, execute the chessboard accumulations (step 3) to furnish the parallel results as in (1). In Figure 3.33, the $S_D \times S_D$ parallel outputs are labelled as *5×5_r0*, *5×4_r*, *4×5_r* and *4×4_r*, respectively. Subsequently, the external module ATs for TCONVs sums the referred outputs to the homologous results furnished by the other CTCEs operating in parallel. Other than the

9×9 TCONV, the referred TAs and TBs can be used to perform different CONVs, considering the same circuit of Figure 3.33:

- twelve 1×1 CONVs, whose results are *1×1_ru*, with *u= 0, …,11*;
- nine 3×3 CONVs, with the furnished results being *3×3_rx*, with *x=0,…,8*;
- three 5×5 CONVs, whose results are *5×5_ry*, with *y=0,…,2*;
- one 7×7 CONV; in this case the results *5×5_r0* and the *5×5_r1* are added by the external module ATs for CONVs;
- one 9×9 CONV; in such a case the results *5×5_r0*, the *5×4_r*, *4×5_r* and *4×4_r* are summed up by the external module ATs for CONVs.

Finally, it is worth noting that, in order to make the above-described CTCE able to support different up-sampling factors, just a few and simple modifications are required either on the viable paths or on the compositions of the sub-circuits TAs and TBs.

3.4.4. EXPERIMENTAL RESULTS

As a case study, the super-resolution imaging was referred to and the proposed approach was adopted to accelerate the popular FSRCNN model [31]. To this purpose, the hardware architecture described in Section 3.4.3 was tailored to comply with the configurations summarized, layer by layer, in Table 3.5. It is worth noting that how many instances of the CTCU module are used (i.e., the value of the parameter *R* set at design time) is established at design time to achieve a better trade-off between speed performances and area occupancy. For the referred case study, *R*=12 was chosen since it well complies with the requirements of the overall network model and allows reducing the inference time by more than 90% with respect to the case in which *R*=1. Indeed, while the configuration *R*=1 takes ~58.6 ms, the parallelized counterpart requires only ~5.2 ms. Table 3.4 also reports the parameter P_N that indicates how many output values are computed in parallel for each of the T_N furnished *ofmaps*. When the TCONV layer is executed, P_N is equal to $S_D \times S_D$, with S_D being set to 2, 3 or 4, as established at design time. The parameters *M, N, k* are elaborated by the FSM that: 1) run-time configures the proposed hardware accelerator, thus ensuring that T_M and T_N change properly as required by each layer; 2) scans the various computational steps, by determining in which cycles data must be read or not, as well as at which time outputs can be delivered.

Table 3.5 The run-time configurations of the novel hardware accelerator for adaptive CONVs/TCONVs related to the FSRCNN.

Layer	Op Mode	M	N	k	S_D	T_M	T_N	P_N
1	CONV	1	56	5	1	1	$3\times R$	1
2	CONV	56	12	1	1	56	R[1]	1
3	CONV	12	12	3	1	9	R	1
4	CONV	12	12	3	1	9	R	1
5	CONV	12	12	3	1	9	R	1
6	CONV	12	12	3	1	9	R	1
7	CONV	12	56	1	1	12	$3\times R$	1
8	TCONV	56	1	9	2 or 3 or 4	R	1	4 or 9 or 16

[1] R is set at design time.

The novel accelerator exploits fixed-point arithmetic with activations and filters quantized, respectively, to 16 and 10 bits. Such a choice, which arises from a preliminary analysis conducted to evaluate the impact of different quantization levels on the quality of reconstructed images, allows to improve the area occupancy by 60% and 18% with respect to 32- and 16-bits fixed-point versions, respectively, with limited effects on the quality of the reconstructed images. Three different versions of the novel accelerator, each performing the TCONV layer with a specific up-sampling factor, were designed. Implementation results, obtained utilizing the Xilinx XC7K410T [88] and XCZU9EG FPGA [60] devices and the Vivado Design Suite (v.2019.2) [71], are collected in Table 3.6. There, data collected are related to: (a) the amount of occupied LUTs, FFs, BRAMs and DSPs; (b) the power consumption, estimated through the Switching Activity Values File (SAIF) that, referring to several benchmark images, has taken into account the real activities of all nodes within the analyzed circuit; (c) the performance, evaluated across different metrics, such as the maximum running frequency and the Giga Operations per Second (GOPS), which is the ratio between the overall computational complexity of the referred model and the inference time; (d) the energy efficiency (GOPS/W), which is defined as the ratio between the GOPS and the power consumption. Figures 3.34-3.36 report the relevant percentage changes, with respect to counterparts, in terms of resource utilization, clock frequency and energy efficiency.

Table 3.6 Characterization of the novel hardware accelerator for adaptive CONVs/TCONVs and state-of-the-art comparisons.

Accelerator			New	New	[47]	[84]	[86]	[89]
FPGA Device			XCK410T	XCZU9EG	XCK410T	XCVU095	XCZU9EG	XCVU9P
FSRCNN*(x,y,z,w)*			(56,12,4,9)	(56,12,4,9)	(25,5,1,7)	(56,12,4,8)	(32,5,1,9)	(32,5,1,-)[2]
Variable *k*,S_D			Yes, No	Yes, No	No, Yes	Yes[1], Yes	No, No	No, No
Supported S_D			2, 3, 4	2, 3, 4	2, 3, 4	2, 3, 4	2	2
#bits (activations, filters)			(16, 10)	(16, 10)	(13, 13)	(16, 8)	(16, 16)	(14, 10)
Resources	**LUTs**	**S_D=2**	63.1k	60.6k			168.6k	94k
		S_D=3	56.9k	54.6k	167k	42k	-	-
		S_D=4	77.2k	74.4k			-	-
	FFs	**S_D=2**	101.2k	101.2k			NA	19k
		S_D=3	85.5k	85.5k	158k	20k	-	-
		S_D=4	122.8k	122.8k			-	-
	BRAMs [Mb]	**S_D=2**	14.3	12			10.9	0.4
		S_D=3	14.3	12	7.2	4.85	-	-
		S_D=4	18.6	15.5			-	-
	DSPs	**S_D=2**	1212	1212			746	2146
		S_D=3	1140	1140	1512	576	-	-
		S_D=4	1296	1296			-	-
Performance	**Freq. [MHz]**		227	250	130	200	200	200
	GOPS	**S_D=2**	654.3	720.6	780	605. 6	795.2 [3]	541.4 [4]
		S_D=3	1223.5	1347.5	1576.3	1086.1	-	-
		S_D=4	2022.2	2227	2691	1868.8	-	-
	Power [W]	**S_D=2**	3.6	3.8	5.4	3.71	NA	6.9
		S_D=3	3.5	3.85	-	-	-	-
		S_D=4	3.9	4	-	-	-	-
Eff.	**GOPS/ W**	**S_D=2**	181.8	189.6	144.9	163. 7	NA	78.5
		S_D=3	349.6	350	293	293.5	-	-
		S_D=4	518.5	556.8	500.2	505.1	-	-

[1] The CONV kernel sizes range from 1×1 to 4×4.

[2] The TCONV layer is replaced with an ESPCN layer.

[3] Calculated considering the 120.4 frames per second declared in [86].

[4] Calculated considering the 60 frames per second declared in [89].

Table 3.6 also summarizes the implementation characteristics of representative state-of-the-art FPGA-based designs that, being devoted to the acceleration of CNNs for the SR imaging, were selected as the direct competitors, even though they refer to somewhat different models from the original FSRCNN presented in [31]. While the designs proposed here were characterized referring to the whole model reported in Table 3.4, thus performing four cascaded CONV layers with k=3 (i.e., the layers 3, 4, 5, and 6), the accelerators presented in [47, 86, 89] refer to simplified models and perform only one CONV layer with k=3. As a further simplification, to relieve the computational load, the design described in [89] replaces the TCONV with an Efficient Sub-Pixel CONV (ESPCN) layer that provides up-sampled *ofmaps* through a periodic shuffling. Conversely, the reconfigurable design presented in [84] refers to the original FSRCNN

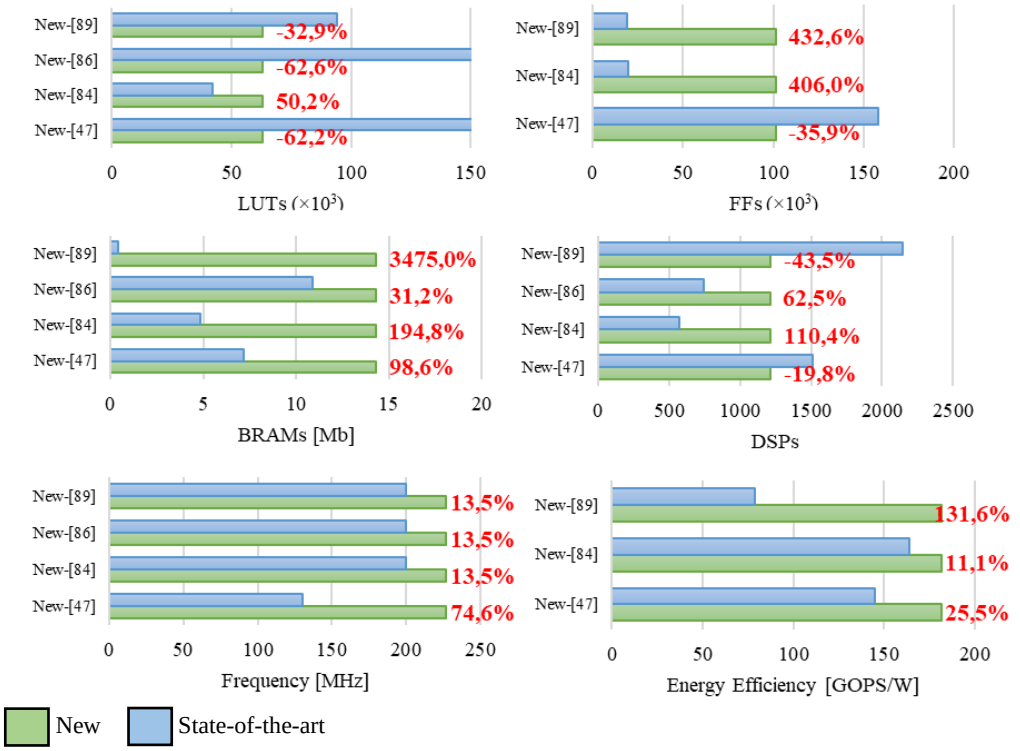


Figure 3.34 Percentage change comparisons (*S*=2, XCK410T): resources, frequency, energy efficiency.

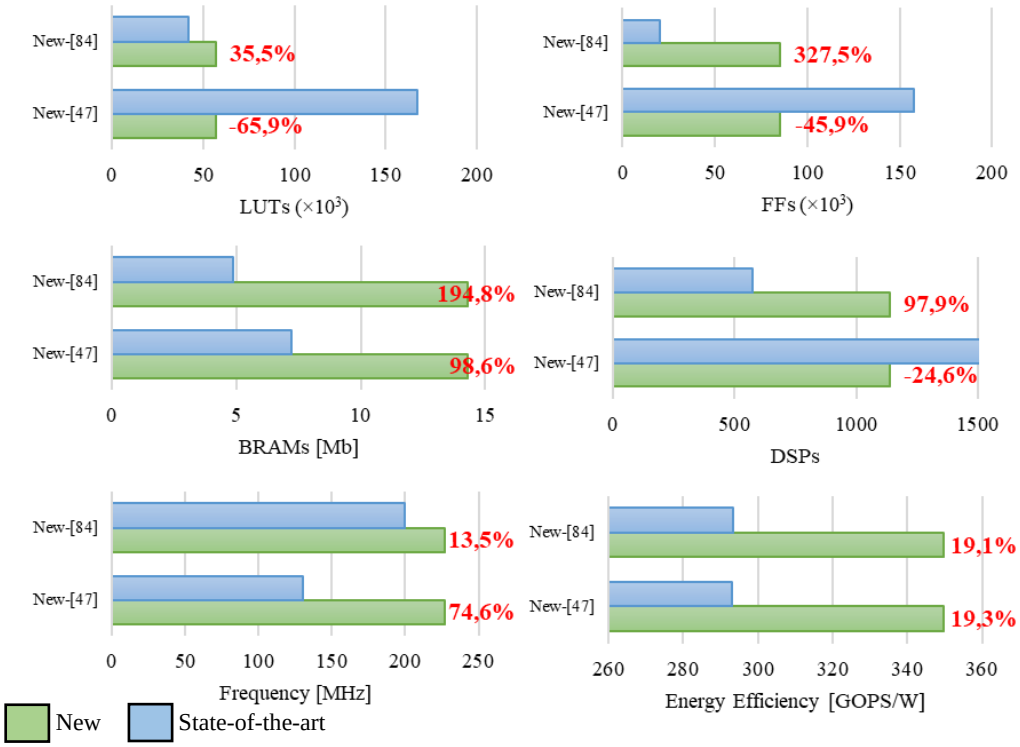


Figure 3.35 Percentage change comparisons (*S*=3, XCK410T): resources, frequency, energy efficiency.

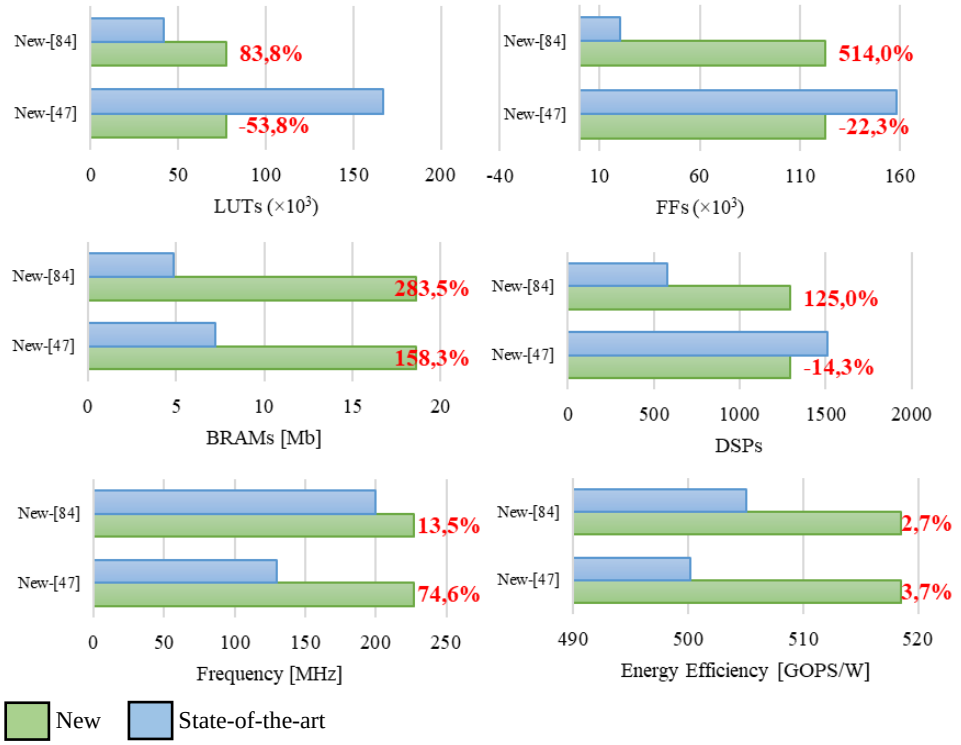


Figure 3.36 Percentage change comparisons (*S*=4, XCK410T): resources, frequency, energy efficiency.

model, but it performs CONVs with kernels sizes ranging from 1×1 to 4×4 and changes the TCONV kernel size from 9×9 to 8×8.

In order to point out the main differences between the network models accelerated by the compared designs, they are referenced in Table 3.6 as FSRCNN(*x*,*y*,*z*,*w*). There, *x*, *y*, *z* and *w* are, respectively, the number of *ofmaps* outputted by the first CONV layer, the number of *ofmaps* furnished by the subsequent CONV layers, the last excepted, the number of cascaded CONV layers with kernel size *k*=3, and the TCONV kernel size.

By examining the results summarized in Table 3.6, it can be observed that the proposed accelerators lead to lower power consumptions, though referring to the most complex CNN model, due to their particularly efficient flexible architecture. The power savings achieved with respect to [47] and [89] come from the capability of the proposed designs of run-time adapting themselves to different CONV kernel sizes. Without such a capability, the implementations characterized in [47, 86, 89] must employ a different ad-hoc architecture for each layer, thus negatively affecting the power consumption and the resources requirements.

In comparison to [47], the proposed XCK410T-based implementations save more than 53.8% LUTs, 22.3% FFs and 14.3% DSPs, and improve the energy efficiency by up to 25.5%, which is also the result of avoiding multiplications with sparse filters as required by the referred counterpart. These advantages are obtained even though the CNN model referenced in the novel designs is quite more complex.

The design demonstrated in [84] is particularly efficient in terms of occupied hardware resources. However, the novel accelerators implemented on the XCZU9EG chip consume ~3% less power and achieve up to ~16% higher GOPS, although they perform CONVs and TCONVs with greater kernel sizes and coefficients bit width.

The accelerator presented in [86] sacrifices a certain amount of hardware resources to deeply pipeline the circuit, thus reaching the highest GOPS. However, such an advantage is obtained costs more LUTs, as a consequence of the Winograd algorithm implementation: indeed, at a parity of implementation chip and S_D, [86] performs ~9.5% more GOPS, but the amount of occupied LUTs is ~2.8 times higher.

Finally, from Table 3.6 it can be seen that, despite the simplifications introduced to reduce the computational complexity of the referred CNN model, at a parity of the up-sampling factor S_D=2, the design proposed in [89] occupies ~48.9% and ~77% more LUTs and DSPs than the novel accelerator targeting the XCK410T chip. Furthermore, this proposal exhibits considerably improved speed performances and power consumption, which lead to an energy efficiency ~2.3 times higher.

For the sake of a fair analysis, the FSRCNN models above referenced were compared also in terms of the quality achieved at different up-sampling factors.

Software routines modelling the proposed accelerators were written to process the popular Set-5 [90], Set-14 [91] and B100 [92] datasets and to evaluate the Peak Signal to Noise Ratio (PSNR) and the Structural Similarity Index Measure (SSIM) [93]. Table 3.7 clearly shows that the strategy here adopted to transform TCONVs into CONVs does not affect the quality of reconstructed images. Indeed, in most of the analyzed cases, slightly improved PSNR and SSIM are achieved with respect to [47, 84, 89]. The competitor [86] is not included in the comparison because the quality metrics furnished in the original paper are related to quite different datasets.

Finally, Figure 3.37 shows a sample image from the Set-5 dataset that was up-sampled by using the proposed approach at S_D=2. As expected, details are well re-constructed and, in this case, the achieved PSNR is 31.48 dB.

Table 3.7 Quality results of the novel hardware accelerator for adaptive CONVs/TCONVs and state-of-the-art comparisons.

		New		[47]		[84]		[89]	
Dataset	S_D	PSNR	SSIM	PSNR	SSIM	PSNR	SSIM	PSNR	SSIM
Set-5	**2**	35.68	0.9459	36.40	0.9527	35.85	NA	36.42	0.9529
Set-14	**2**	31.34	0.8650	32.21	0.9047	NA	NA	32.27	0.9045
B100	**2**	30.28	0.8765	31.15	0.8858	NA	NA	31.18	0.8859
Set-5	**3**	32.52	0.8816	32.48	0.9043	32.03	NA	NA	NA
Set-14	**3**	29.04	0.7975	29.03	0.8146	NA	NA	NA	NA
B100	**3**	28.27	0.7854	28.25	0.7808	NA	NA	NA	NA
Set-5	**4**	30.6	0.8577	30.17	0.8532	29.48	NA	NA	NA
Set-14	**4**	27.52	0.7480	27.24	0.7414	NA	NA	NA	NA
B100	**4**	26.90	0.7135	26.71	0.7041	NA	NA	NA	NA

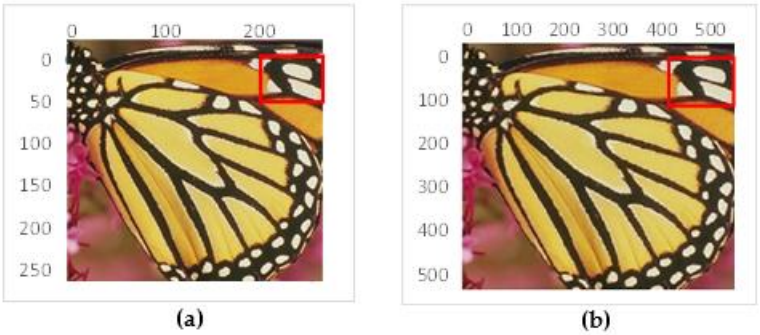


Figure 3.37 Sample result obtained with S_D=2: **(a)** the original image; **(b)** the reconstructed image.

3.5. SUMMARY

Several VHDL-based implementations of convolution-based layers for deep learning were presented in this chapter.

A novel DCONV engine supporting the existing ASPP approach was proposed to accelerate CNN models for semantic segmentation. When realized within the Xilinx Zynq-7000 XCZ7020 SoC device, the novel hardware accelerator achieves the 181 MHz running frequency, occupies just 7.3%, 2.2%, 26.1% and 21% of the on-chip LUTs, FFs,

DSPs and BRAMs and dissipates only 265 mW. Purposely performed tests showed that this engine efficiently scales with the sizes of feature maps and kernels.

To address redundant computations in conventional deconvolution layers, an effective parametric architecture using the IOM strategy was proposed and examined over both low- and high-end devices by adequately scaling the adopted parallelism. When exploited to accelerate the DCGAN model within the Xilinx Zynq XC7Z020 SoC device, it reaches 72 GOPS, by dissipating 500mW at a frequency of f=200MHz. Thanks to the increased parallelism exploitable, more than 900 GOPS can be executed when the high-end Virtex-7 XC7VX690T device is used as the implementation platform. Moreover, in comparison with state-of-the-art competitors implemented within the Zynq XC7Z045 device, the system proposed here reaches a computational capability up to ~20% higher and saves more than ~60% and ~80% of power consumption and logic resources, respectively, using ~5.7× less on-chip memories.

The reconfigurable architecture for CONV Layers discussed in Section 3.3, able to run-time adapt itself to different kernels, strides and *fmap* sizes, dissipates only ~2.75 W at the 150 MHz running frequency and occupies ~13.1%, ~13.2%, ~39% and ~88.9% of the available LUTs, FFs, BRAMs and DSPs, respectively, when the mid-end XC7Z045 device is used as the evaluation platform. When compared to several state-of-the-art counterparts, it exhibits the lowest resource requirement and power consumption with the most favorable effective-throughput/parallelism ratio.

Finally, an efficient hardware-oriented algorithm to accelerate both CONVs and TCONVs was presented in order to simplify the data acquisition policy for both the layer types. The proposed strategy was validated using a reconfigurable hardware accelerator purposely designed to adapt itself to different operating modes set at run-time. When characterized using the Xilinx XC7K410T FPGA device, it achieves a throughput of up to 2022.2 GOPS and, in comparison with state-of-the-art competitors, it reaches an energy efficiency up to 2.3× higher, without compromising the overall accuracy.

4. HIGH-LEVEL SYNTHESIS FOR CONVOLUTIONAL NEURAL NETWORKS

This chapter focuses the attention on the design of accelerators for TCONV-based CNNs using the High-Level Synthesis (HLS) paradigm. The latter design flow, indeed, has proven successful to implement in hardware quantized CNNs for image processing. However, research about quantized CNNs using TCONV Layers is still under-explored.

First, a preliminary analysis about the impact of bit-width over the accuracy of state-of-the-art models is provided. These experiments, performed on GPUs, can be thought as a feasibility study to determine how deep quantization can be for later management on FPGAs. Afterwards, proper case-studies of FPGA-based hardware implementations are presented, through a characterization in terms of resources, latency and power that is constrained over proper design-space exploration.

4.1. ACCURACY ANALYSIS OF QUANTIZED NEURAL NETWORKS USING TRANSPOSED CONVOLUTIONS

4.1.1. BACKGROUND

As stated in Section 2.4, hardware architectures that overcome CPUs and GPUs in terms of performance-per-Watt are imposing themselves forcefully. This is particularly true in those environments that benefit from high performance and low power (e.g., Edge Computing [94]). FPGAs are representative examples of hardware platforms to accommodate DL workloads by trading-off speed performances, accuracy, area, and power consumption [95]. However, these architectures may be relatively small to accommodate wide models for high-quality image processing. As a consequence,

compression techniques are mandatory, including filters pruning [96], knowledge distillation [97] and data quantization [18]. The latter has proven to be effective to deal with NNs using FC Layers and conventional CNNs for classification when considering FPGAs as the target platform, by exhibiting negligible loss of accuracy even after significant compression [98, 99, 100].

In addition, more sophisticated models dealing with TCONV-based image up-sampling could strongly benefit from a reduced precision of activations and weights to meet a reasonable trade-off in terms of area, speed and power. Nevertheless, to the best of our knowledge, the effects caused by the quantization of TCONVs are still underexplored. For instance, authors in [67] evaluated remote sensing image segmentation using the compressed U-Net [23], where data were quantized to 8 bits. Super resolution imaging was examined in [16], where 16-bit activations and 10-bit weights were exploited. A thorough analysis of deep quantized generative networks was proposed in [101], where authors observed that good quality could be maintained even using very low-precision filters. Collaborative inference was evaluated in [102], where a single autoencoder was used within deep neural networks to minimize the overall latency through both weight-sharing and dynamic quantization down to 2 bits.

The aim of this section is to provide a systematic evaluation of the effects achieved by trading-off accuracy and deep quantization of several TCONV-based neural networks [19]. This preliminary investigation is needed to verify the suitability of implementing very deep quantized models on FPGAs.

Three applications were considered:

- image compression/decompression supported by an autoencoder architecture;
- image generation performed through the Deep Convolutional Generative Adversarial Network (DCGAN) [22];
- image segmentation performed by the U-Net [23].

Specifically, the achieved accuracy was evaluated using subjective visual inspection as well as objective analytical metrics, including the *Peak Signal-To-Noise Ratio* (PSNR), the *Inception Score* (IS) [103], the *Fréchet Inception Distance* (FID) [104] and the *mean Intersection over Union* (mIoU). The neural networks were trained over widely used public datasets: MNIST [105], Fashion-MNIST [106], CIFAR-10 [107], CelebA [108], Oxford-IIIT Pet [109] and Cityscapes [110].

After a brief review about the adopted quantization approach, the CNNs used as case-studies as well as the accuracy metrics are presented. Then, the experiments are discussed in detail.

4.1.2. THE QUANTIZATION METHOD

Quantization [111] is a compression technique that transforms 32-bit floating point data into low-precision fixed-point words. DL can use quantization either during or after training. The former scenario is known as Quantization-Aware Training (QAT), while the latter is the Post-Training Quantization (PTQ) approach. Both the strategies can be used to compress: (a) the pixels of the input images, (b) the trainable weights, and (c) the non-linear activations provided by each layer. The experiments that will be later presented in Section 4.1 refer to the quantization of the weights and the activations.

Among different high-level framework to investigate the behavior of DL models, *PyTorch* [112] has proven successful for simplicity, flexibility and the dynamic management of computational graphs. *Brevitas* [113] is a research library from Xilinx that can be used within *PyTorch* to simulate QNNs suitable for hardware implementation and adopting the QAT approach.

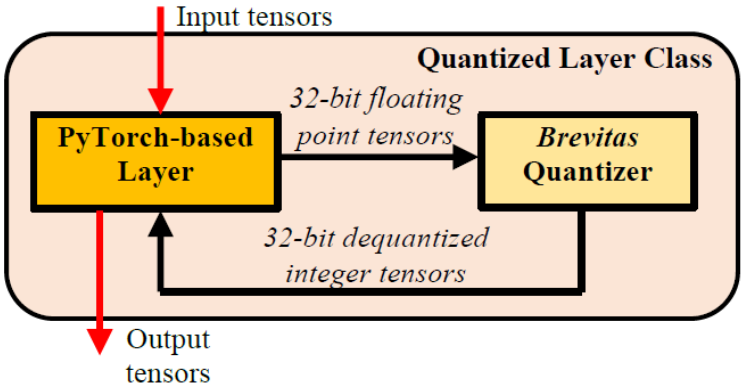


Figure 4.1 Block diagram of the generic *Brevitas*-based Quantized Layer class.

In *Brevitas*, each quantized layer inherits the conventional 32-bit floating point class, which is linked to a quantizer. As schematized in Figure 4.1, the *Quantized Layer Class* quantizes the incoming tensors to integers, and translates them to a dequantized format, compliant with the related full-precision layer. Among different quantization strategies, this work refers to the uniform affine quantization defined in (1):

$$Quant(v) = scale \times round\left(\underset{min,max}{\text{clamp}}\left(\frac{v}{scale}\right)\right) \quad (1)$$

At the beginning, the generic 32-bit floating point value *v* is scaled by a proper *scale* factor, computed from backpropagated statistics of the input tensors. The latter considers both the actual values range of the input tensor and the domain provided by the given bit-width (i.e., the top and the bottom bounds). Then, the scaled value *x* is clamped according to (2) where, in the case of signed number representation, $min = -2^{N-1}+1$, $max = +2^{N-1}-1$, and *N* is the bit-width. It is worth underlining that *min* is chosen as $-2^{N-1}+1$ instead of $min=-2^{N-1}$ for symmetry purposes.

$$\underset{min,max}{\text{clamp}}(x) = \begin{cases} min & x < min \\ x & min \leq x \leq max \\ max & x > max \end{cases} \quad (2)$$

Finally, the nearest-integer rounding provides the discretized integer value. In order to make it compliant with the generic 32-bit conventional *PyTorch* class, the latter is represented in a dequantized format, obtained by multiplying the quantized tensor by the *scale* factor.

4.1.3. THE QUANTIZED NEURAL NETWORKS USED AS CASE-STUDIES

The first QNN model considered is the *Convolution Autoencoder* (CA) and depicted in Figure 4.2. The encoder processes *ch* input channels, with *ch* varying according to the colors channels of the given dataset (i.e., *ch*=1 for gray-scale images; *ch*=3 for RGB images), by means of two 3×3 CONV Layers, each followed by the *ReLU* as non-linear activation [25], and a *Max Pooling* Layer (MaxPool) that infers down-sampling to the incoming *fmaps*. Conversely, the decoder consists of two 3×3 TCONV Layers, with the up-sampling factor *S*=2, that progressively quadruple the size of the *fmaps*. ReLU [25] and Sigmoid [24] equip the decoder with the non-linearity.

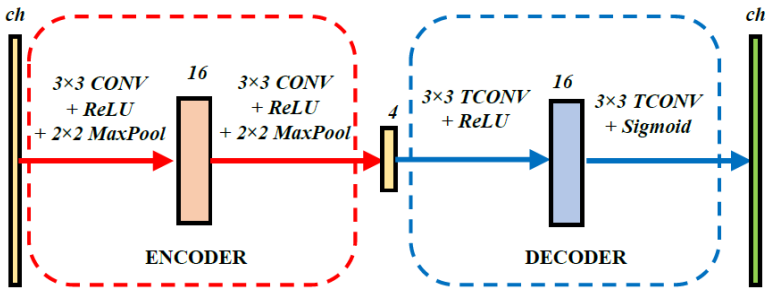


Figure 4.2 The Convolution Autoencoder.

Figure 4.3 depicts the second analyzed model: the DCGAN architecture [22], responsible for adversarial learning through the competitive generator and discriminator portions. The former provides *ch* channels by means of four 4×4 TCONV Layer that infer informative content to data distribution and up-sample *fmaps* to the desired resolution. The generator exploits ReLU and Tanh [24] to introduce non-linearity, and Batch Normalization (BN) to achieve more stable training [26]. On the contrary, the discriminator progressively down-samples *fmaps* and extracts features of interest by means of strided CONVs (with stride=2) followed by either LeakyReLU [114] or Sigmoid activations [24].

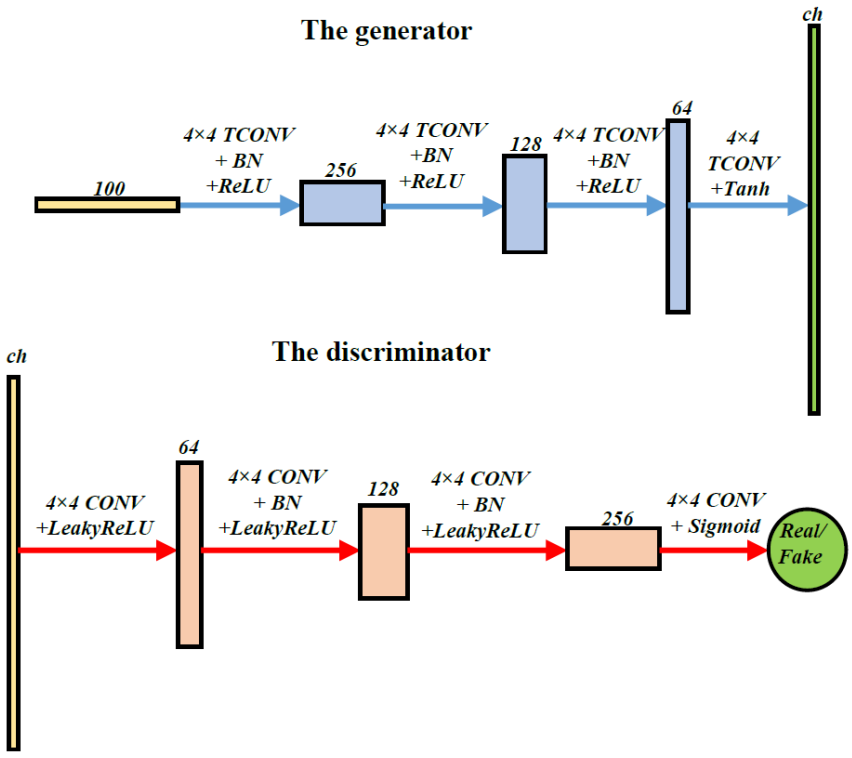


Figure 4.3 The Deep Convolutional Generative Adversarial Network [22].

The encoder-decoder U-Net architecture [23], chosen as the third analyzed model, is depicted in Figure 4.4. The encoder consists of a sequence of down-sampling blocks, each made of a stack of CONV Layer, BN, ReLU and MaxPool, which progressively abstract the features representation. The decoder progressively recovers the resolution of data by exploiting TCONV Layers that also strengthen the localization ability [23].

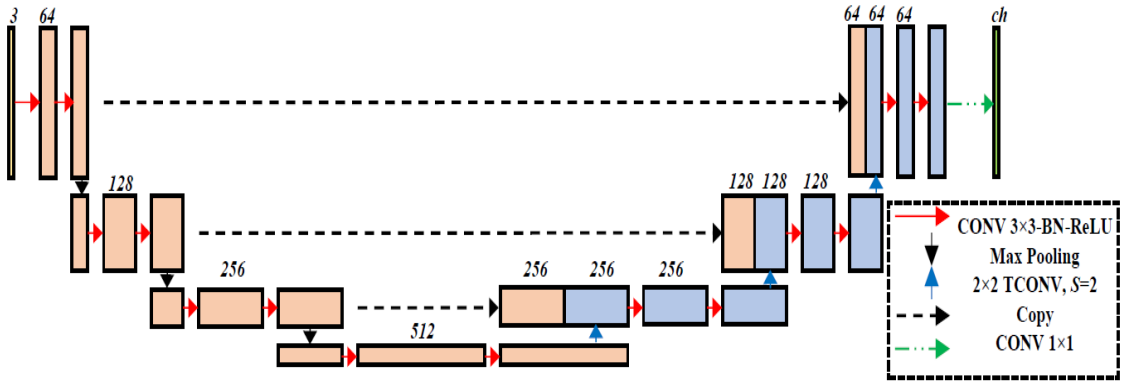


Figure 4.4 The U-Net architecture.

4.1.4. THE DATASETS

As previously stated, in order to consider a wide range of operating conditions, the experiments were conducted using the datasets MNIST [105], Fashion-MNIST [106], CIFAR-10 [107], CelebA [108], Oxford-IIIT Pet [109], and a tiny version of Cityscapes [110]. MNIST collects 28×28 grayscale images of handwritten digits: 60,000 of them are used for training, whereas 10,000 for validation. The Fashion-MNIST uses Zalando's clothes images and follows the same structure of MNIST. The CIFAR-10 dataset is made of 60,000 32×32 pictures, representing RGB objects that belong to 10 classes. The Large-scale CelebFaces Attributes dataset (CelebA) is made of 202,599 images representing human faces. The Oxford-IIIT Pet dataset consists of more than 7,000 images showing cats and dogs breeds. The tiny version of the Cityscapes dataset collects 3475 256×512 images related to street scenes.

The CA was trained over the dataset MNIST, Fashion-MNIST and CIFAR-10. MNIST, CIFAR-10 and CelebA were used to train the DCGAN network. Finally, U-Net performed semantic segmentation over the Oxford-IIIT Pet and Cityscapes datasets.

4.1.5. THE ACCURACY METRICS

The accuracy achieved by the above described QNNs was evaluated through subjective and analytical quality metrics. *Visual inspection* (VI) was selected as the subjective metric, by allowing:

- to judge the similarity of reconstructed images for image decompression.

- to recognize the verisimilitude of DCGAN generated images;
- to establish if segmented images are split into easily identifiable regions, when the U-Net model is considered.

However, subjective evaluation is not able to determine: (a) the exact similarity level between the original image and the decompressed one, (b) the variety of a huge generated dataset of images using GAN models, (c) the segmentation accuracy of each image (i.e., the correct classification of each pixel). To address the latter issues, analytical metrics were used. They include the *Peak Signal-To-Noise Ratio* (PSNR), the *Inception Score* (IS) [103], the *Fréchet Inception Distance* (FID) [104], and the *mean Intersection over Union IoU* (mIoU).

The PSNR is an index of similarity, measured in decibel (dB) and derived by the comparison of two images. It is consists of the logarithm of the ratio between the maximum pixel value and the cumulative mean square error between the compared images. The higher the PSNR, the better the similarity.

The *Inception Score* (IS) and the *Fréchet Inception Distance* (FID) are used to evaluate the quality of synthetic images and the variety of the represented classes for the DCGAN model. The IS exploits the GoogleNet Inception Network [115], trained over the ImageNet dataset [116], to classify the generated dataset from the DCGAN. The overall classification provides a probability distribution, called *marginal distribution*, which indicates both the accuracy of the synthetic images and the variety of the dataset. The higher the IS the better the task carried out by the DCGAN. The FID metric provides a further element of evaluation with respect to the IS: indeed, it is able to determine the similarity between the generated dataset and the original dataset used for the adversarial learning. Low values of FID mean strong image generation.

The *Intersection over Union* (IoU) measures the similarity between the segmented class predicted by a QNN model and a ground-truth, by computing the ratio between the overlapping area and the union area. While the former is the area that the prediction and the ground-truth have in common, the latter is the area encompassed by both. The *mean IoU* (mIoU) averages the IoU over all the segmented classes and its domain is the range (0, 1). The higher the mIoU, the better the similarity.

4.1.6. TRAINING SETTINGS

The Convolution Autoencoder (CA) was trained over 100 epochs, using batches of 64 elements, the *Mean Square Error* as loss function and *Adam* as optimizer at a learning rate equal to 0.001. Both the activations and the weights were quantized at the same bit-width. Experiments were conducted using the NVIDIA Tesla T4 GPU [117] within the Google Colaboratory [118] environment.

According to MNIST, the DCGAN was trained over 30 epochs, using batches of 128 elements, the *Cross Entropy* as loss function and *Adam* as optimizer (learning rate 0.0002, β_1=0.5, β_2=0.999). To avoid overconfidence, label smoothing was exploited, thus penalizing the labels by 10%. In this case, data and weights were quantized to bit-widths ranging between 3 and 8. To train the DCGAN over CIFAR-10 and CelebA, 100 and 30 epochs were performed, respectively, with the same settings used for MNIST and in the range 8-4 bits. CelebA images were resized to 64×64 tiles for batch uniformity and quicker training. Also in this case, the NVIDIA Tesla T4 GPU [117] was used as the training platform.

U-Net was trained over 100 epochs, using batches of 16 elements, the *Cross Entropy* as loss function and *Adam* as optimizer, with the learning rate set to 0.001 and 0.01 for the Oxford-IIIT Pet and Cityscapes datasets, respectively. Oxford-IIIT Pet images were resized to 64×64 tiles, whereas Cityscapes images were center-cropped to 128×128 frames. Experiments were conducted using the NVIDIA Tesla K80 GPU [119] and the bit-width range was spanned in the range 8-2 bit.

4.1.7. VISUAL INSPECTION RESULTS

The VI analysis of the CA tests the ability of the autoencoder to capture the *essence* of the input images, by examining how well the outputs are similar to inputs. According to the results, quantization between 8 and 4 bits provides acceptable accuracies.

Sample images, from MNIST and Fashion-MNIST experiments, are reported in Figure 4.5 and show how coarser is the image reconstruction achieved using 3 bits and how many details are lost, especially when the Fashion-MNIST dataset is referred to. To further validate the VI, 10,000 images provided by the CA were provided to the LeNet-5 classifier [120] to determine the classification behavior. The LeNet-5 classifier was also

used to infer results over the ground-truth (GT) dataset, thus providing a baseline for comparisons.

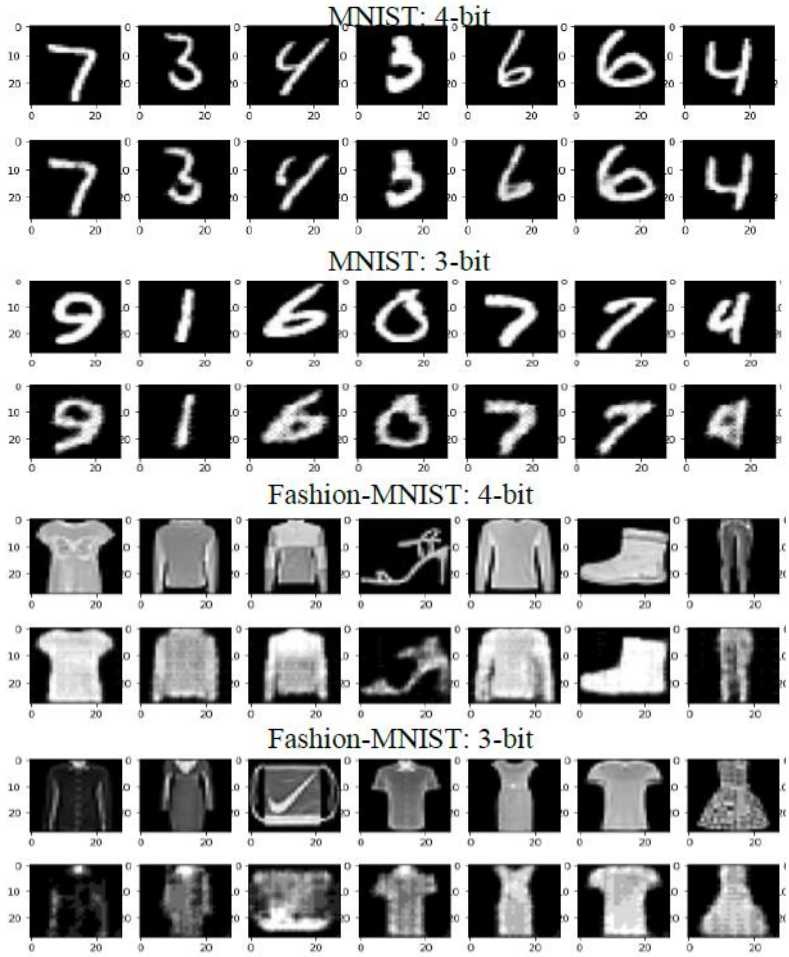


Figure 4.5 Samples of reconstructed images from MNIST and Fashion-MNIST. For each scenario, the top images refer to the original dataset, while the bottom ones are extracted from the CA.

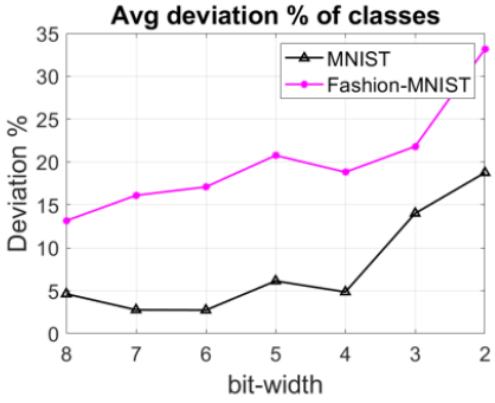


Figure 4.6 Average deviation of classes distribution w.r.t. the classification of the ground-truth test. Baseline: LeNet-5 over MNIST (Fashion-MNIST) with accuracy 99% (90%).

For the 10 classes, the average percentage change of the occurrences with respect to the baseline is reported in Figure 4.6 at the different bit-widths. The trend clearly shows that lower quantized bit-widths make the deviation even more evident, given that the quantized TCONV Layers increase the number of misclassified images.

Finally, it is worth underlining that CIFAR-10 was not considered for VI due to the poor resolution of its 32×32 RGB images.

The VI analysis of DCGAN shows that, when trained over MNIST, the model exhibits satisfactory generation for the range 8-5 bits. Conversely, when the 4-bit quantization is exploited, though fake images represent digits quite well, the generator provides limited classes, thus incurring in the training failure named *mode collapse* [121]. Lower quantization (i.e., 3-bit) introduces extra noise that corrupts the informative content of each image.

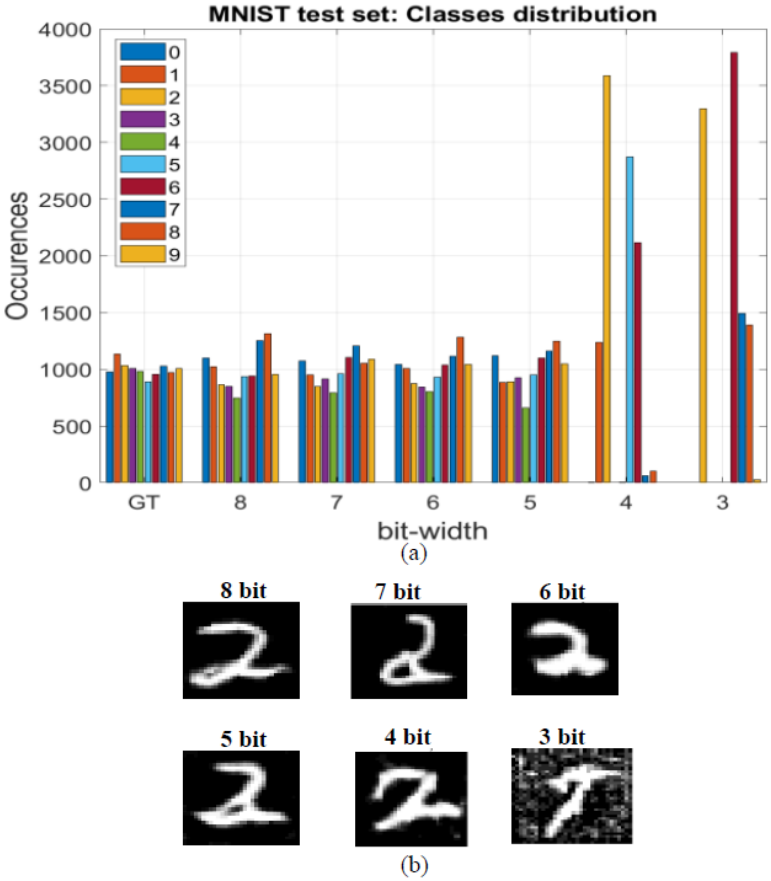


Figure 4.7 (a) The classes distribution of the synthetic MNIST datasets over the bit-width range. GT refers to the ground-truth dataset. **(b)** Example of generation of the digit '2' at different bit-widths.

In order to analyze the quality of the fake images, the synthetic 10,000 digits dataset generated by DCGAN was classified by LeNet-5. Figure 4.7(a) illustrates the classes distribution in comparison to that related to the ground-truth dataset. It can be noticed that quantization below 4 bits causes failures to get some classes. In addition, from sample images reported in Figure 4.7(b), it can be observed that 3-bit precision adds noise to images thus making the represented digit no longer understandable.

When the more complex CelebA dataset is referred to, the acceptable quantized bit-width ranges between 8 and 6. The sample synthetic images reported in Figure 4.8 show that the 5-bit quantization causes training failure.

(a) **(b)**

Figure 4.8 Example images from the synthetic CelebA at **(a)** 6 bits and **(b)** 5 bits.

Finally, VI indicates that the segmented masks generated by the U-Net for the Oxford-IIIT Pet and for the Cityscapes dataset achieve acceptable prediction for quantized bit-widths in the ranges 8-2 and 8-3, respectively. Figure 4.9 illustrates examples of segmented masks obtained at 2- and 3-bit quantization.

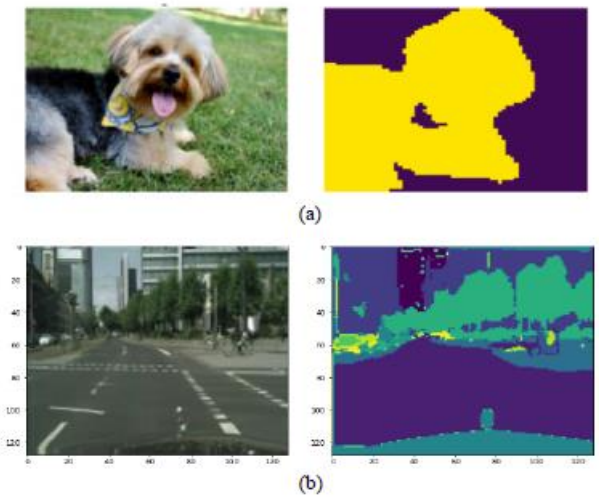


Figure 4.9 Example of predictions using U-Net **(a)** from the Oxford-IIIT Pet dataset at 2 bits, and **(b)** from Cityscapes at 3 bits.

4.1.8. OBJECTIVE ANALYSIS

The output images obtained by the CA were compared to the ground-truths in terms of PSNR for objective analysis: the PSNR values computed at the end of the training steps were averaged over the validate batch.

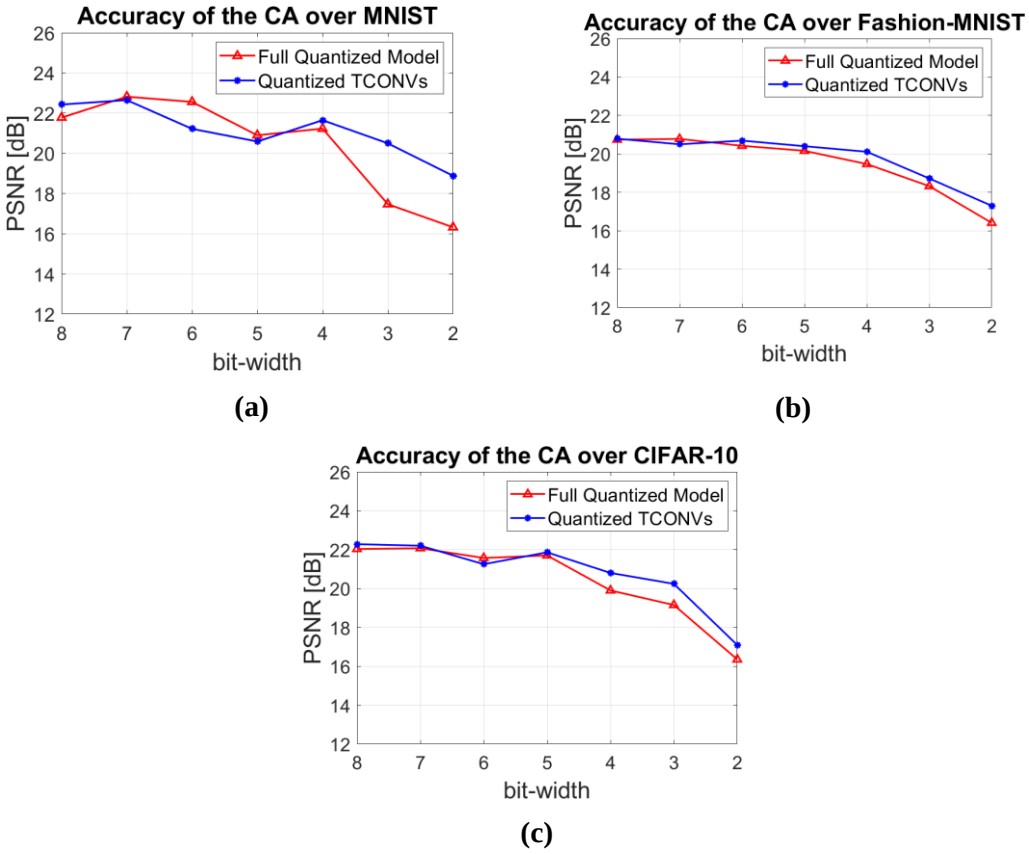


Figure 4.10 Accuracy of the convolution autoencoder vs the bit-width range over **(a)** MNIST, **(b)** Fashion-MNIST and **(c)** CIFAR-10, considering the full quantized model as well as the quantization of TCONV Layers only.

Figure 4.10 shows how the PSNR varies versus the bit-width for the three datasets in two scenarios. In the case of the full quantized model, while the PSNR achieved for the Fashion-MNIST and CIFAR-10 datasets slightly decreases when the quantized bit-width moves from 8 to 5 bits, for the MNIST dataset the PSNR oscillates till the 4-bit quantization. Then, a significant decrease initiates at the 3-bit quantization, with a 17.71% decay with respect to the previous value. Similar is the behavior when only the TCONV Layers are quantized to bit-widths higher than the sweet-spot. At lower bit-widths, quantizing only the decoder improves the accuracy. As an example, considering MNIST,

the 3-bit quantization of the entire CA leads to a PSNR ~14.8% lower than that obtained by quantizing only the TCONV Layers.

The effectiveness of quantization is confirmed by the 32-bit floating point CA. Indeed, the PSNR values of 21.82 dB, 20.83 dB and 21.59 dB obtained for MNIST, Fashion-MNIST and CIFAR-10, respectively, are very close to the highest PSNR exhibited by the quantized model.

The synthetic datasets generated by DCGAN over CIFAR-10 and CelebA were characterized in terms of IS and FID. While the latter was examined for both the considered datasets, the former was evaluated only for CIFAR-10, due to its lack of meaning for CelebA [122]. Figure 4.11 reports the analytical metrics trend considering the experimented bit-widths. Taking into account that, when adopting the 32-bit floating point representation, the DCGAN reaches IS=6.64 and FID=34.41 for CIFAR-10, and FID=66.28 for CelebA, the results plotted show a sweet-spot at 5-bit for CIFAR-10, whereas the accuracy achieved for CelebA is acceptable only at the 8- and 7-bit quantization.

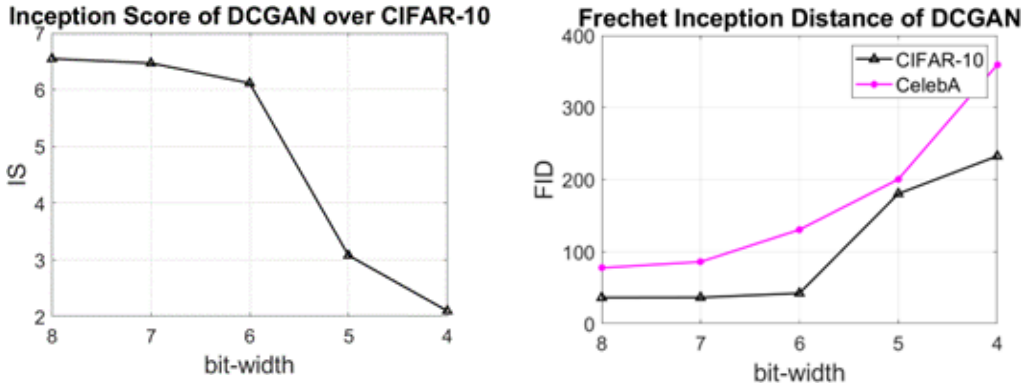


Figure 4.11 The DCGAN accuracy: **(a)** IS of CIFAR-10 and **(b)** FID of both CIFAR-10 and CelebA experiments.

Figure 4.12 shows the accuracy evaluation for the U-Net model. It can be observed that, for bit-widths in the range 8-3, the accuracy reached by the full quantized model is quite similar to that achieved by quantizing only the TCONV Layers. In particular, Figure 4.12(a) shows a sweet-spot at the 2-bit quantization, which causes the mIoU to vary by 1.76%, in the case of the fully quantized model, and by –0.98%, when only TCONV Layers are quantized, with respect to the previous value. Furthermore, as visible in

Figures 4.12(b) and 4.12(c), for the Cityscapes dataset, lower mIoU values are achieved. This is mainly due to the more complex scenes into the images and the higher number of classes to be predicted. Indeed, while the Pet dataset refers to binary segmentation (i.e., foreground/background), experiments using Cityscapes examine 12 classes. Figures 4.12(b) and 4.12(c) also show how the mIoU varies versus the quantized bit-width and considering the training set and the test set, respectively. The full quantized U-Net model reaches the major gap between 3- and 2-bit quantization, with a decay of about 31.91%, for the training set, and 14.37%, for the test set. With the quantization of TCONV Layers only, the gap below the 3-bit quantization becomes less evident: –9.22% and –2.03% for the training and test datasets, respectively.

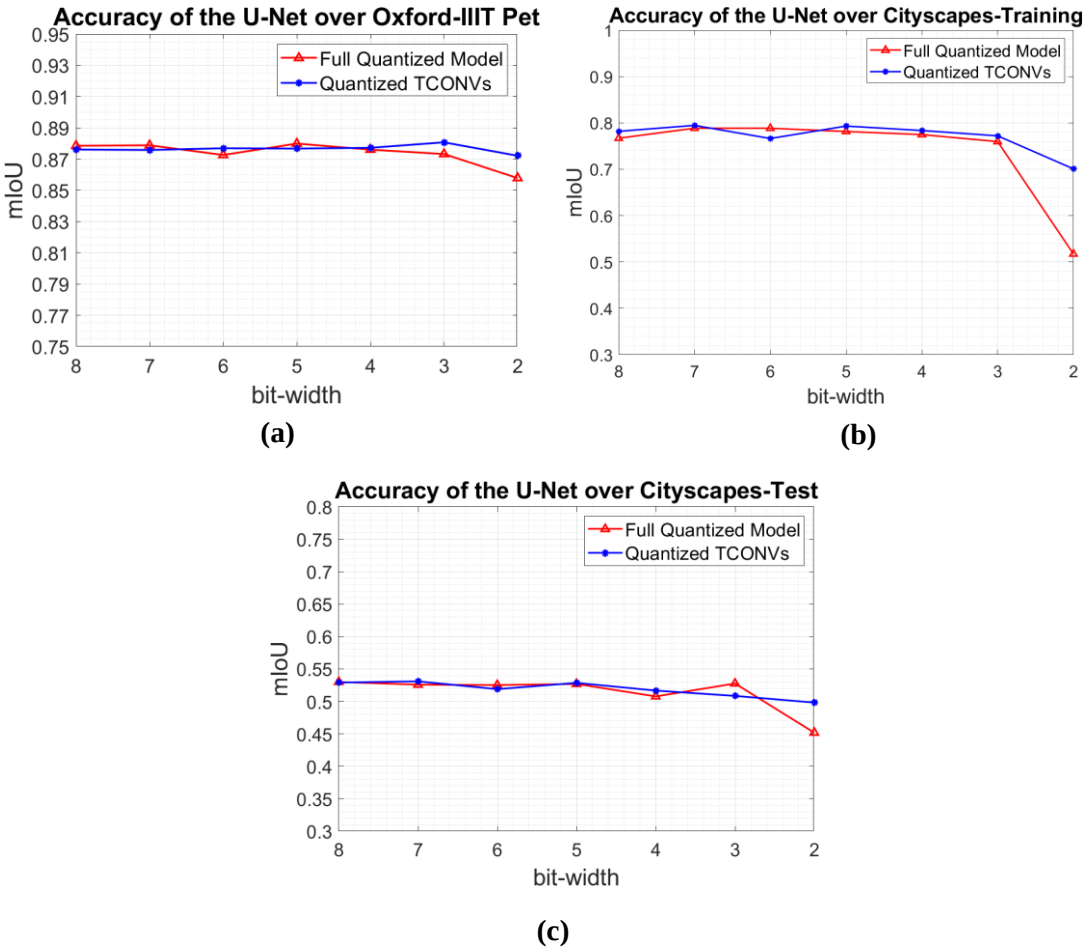


Figure 4.12 Accuracy of the U-Net vs the bit-width range over **(a)** Oxford-IIIT Pet, **(b)** Cityscapes-training set and **(c)** Cityscapes-test set, considering the full quantized model as well as the quantization of TCONV Layers only.

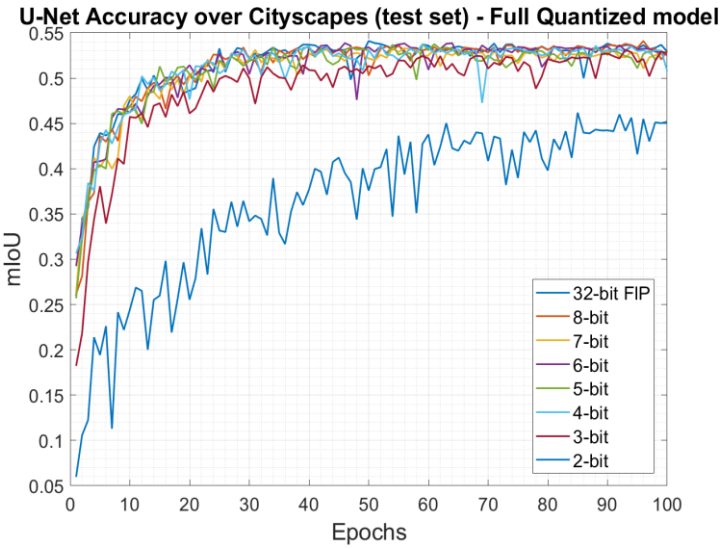


Figure 4.13 The accuracy trend of the full quantized model vs the epochs over Cityscapes (test set) at different bit-widths.

As a further analysis, the accuracy achieved by the full quantized U-Net model was evaluated varying the epochs considered for training. This is useful to understand how many epochs are needed to lead the model to converge. Figure 4.13 illustrates the behavior of the Cityscapes (test set) training. After about 40 epochs, the model reaches the maximum mIoU, which is practically the same for bit-widths in the range 8-4 bits. The 3-bit model requires more training, up to 100 epochs, to approach accuracy values comparable to those reached with higher bit-widths.

Also in this case, the effectiveness of the examined quantization is confirmed by the results obtained with the 32-bit floating point U-Net model: mIoU=0.88 for Oxford-IIIT Pet, mIoU=0.81 for Cityscapes (training set), and mIoU=0.53 for Cityscapes (test set).

4.1.9. OVERALL OVERVIEW OF THE EXPERIMENTS

Table 4.1 provides a final overview of the experiments, by highlighting: the referred models and applications; the quantized layers; the datasets; the accuracy, expressed in terms of both VI and analytical metrics. For what concerns the VI, the numbers indicates the minimum bit-width that guarantees acceptable accuracy. Conversely, the value provided for the analytical metrics is related to the measure at the minimum bit-width that ensures acceptable accuracy.

Table 4.1 Overview of the accuracy evaluation of TCONV-based QNNs.

Model *Application*	Quantized Layers	Dataset	Accuracy	
			Visual Inspection	Analytical metrics
CA *Image compression/ decompression*	All	MNIST	4b	PSNR=21.23 dB (4b)
		Fashion-MNIST	4b	PSNR=20.16 dB (5b)
		CIFAR-10	--	PSNR=21.70 dB (5b)
DCGAN *Image generation*	Generator only	MNIST	5b	--
		CIFAR-10	--	IS=6.12 (6b) FID=42.27 (6b)
		CelebA	6b	FID=85.80 (7b)
U-Net *Image segmentation*	All	Oxford-IIIT Pet	2b	mIoU=0.87 (3b)
		Cityscapes	3b	mIoU=0.76 (3b)

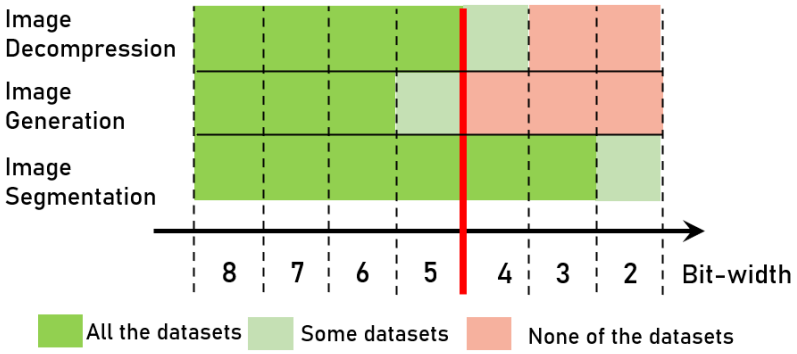


Figure 4.14 Summary diagram: accuracy levels vs bit-width.

Experiments related to the CA showed that quantization down to 5 bits may be performed by satisfying both VI and the PSNR. However, the 4-bit implementation is also successful when VI evaluates MNIST and Fashion-MNIST decompression. DCGAN results are worse. RGB images from CIFAR-10 and CelebA satisfy the range 7-6 bits overall. MNIST generation behaves well even below, down to 5 bit, but considering the VI only. Image segmentation through the U-Net model provides the most meaningful results. Indeed, quantization down to 3 bits satisfy both the Oxford-IIIT Pet and the Cityscapes dataset and considering both the VI and the mIoU metric.

The diagram in Figure 4.14 provides a more intuitive way to interpret all the results. For each experiment (i.e., image decompression, image generation, image segmentation), the green cells indicate that, at the given bit-width, the test satisfy all the datasets considered. Conversely, the light-green cells indicate that some datasets are not satisfied

for the given bit-width. Finally, red cells indicate that the given bit-width does not satisfy any dataset. Overall, it can be concluded that the impact of bit-width over the accuracy of TCONV-based model is successful in the range 8-5 bits.

4.2. C++ DESIGN OF A PLATFORM-INDEPENDENT TRANSPOSED CONVOLUTION LAYER

4.2.1. BACKGROUND

As previously stated, the accuracy evaluation of TCONV-based QNNs, presented in the previous Section, was a feasibility study to move towards the design and evaluation on FPGAs. The successful results, which had highlighted that deep quantization infers good accuracy to the examined models, were the jumping-off point for hardware implementation. Indeed, even though quantization leads to some loss of accuracy, by sacrificing a given number of bits to represent activations and weights, this makes CNNs architectures able to: (1) use limited memory to store the weights; (2) use lower resources and, thus, less power to perform MACs; (3) accordingly, replicate several times each MAC core to improve the throughput.

In addition, considering that TCONV acceleration using FPGAs was examined at relatively high bit-widths, proper effort towards lower precision is mandatory to follow the technology scaling. Among the state-of-the-art solutions, the FlexiGAN architecture [46] performs TCONV Layers based on conventional zero-insertion strategy. The architecture uses 16-bit data to benchmark GAN models and run over a high-end Xilinx Virtex UltraScale device [76]. Acceleration of super-resolution networks was analyzed in [47, 16], by transforming transposed convolutions into conventional convolutions by either filters or *fmaps* re-organization. While the former [47] complies with the 13-bit data format for both weights and pixels, the latter [16] manages 16-bit *fmaps* and 10-bit filters. High-performance parts, including the Kintex-7 [88] and Zynq UltraScale [60] were used to reach satisfactory performances.

The IOM strategy was examined in [48, 66] to perform image generation and segmentation. While the former [48] exploits the 12-bit data format on the low-end Zynq-7020 [58], the latter [66] adopts the Zynq-7045 [58] at the 32-bit precision. The lowest

bit-width is reached by Uni-OPU [50] that exploits the high-end Zynq-7100 [58] to benchmark models at 8-bit.

Unlike the referred works, which either conceived just a single accelerator to execute multiple TCONV Layers or used high-precision data to meet acceptable accuracies, the architecture proposed in the following Section refers to a circuit that can be integrated into dataflow hardware models, consisting of stacked accelerators of TCONV layers, where quantization is tunable even below 8 bits.

4.2.2. THE PROPOSED DESIGN

This Section presents the design of the novel TCONV Layer, starting from a C++ description. This is the starting point for a lower-level implementation, by using the HLS. Indeed, the C++ model is translated into the RTL abstraction, before being synthesized and implemented on FPGAs. The use of HLS allows the TCONV Layer to be configured at design time, by selecting both functional and non-functional parameters, including the kernel size K, the up-sampling factor S, the image sizes, the parallelism and the bit-width of both weights and activations. As a result, it can be used to emulate the functionality of different TCONV Layers and using both low-end and high-end FPGAs, according to the resources constraints, as well as the expected performances and energy efficiency.

The proposed TCONV Layer is modelled using the high-level C++ method *TranspConvLayer*. Algorithm 4.1 shows the pseudo-code, while Figure 4.15 illustrates the top-level scheme, to clarify the way in which the inputs are supplied. Indeed, the *TranspConvLayer* function is equivalent to an architecture able to process T_{IC} *ifmaps* in parallel. A further parameter, T_{OC}, manages the number of *ofmaps* generated in parallel. Weights and biases are defined as static arrays to be stored on-chip offline. Given O_C as the number of *ofmap* channels and established that the equivalent hardware can generate T_{OC} *ofmaps* at a time, O_C/T_{OC} steps (line 1) are needed to complete the processing of the current layer. Within each step, the I_C *ifmaps* are subjected to as many TCONVs to generate a group of T_{OC} *ofmaps*. I_C/T_{IC}+1 sub-steps (line 2) are used to generate the each group of *ofmaps*.

During the former I_C/T_{IC} sub-steps (*if* condition of line 3), after having loaded from on-chip memories the current weights (*curr_w*) and biases (*curr_b*) (line 4), each sub-step executes the IOM algorithm [49]. This is accomplished by the *TranspConvIOM*

software procedure (line 5), which reads one input activation (*in_act*) and generates $S \times S$ output activations (*out_act*) per cycle, other than being supplied by the current weights stored on-chip. During the extra '+1' step (from *else* condition in line 6), biases accumulations, quantization and output data movement are performed.

Algorithm 4.1: The pseudo-code of the *TranspConvLayer* function

Inputs: Stream of T_{IC} *ifmaps*,
$O_C \times I_C \times K \times K$ weights,
O_C biases
Output: Stream of T_{OC} *ofmaps*

```
1:  for oc=0 to O_C/T_OC–1 do
2:    for ic=0 to I_C/T_IC do
3:      if ic < I_C/T_IC then
4:        Load weights curr_w and biases curr_b;
5:        TranspConvIOM(curr_w, in_act, out_act);
6:      else
7:        for iter=0 to H_O*W_O–1 do
8:          #pragma HLS PIPELINE II=1
9:          Read the provisional T_OC activations and add biases;
10:         Quantize and move the final T_OC activations out as stream;
11:       end for
12:     end if
13:   end for
14: end for
```

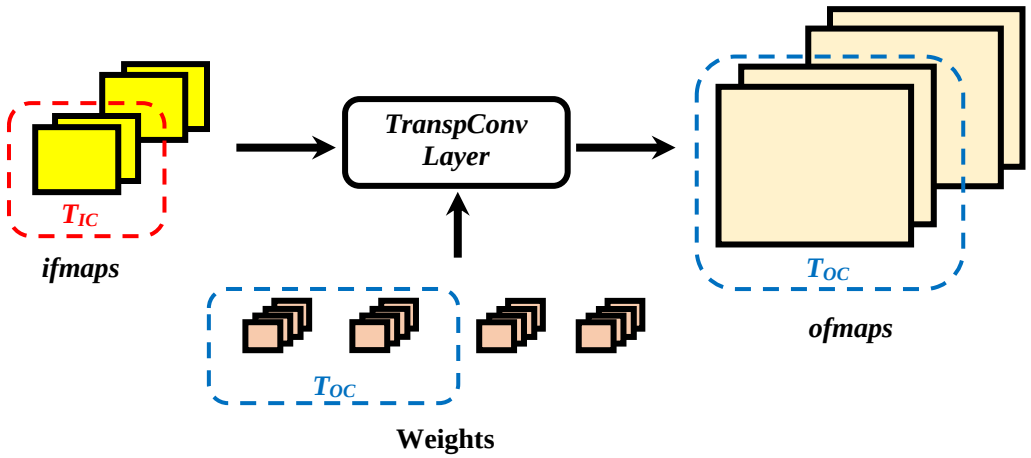


Figure 4.15 The inputs and outputs of the *TranspConvLayer*, when T_{IC}=2, T_{OC}=2.

Algorithm 4.2: The pseudo-code of the *TranspConvIOM*

Inputs: Stream of T_{IC} *ifmaps'* activations *in_act*,
$T_{OC}\times T_{IC}\times K\times K$ weights *curr_w*,
Output: Stream of $T_{OC}\times S\times S$ *ofmaps*' activations *out_act*

1: **for** *hi*=0 to H_I–1 **do**
2: **for** *wi*=0 to W_I–1 **do**
3: *#pragma HLS PIPELINE II*=1
4: inAct = in_act.read();
5: **for** *toc*=0 to T_{OC}–1 **do**
6: **for** *tic*=0 to T_{IC}–1 **do**
7: **for** *kr*=0 to *K*–1 **do**
8: **for** *kc*=0 to *K*–1 **do**
9: **if** *kc* < *K*–*S* **then**
10: **if** *wi*=0 **then**
11: outCol(*toc*,*tic*,*kr*,*kc*) = inAct(*tic***N*+*N*–1:*tic***N*)*curr_w(*toc*,*tic*,*kr*,*kc*);
12: **else**
13: outCol(*toc*,*tic*,*kr*,*kc*) = inAct(*tic***N*+*N*–1:*tic***N*)*curr_w(*toc*,*tic*,*kr*,*kc*)+ColBuff(*toc*,*tic*,*kr*,*kc*);
14: **end if**
15: **else**
16: outCol(*toc*,*tic*,*kr*,*kc*) = inAct(*tic***N*+*N*–1:*tic***N*)*curr_w(*toc*,*tic*,*kr*,*kc*);
17: **end if**
18: **if** *kc* ≥ *S* **then**
19: ColBuff(*toc*,*tic*,*kr*,*kc*–*S*) = outCol(*toc*,*tic*,*kr*,*kc*);
20: **end if**
21: **if** *kr* < *K*–*S* **then**
22: **if** *kc* < *S* **then**
23: **if** *hi*=0 **then**
24: outRow(*toc*,*tic*,*kr*,*kc*) = outCol(*toc*,*tic*,*kr*,*kc*);
25: **else**
26: outRow(*toc*,*tic*,*kr*,*kc*) = outCol(*toc*,*tic*,*kr*,*kc*)+RowBuff(*toc*,*tic*,*wi*,*kr*,*kc*);
27: **end if**
28: **end if**
29: **else**
30: outRow(*toc*,*tic*,*kr*,*kc*) = outCol(*toc*,*tic*,*kr*,*kc*);
31: **end if**
32: **if** *kr* ≥ *S* **then**
33: **if** *kc* < *S* **then**
34: RowBuff(*toc*,*tic*,*wi*,*kr*–*S*,*kc*) = outRow(*toc*,*tic*,*kr*,*kc*);
35: **end if**
36: **end if**
37: **if** *tic*=0 **then**
38: outAcc(*toc*,*kr*,*kc*) = outRow(*toc*,*tic*,*kr*,*kc*);
39: **else**
40: outAcc(*toc*,*kr*,*kc*) += outRow(*toc*,*tic*,*kr*,*kc*);
41: **end if**
42: **end for** (lines 8-6)
43: Accumulate results of steps *ic* and *ic*–1 and store on-chip as *out_act*.
44: **end for** (lines 5, 2-1)

Algorithm 4.2 details the pseudo-code of the *TranspConvIOM* method. In order to equip the model with pipelining and parallelism capabilities, the *#pragma HLS PIPELINE* is exploited (line 3). The latter ensures a concurrent execution of the underlying instructions, by allowing reading a new valid input each *II* clock cycles, with *II* being the Initiation Interval. The code refers to *II*=1, meaning that the equivalent circuit will be able to receive new inputs per each new clock cycle. Following the previous remark, the T_{IC} input activations are multiplied by the respective weights, thus producing windows of provisional outputs. For each cycle, the first *K*–*S* columns of the current $T_{OC}\times T_{IC}$ windows must be accumulated to the last *K*–*S* columns of the windows produced in the previous cycle. This column overlapping is managed within lines 9-20. There, the *outCol* 4D array collects the actual multiply-accumulations, while *ColBuff* is responsible to temporarily store the pixels that must be accumulated in the following clock cycle to comply with the referred overlap. Similarly, the row overlap is handled by means of the *outRow* and the *RowBuff* multi-dimensional arrays (lines 21-36). It is worth underlining that, with respect to *ColBuff*, *RowBuff* is characterized by an additional dimension. Indeed, it has to store all the provisional results, related to the current input image row, which must be accumulated to the results later furnished by the next image row.

Afterwards, taking into account that multiple *ifmaps* are processed in parallel, the T_{IC} results are accumulated in a pixel-wise manner (lines 37-41). Finally, the results provided during the current *ic* sub-step are accumulated to those generated during the step *ic*–1 (line 43).

For each group of T_{IC} input activations, $T_{OC}\times S\times S$ outputs are provided at the same time, as a consequence of the IOM algorithm which manages outputs as windows instead of single pixels. The latter behavior ensures the improvement of the output throughput by a factor S^2, however it unbalances the ratio between the input and the output throughputs of the proposed TCONV Layer. This is the reason behind the +1 step reported from line 6 of Algorithm 4.1. During this piece of time, the outputs are read from the output buffer *outBuff* as a stream of T_{OC} pixels at a time, instead of $T_{OC}\times S\times S$. Biases accumulation is also performed during this extra-step.

The referred *outBuff* is a 3D array: the first dimension indicates the number T_{OC} of *ofmaps* generated in parallel; the second dimension refers to the up-sampling factor *S*; the third dimension represents the area of the generic *ifmap* and equal to $H_I\times W_I$. In other

words, we can suppose this array as T_{OC} banks of S memories, each being $H_I \times W_I$ wide. For each cycle, the generic bank is responsible to accommodate the $S \times S$ results, provided by the IOM TCONV, into the S memories. This is accomplished by placing rows of S pixels into each memory cell. At the completion of the generation of the parallel *ofmaps*, a proper control logic is used to read the stored pixels as T_{OC} outputs per cycle, using the raster-order policy to provide them to either an external memory resource or a subsequent computing layer. This is realized by managing proper memory pointers, namely *buff_idx* and *cell_idx* that indicates, respectively, which memory of the bank and which specific cell are under analysis.

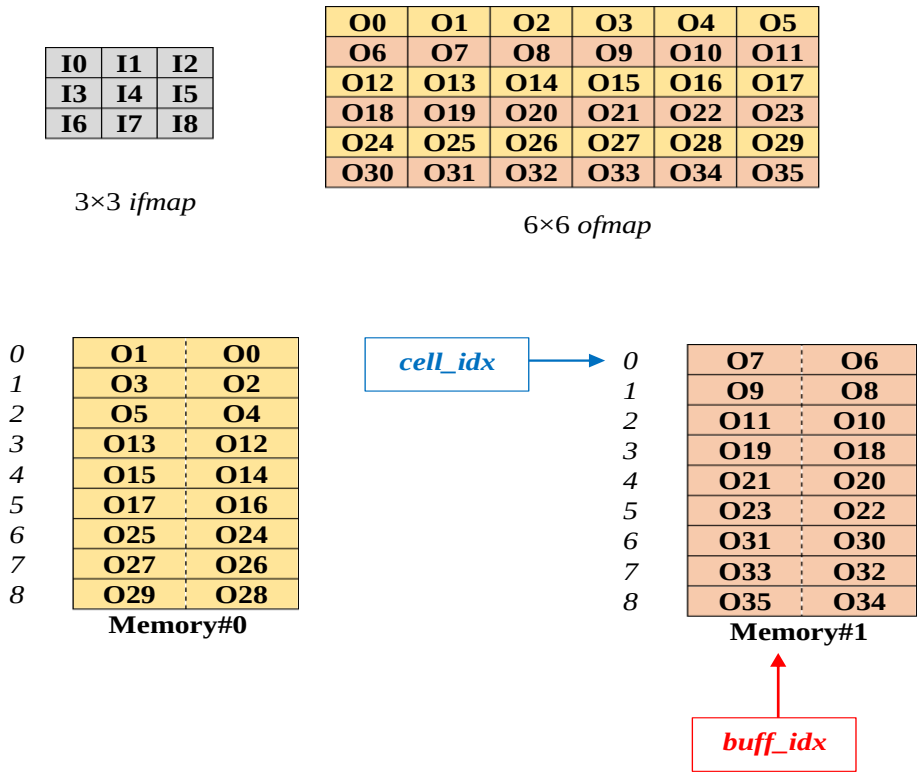


Figure 4.16 Example of *outBuff* to store a 6×6 *ofmap* generated from a 3×3 *ifmap*.

To better understand the behavior of the *outBuff*, we refer to the example in Figure 4.16, where a 3×3 *ifmap* is used to generate a 6×6 *ofmap* (at S=2). We suppose that T_{OC}=1. Accordingly, one bank of S=2 memories, each being 3×3-wide, is exploited. The generic memory cell hosts S=2 adjacent output pixels. The numbers indicated in each cell refer to the spatial position of those pixels within the generic *ofmap*. The control logic follows that numbering strategy to output T_{OC}=1 data per clock cycle. During the first clock cycle

of the reading step, *buff_idx*=0 and *cell_idx*=0. This state is preserved for two cycles, in order to get the first two pixels stored within the *Memory#0*. During the third and fourth cycles, *cell_idx*=1 and the pixels *O2,O3* are read. The fifth and the sixth cycles are interested by the reading of pixels *O4,O5* by pointing *cell_idx*=2. Afterwards, the *cell_idx* value is wound back by W_I–1=2 positions, *buff_idx*=1, and the pixels *O6-O11* are read following the previous increment strategy. Thus, the *Memory#0* leads the process again considering the subsequent cells (*cell_idx* = 3 to 5), and so on for the remaining values, in a ping-pong manner.

4.2.3. PARAMETRIC ANALYSIS

The C++ TCONV Layer was conceived as a parametric template able to adapt itself, at design time, to different configurations. These are managed by the filer size *K*, the up-sampling factor *S*, the image sizes, as well as the bit-width *N* and the parallelism through the parameters T_{IC} and T_{OC}.

This layer was translated into the RTL abstraction, using Vivado HLS v2019.2, where the design was also synthesized and implemented in hardware using the XC7Z020 part [58] at the fixed clock frequency *f*=100 MHz.

In order to evaluate the impact of the specific TCONV parameters, Table 4.2 illustrates their impact over both the resource occupation and the speed throughput, expressed as the number of frames-per-second (*fps*). Furthermore, Figure 4.17 shows the referred trends through plots: (1) bar-charts for the resource utilization, expressed as percentages related to the XC7Z020 part; (2) lines for the throughput (*fps*).

The impact of the up-sampling factor *S* was evaluated considering the kernel size *K*=9, used for super-resolution imaging tasks [31]. Considering that the sizes of *ofmaps* directly depend on the referred variable (i.e., the higher *S*, the wider the *ofmap* area), it can be noticed that the number of BRAMs grows by a factor 3.6 when moving from *S*=2 to *S*=4. As expected, the higher the *ofmaps* sizes, the lower the throughput, which exhibits a decrease of 3.2 times throughout the whole examined range.

Table 4.2 TCONV Layer parametric analysis varying *K*, *S*, and the image sizes.

Fixed Parameters	Variable Parameter	Resources				Throughput [fps]
		LUTs	FFs	BRAMs (18K)	DSPs	
N=8, H_I=W_I=32, K=9,T_{IC}=2,T_{OC}=2	S=2	13091	7163	9	220	18348
	S=3	13234	7124	19	220	9433
	S=4	13106	6929	33	220	5649
N=8, H_I=W_I=32, S=2,T_{IC}=2,T_{OC}=2	K=3	598	408	8	36	19342
	K=4	912	649	8	64	19267
	K=5	1245	836	8	100	19120
	K=7	1821	1111	9	196	18761
	K=9	13091	7163	9	220	18348
N=8,K=4,S=2, T_{IC}=2,T_{OC}=2	H_I=W_I=8	940	617	4	64	255102
	H_I=W_I=16	899	633	4	64	74074
	H_I=W_I=32	912	649	8	64	19267
	H_I=W_I=64	756	395	48	64	4854
	H_I=W_I=128	922	421	128	64	1219

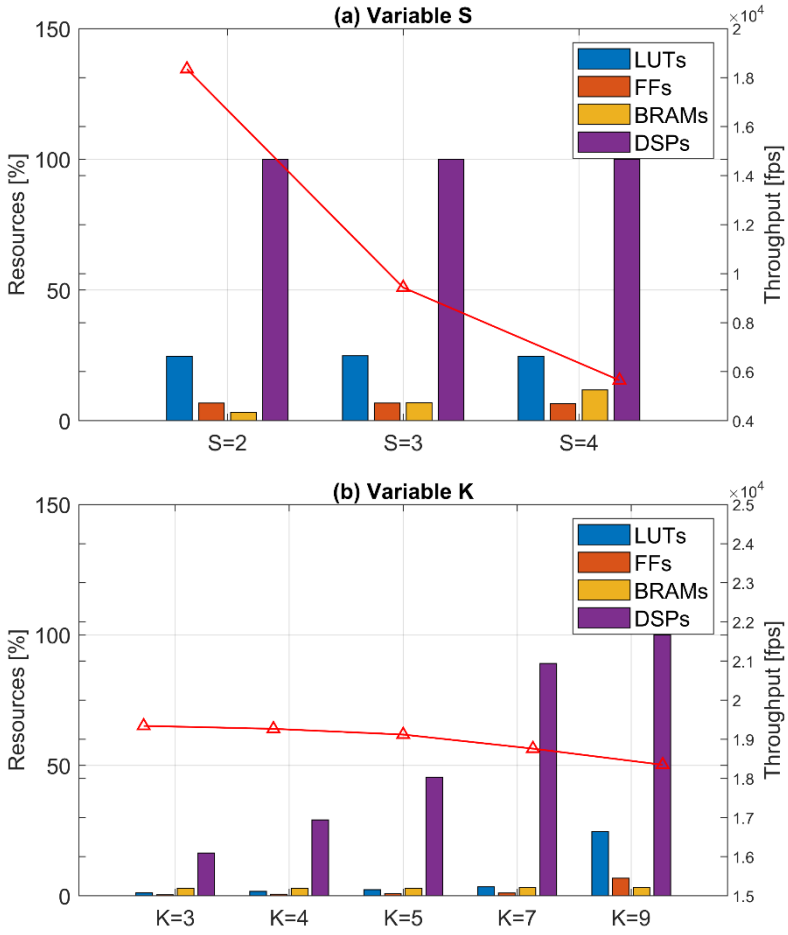


Figure 4.17 Parametric analysis of the TCONV layer **(a)** varying *S*, **(b)** varying *K*.

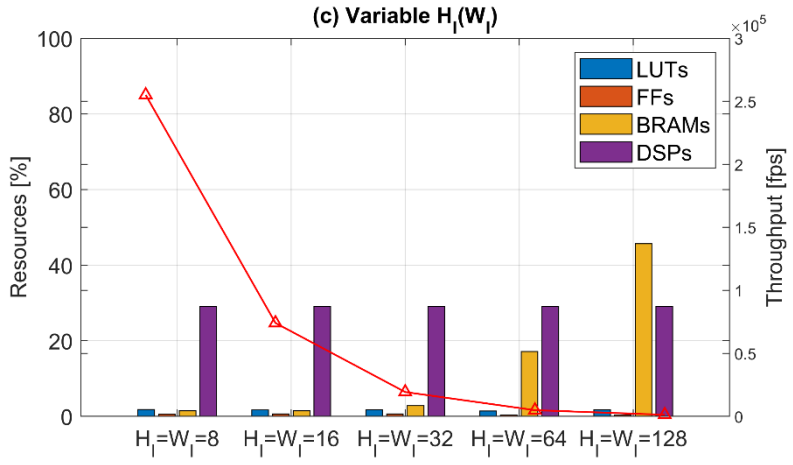


Figure 4.17 (*cont.*) Parametric analysis of the TCONV layer **(c)** varying H_I (W_I).

The computing resources (i.e., LUTs and DSPs) are strongly influenced by the kernel size *K*. This is due to the use of *#pragma HLS PIPELINE* that completely unrolls the loops related to the multiplications between the given input pixels and the respective filters to meet the *II*=1 requirement. Accordingly, the higher the filter size, the higher the number of replicas to parallelize the computations. The throughput is weakly affected by *K*. Indeed, only the fast filters' loading step constitutes an extra contribution to the overall latency. Specifically, when moving from $K = 3$ to $K = 9$, the *fps* metric shows a negligible ~5.1% loss.

Conversely, the throughput exhibits a relevant dependence on the image sizes $H_I \times W_I$. This is due to the use of even-more wider *ifmaps* and, in turn, to the generation of *ofmaps*. whose area grows with the product between H_I,W_I and *S*. For what concerns the on-chip memories, their usage dominates the scenario for H_I,$W_I \geq 64$. Below, the synthesizer alternates the use of the latter with the employment of logic to infer memory for both overlapping and temporary output storage.

4.2.4. STATE-OF-THE-ART COMPARISONS

For purposes of comparisons with state-of-the-art FPGA-based TCONV accelerators, the proposed C++ template was used to implement proper engines following the configuration parameters of the competitors [123, 14, 50]. Furthermore, given the platform-independency provided by the HLS, the architecture was implemented considering both the low-end XC7Z020 part and the high-end XC7Z100 FPGA [58].

Table 4.3 illustrates the characterization in terms of: (a) the specific configuration (i.e., bit-width, kernel size *K* and stride *S*), (b) resource occupation, (c) achieved clock frequency, (d) throughput expressed as the number of Giga Outputs per Second (Gout/s), (e) power dissipation in Watt, and (f) the energy efficiency as the ratio between the Gout/s and the power. The main percentage changes are reported in Figure 4.18.

Table 4.3 Characterization of the HLS TCONV Layer and state-of-the-art comparisons.

	New	New	New	[123]	[14]	[14]	[50]
Device	XC7Z020	XC7Z020	XC7Z100	XC7Z020	XC7Z020	XC7Z100	XC7Z100
Bit-width	16	16	16	16	16	16	16
K,S	3,2	5,2	5,2	3,2	5,2	5,2	3,2
LUTs	2.52k	2.99k	11.71k	3.82k	2.90k	15.50k	115.2k
FFs	0.82k	1.41k	10.04k	5.09k	4.30k	22.90k	241.4k
BRAMs [Mb]	1.69	1.40	4.52	0.45	0.84	0.84	17.38
DSPs	144	200	800	29	210	1120	1987
Freq. [MHz]	125	125	200	125	200	300	200
Gout/s	3.90	1.95	6.25	1.95	1.56	9.37	12.5
Power [W]	0.29	0.36	1.53	0.09	0.42	2.62	2.89
Gout/s/W	13.45	**5.42**	**4.08**	**20.97**	3.71	3.58	4.33

Best performance in **bold** at the parity of device and *K*, *S*.

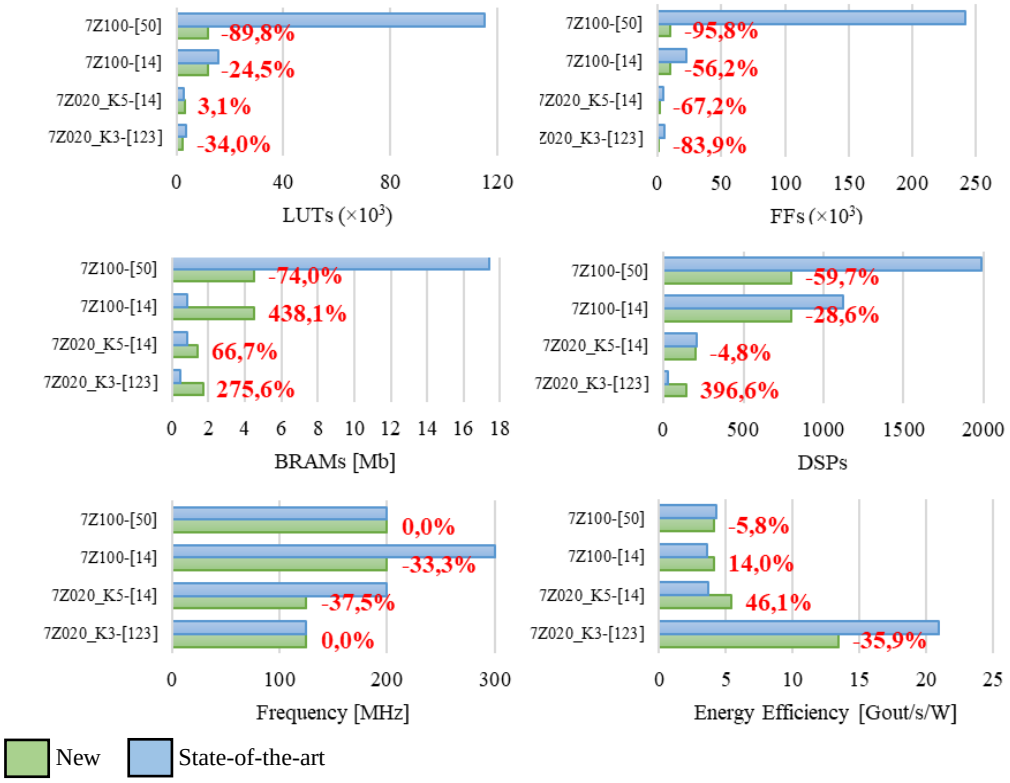


Figure 4.18 Percentage change comparisons: resources, frequency, energy efficiency.

The circuit having K=3 was implemented within the XC7Z020 FPGA at the 16-bit precision to be compared with the direct counterpart [123]. The proposed solution doubles the speed throughput of [123], thanks to the parallelism capability supported. However, the overall energy efficiency is ~36% lower due to the higher power dissipation. This is mainly due to the fact that the latter adopts an approximation strategy that minimizes the number of TCONVs, by replacing some of them with simple averages. Accordingly, the generic model accelerated may suffer from a detrimental impact on accuracy.

The same FPGA was used to test the configuration with K=5 for fair comparison with [14]. It is worth underlining that both the accelerators adopt the IOM strategy for TCONV computations. However, the latter [14] is designed by using VHDL templates to maximize the performance. Despite this, the novel engine exhibits a ~25% higher throughput at a ~37.5% lower frequency.

The implementation within the XC7Z100 part shows competitiveness with the design [14, 50]. The proposed solution is only ~5.8% less energy-efficient, but using ~89.8%, ~95.8%, ~59.7%, ~74% less LUTs, FFs, DSPs and BRAMs, respectively. When compared with [14], our circuit shows an efficiency ~12.3% higher but suffering from a 1.5 times lower performance, due to the lower frequency.

Overall, the proposed design has proven to be competitive with respect to state-of-the-art, thus suggesting that a careful high-level design by a proper use of *#pragmas* may lead to efficient hardware implementation.

4.3. TOWARDS A DATAFLOW ARCHITECTURE OF STACKED TRANSPOSED CONVOLUTION LAYERS: THE CASE-STUDY OF A DECODER FOR IMAGE DECOMPRESSION

Considering that low bit-widths result in lower area occupation to manage a single TCONV Layer, a dataflow architecture consisting of stacked layers can be used to avoid frequent off-chip communications to move data, thus positively affecting both speed and power. Stimulated by the impressive results in low-precision Fully-Connected NNs and conventional CNNs [18, 100], we examined the hardware implementation of a simple two-layered decoder [21], based on the accuracy trend of our previous study [19]. To emulate a real-life scenario, we accommodated it into a complete embedded system. A

detailed design-space exploration of both resources and latency under the bit-width and the parallelism constraints is presented, as well as the suitability of a careful HLS design by means of a proper usage of *#pragmas*.

4.3.1. THE PROPOSED DESIGN

The *TCONV-based Decoder* (TCD) accelerator complies with the *TranspConvDecoder* method, reported in Algorithm 4.3, which calls the instances of *TranspConvLayer* and *TranspConvLayerReLU*. The former is nothing but the high-level model introduced in the previous section, while the latter is also equipped with ReLU non-linearity [25].

Algorithm 4.3: The pseudo-code of the *TranspConvDecoder* function

Input: *inStream* of T_{IC} *ifmaps*
Output: *outStream* of T_{OC} *ofmaps*
1: *#pragma HLS INTERFACE axis port=inStream*
2: *#pragma HLS INTERFACE axis port=outStream*
3: *#pragma HLS DATAFLOW*
4: TranspConvLayerReLU (*inStream*, *L1Stream*);
5: TranspConvLayer (*L1Stream*, *outStream*);

The specific composition of the neural network under examination is as follows: (1) the first layer processes 4 *ifmaps* using 16 filters, each consisting of 4×3×3 weights, thus producing 16 *ofmaps*; (2) the second layer processes the previous 16 *fmaps* using 16×3×3 weights to produce the final *ofmap*. For each layer, the up-sampling factor is S=2, thus the decoder generates a final 32×32 *ofmap* starting from 8×8 *ifmaps*.

The behavior of the equivalent hardware architecture is reported in Figure 4.15, using a timing diagram. For each layer, hereby labelled as *L1* and *L2*, each yellow box indicates one of the I_C/T_{IC} steps to compute T_{OC} provisional *ofmaps*. The generic orange box refers to the extra step to move the final T_{OC} *ofmaps* to the subsequent layer for further computations, and following the reading policy as depicted in Figure 4.19. Thanks to the use of proper *#pragmas*, both function- and task-level parallelism were enabled to limit the overall latency. The *#pragma HLS DATAFLOW* permits the circuit to overlap the execution of consecutive TCONV Layers, while *#pragma HLS PIPELINE* inherits the *#pragma HLS UNROLL* to perform the related computations in parallel, thus allowing consecutive reading of new data per cycle. Indeed, as reported in the bottom sketch of Figure 4.15, where the second yellow box of *L2* is magnified, all the steps associated to

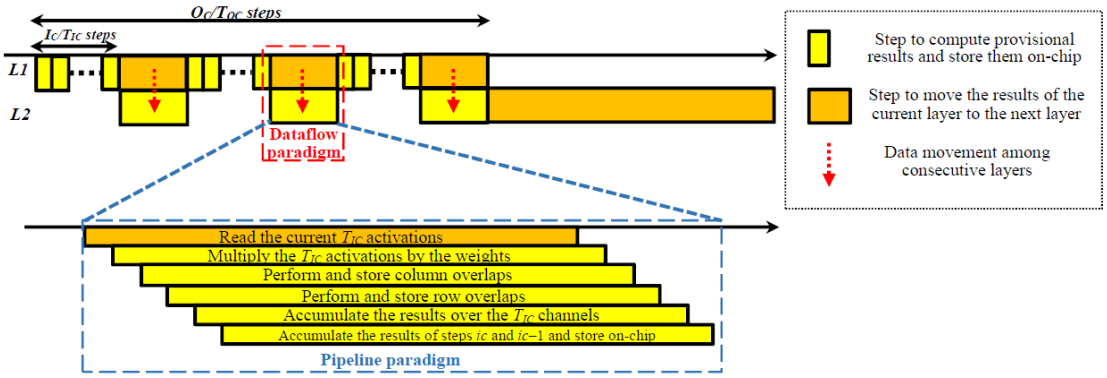


Figure 4.19 Timing diagram describing the behavior of the proposed TCONV-based decoder.

the IOM are pipelined, meaning that the *i-th*+1 step can begin as soon as the first valid data from the *i*-th step is available.

In order to be cooperate effectively with other accelerators, The TCD unit is interfaced using the AXI4-Stream specification [59], which enables high-speed data streaming. The *#pragma HLS INTERFACE* axis allows the custom accelerator to be provided with that infrastructure, as reported in Algorithm 4.3, where both inputs and outputs are equipped with this interface.

4.3.2. ANALYSIS OF THE EFFECTIVENESS OF *#PRAGMAS*

In order to evaluate the effectiveness of the use of *#pragmas*, an extensive design-space exploration was carried out. Resource occupation and latency of different configurations were analyzed, by varying the bit-width and the parallelism parameters. For each specific configuration, both the *baseline design* and the *optimized design* were implemented. The former makes use of a limited number of high-level directives, needed to ensure the correct synthesis and behavior of the model. The latter add other *#pragmas* in order to boost the performance, thus heavily exploiting parallelism. All the used *#pragmas* are summarized in Table 4.4, with an explanation of their meaning and the way they are used for the specific architecture.

Table 4.4 Detail of the used #pragmas for the TCONV-based decoder.

#pragma	**Baseline**	**Optimized**
HLS ARRAY_PARTITION It partitions a multi-dimensional array into multiple sub-arrays.	Used for the output buffer only, to allow the circuit to proper store the T_{OC}×S×S pixels provided each cycle.	
	Not Used	Used for filters and biases to be accessed simultaneously by as many computing elements. Used for row overlap buffers according to the Input-Oriented Mapping Algorithm.
HLS RESOURCE It specifies the type of resource to be used to implement a given variable.	**Not Used**	Used to implement the output buffer as a simple dual-port memory.
HLS PIPELINE It provides pipelining capabilities to the referred function or loop. Thus, new inputs can be processed every *II* clock cycles, with *II* being the initiation interval. It inherits the *#pragma HLS UNROLL* (see below).	**Not Used**	Used to read each new cycle (*II*=1): • Weights and biases; • *ifmaps*. Used to manage the timing of data export from the output buffers.
HLS UNROLL It transforms loops by creating several copies of the body to infer parallelism.	**Not Used**	Used to unroll loops for data acquisition, computations and buffering. It is directly provided by the *#pragma HLS PIPELINE*.
HLS INTERFACE It specifies which interfaces must be used by I/O ports.	Used to equip both input and output ports of the hardware engine with the AXI4-Stream Interface.	
HLS DATAFLOW It specifies the task-level parallelism to improve the concurrency of C++ functions.	Used to manage the relationship between the two layers of the decoder. It is needed to ensure the correct functionality of data transfer using streams when functions are cascaded, as reported in the HLS guide. [128].	

The bit-width range was spanned between 8 and 4 bits, considering that this is the overall acceptability range from our previous study [19]. Specifically, the decoder trained over MNIST [105] well perform down to 4 bits, while its effectiveness is limited to at most 5 bits for Fashion-MNIST [106]. Also in this case, the XC7Z020 [58] was considered as the target platform. Considering the resource availability, the parallelism configurations (T_{IC},T_{OC}) = (1,1), (2,2), (2,4) were examined.

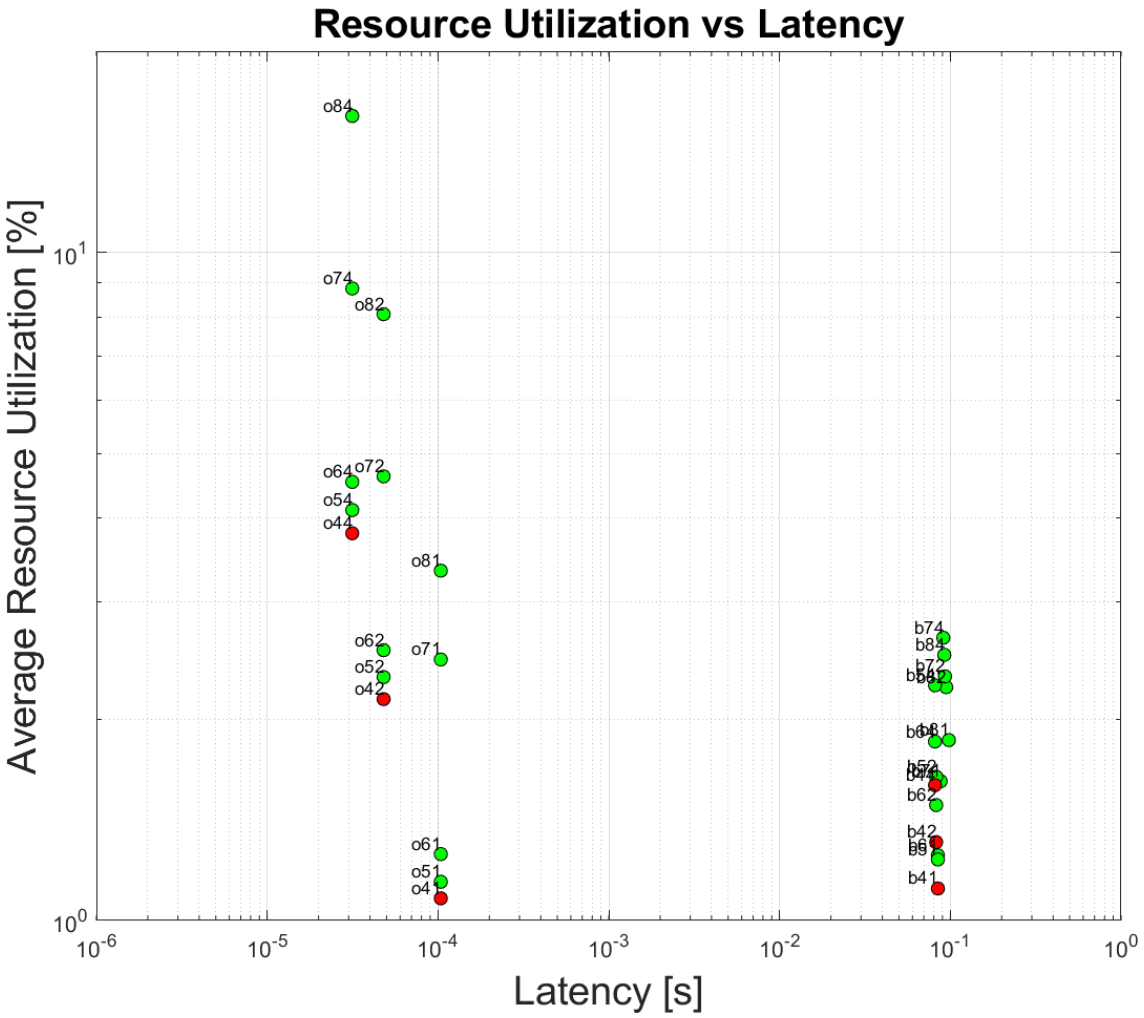


Figure 4.20 Trade-off analysis related to the HLS TCONV-based decoder.

Figure 4.20 illustrates the trade-off trend between resource occupation and latency. The former is expressed as the average percentage over the available resources (i.e., LUTs, FFs, BRAMs and DSPs). For visualization purposes, both the axes are expressed in base-10 logarithmic scale. Each point represents a specific configuration. There, each label indicates the design type (baseline as '*b*', optimized as '*o*'), the bit-width *N* and the T_{OC} factor. For instance, *o62* refers to the optimized design with bit-width N=6 and output parallelism factor T_{OC}=2. Green points refers to the bit-width configurations that satisfy the accuracy analysis for both MNIST and Fashion-MNIST datasets, while red points fail in meeting Fashion-MNIST quality requirements.

The key message is that two main regions can be identified within the plot: on the right, the baseline designs are placed, whereas on the left the optimized counterparts can be found. By the viewpoint of latency, proper usage of *#pragmas* to infer parallelism

results in a considerable improvement, reaching up to 3 orders of magnitude. Obviously, lower latency means higher resources requirements. However, considering the worst case configuration *o84*, the average utilization is limited to 16%.

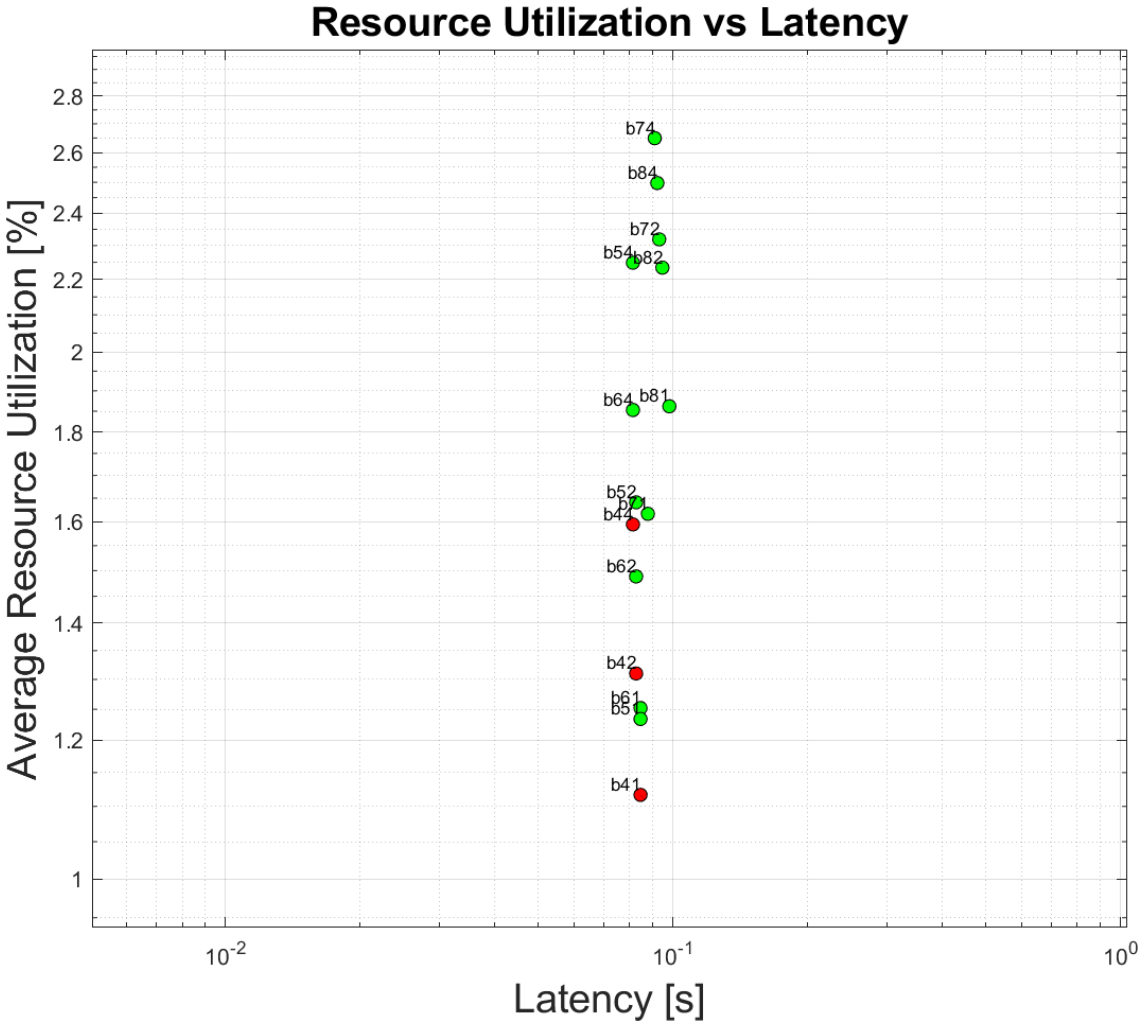


Figure 4.21 Detail of trade-off analysis of the baseline TCONV-based decoder.

Figure 4.21 helps the reader to better interpret the results related to the baseline designs. While the latency variation among the different configuration is very limited, the occupation delta among the different implementations is more noticeable. Considering the bit-width, it can be observed that higher values of N mean higher occupation due to the wider multipliers/adders to be used. In addition, higher T_{OC} reflects in higher occupation, as expected. Finally, it is worth observing that the configuration *b41* requires the lowest amount of resources, while the configurations *b64*, *b54*, *b44* employ the lowest amount of clock cycles to complete the computations.

4.3.3. CHARACTERIZATION OF THE EMBEDDED SYSTEM

Considering the remarkable behavior of the optimized version of the TCD accelerator, its behavior was tested within a real-life scenario, by embedding it into the system depicted in Figure 4.22. Here, The *Programmable Logic* section accommodates the custom TCD unit and the Direct Memory Access (DMA) unit. The latter acts as a bridge between the TCD accelerator and the external memory to move data bursts. A general-purpose processor is the main unit of the *Processing System* section whose task is to configure the DMA to fulfill its functions. Furthermore, a Memory Controller manages the communication between the DMA and the external memory. In order to exchange both informative content and configurations, the embedded system here presented is equipped with the fourth generation of the Advanced eXtensible Interface (AXI4) [59]. The TCD unit is interfaced to the DMA through the AXI4-Stream specification. The DMA, other than accommodating the same streaming interface to communicate with the TCD, also presents the AXI4-Full specification to manage memory-mapped transactions of data bursts. This is needed to enable the correct exchange of data with the external memory, through the Memory Controller. The DMA receives configuration signals by the general-purpose processor using the AXI4-Lite Interface, which handles single memory-mapped transactions.

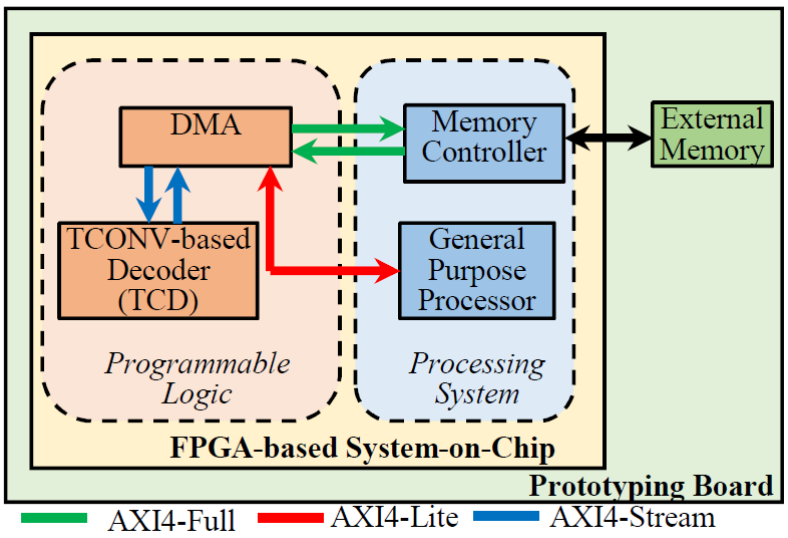


Figure 4.22 The embedded system accommodating the TCONV-based decoder.

The TCD accelerator was designed using the Xilinx Vivado HLS tool (v2019.2), while the whole embedded system was built through the Xilinx Vivado tool. The circuit

was tested using the Digilent Pynq-Z1 board [124]. It accommodates the Xilinx XC7Z020 SoC [58], which consists of both a 650MHz dual-core Cortex-A9 Processor and an Artix-7 FPGA section. The referred board is also equipped with a 512 MB DDR3 memory. The clock frequency of the *Programmable Logic* was set to f=100 MHz.

The design-space exploration carried out considered the bit-width range analyzed in the previous sub-section (i.e., 8-4 bits), while the parallelism was spanned between the rolled configuration (T_{IC},T_{OC}) = (1,1) and the configuration (T_{IC},T_{OC}) = (4,4). The latter has been shown as the most critical scenario for the DSPs usage, reaching the ~87% of their utilization.

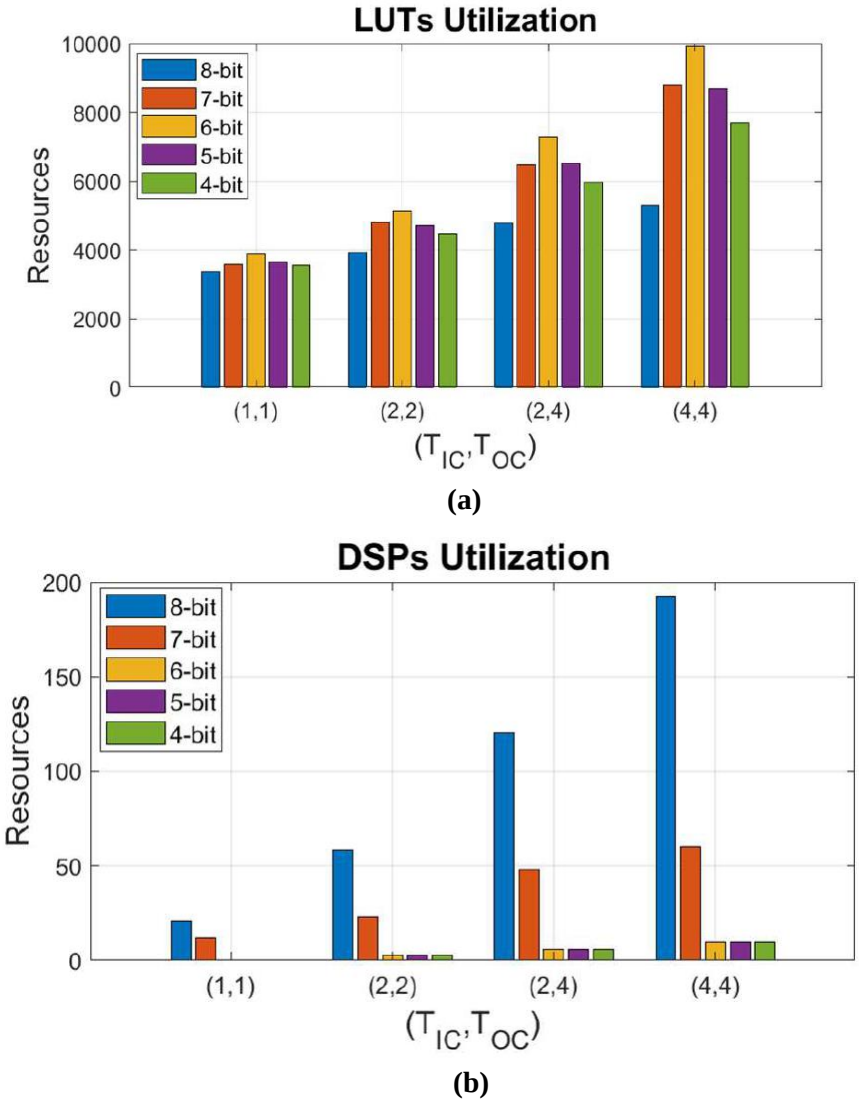
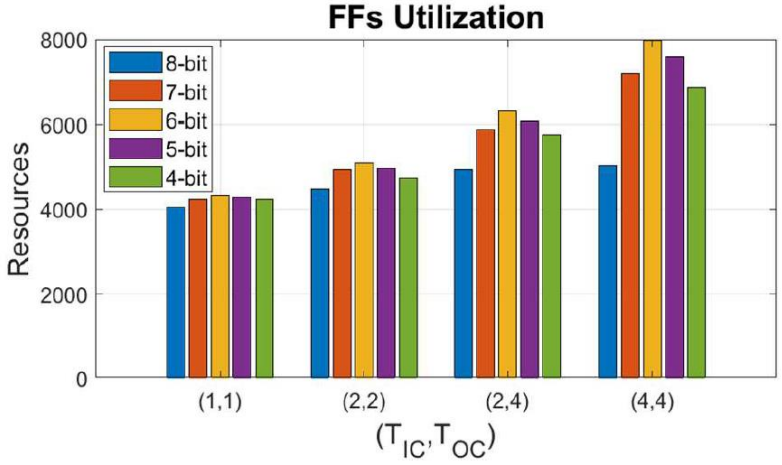
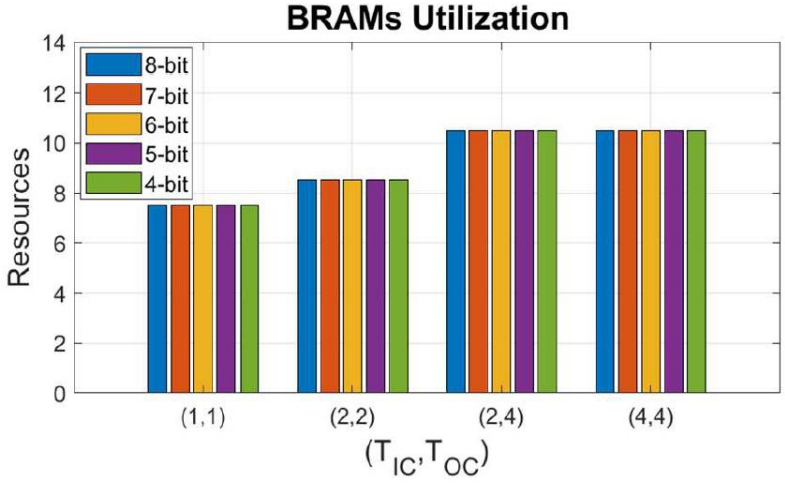


Figure 4.23 Resource utilization trends varying the bit-width N and the parallelism factors T_{IC} and T_{OC}: **(a)** LUTs; **(b)** DSPs.



(c)



(d)

Figure 4.23 (*cont.*) Resource utilization trends varying the bit-width N and the parallelism factors T_{IC} and T_{OC}: **(c)** FFs; **(d)** BRAMs.

Figure 4.23 illustrates the trend of resources varying the mentioned parameters. For each bar graph, 4 groups are represented, each referring to a specific (T_{IC},T_{OC}) configuration. Each group consists of 5 adjacent bars related to the bit-width range. In order to discuss the impact of the bit-width N, let us consider the example scenario (T_{IC},T_{OC}) = (2,4). It is expected that fewer resources should correspond to lower values of N. This because as the bit-width is reduced, also the data-width decreases. However, LUTs occupation follows the opposite trend in the range 8-6 bits. Logic occupies ~35% more area moving from 8 to 7 bits, and a further ~12% increase towards the 6-bit configuration. This counterintuitive behavior is justified by examining the DSPs trend. In fact, when N is reduced from 8 to 6 bits, the DSPs usage shows a huge 95% decrease. This means that, by lowering N, the synthesizer assigns multiplications and additions to

LUTs instead of DSPs. When the 6-4 bits range is considered, the theoretical trend is met, in that the DSPs occupation is kept constant. Thus, the lower *N*, the lower the logic area. FFs confirm the LUTs trend, in that they are mainly involved in pipelining contiguous combinatorial paths.

BRAMs occupation does not vary with *N*. This behavior can be explained by considering the C++ coding style. In general, memories are managed through arrays. The *#pragma HLS ARRAY_PARTITION* directive helps the synthesizer to proper translate such high-level structures into memory elements, by partitioning the given array into sub-arrays corresponding to as many hardware resources. The referred partition took into account the T_{OC} parameter and the up-sampling factor *S*, without constraining over *N*.

When the bars are examined as groups, proper considerations about the parallelism influence can be extrapolated. All the resources show a growth as the (T_{IC},T_{OC}) couples assume higher values, in that the generic computing unit is replicated as many times. BRAMs trend confirms the previous considerations. Since their usage is related to the output parallelism, higher T_{OC} means higher memory occupation. This observation is further confirmed when the configuration (2,4) and (4,4) are compared. In this scenario, since the input parallelism parameter only varies, the memory area is the same.

Finally, the contribution of the TCD accelerator only was evaluated. While DSPs do not present any difference when compared to the embedded system, in that they are completely used within the custom unit, it is worth reporting the percentage fraction of the remaining resources. Indeed, LUTs, FFs and BRAMs occupy the ~48%, ~34% and ~45% of the embedded system, respectively, by averaging all the implemented configurations.

To calculate the latency, the Integrated Logic Analyzer (ILA) IP [125] was accommodated within the referred embedded system to examine the actual waveforms. Two types of analyses were carried out: (1) the latency of both the TCD unit and the DMA; (2) the impact of the TCD unit only. The former was accomplished by triggering the control signals of the AXI4-Full Interface of the DMA. Conversely, the latter considered the AXI4-Stream signals of the custom accelerator.

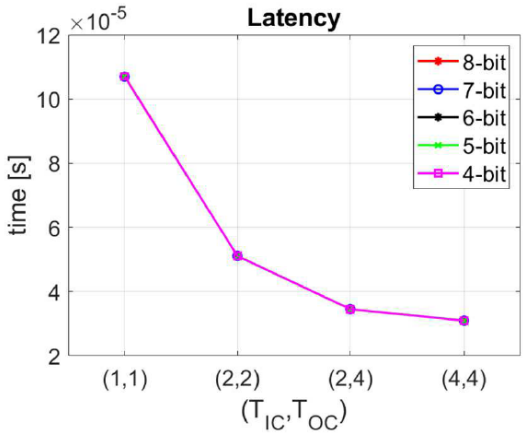


Figure 4.24 Latency trend varying the bit-width N and the parallelism factor T_{IC} and T_{OC}.

Figure 4.24 depicts the contribution of both the TCD unit and the DMA, when the bit-width and the parallelism are varied in the defined range. N does not influence the execution time because the architecture takes the same time to perform transposed convolutions at different bit-widths. The (T_{IC},T_{OC}) couples affect the latency. When the parallelism is not inferred, the circuit takes ~107 µs to complete its tasks. Increasing the parallelism up to the (4,4) configuration leads the circuit to be 3.5× faster. Taking into account the scenario N=4, Figure 4.25 compares the latency of the TCD only and the extra contribution required by the DMA. For all the parallelism configurations, the delta is only ~3 µs, thus occupying the ~7% of the overall latency on average. This means that the DMA transactions infer a negligible extra contribution, thus confirming the suitability of this family of embedded systems to accelerate tasks like those related to CNNs.

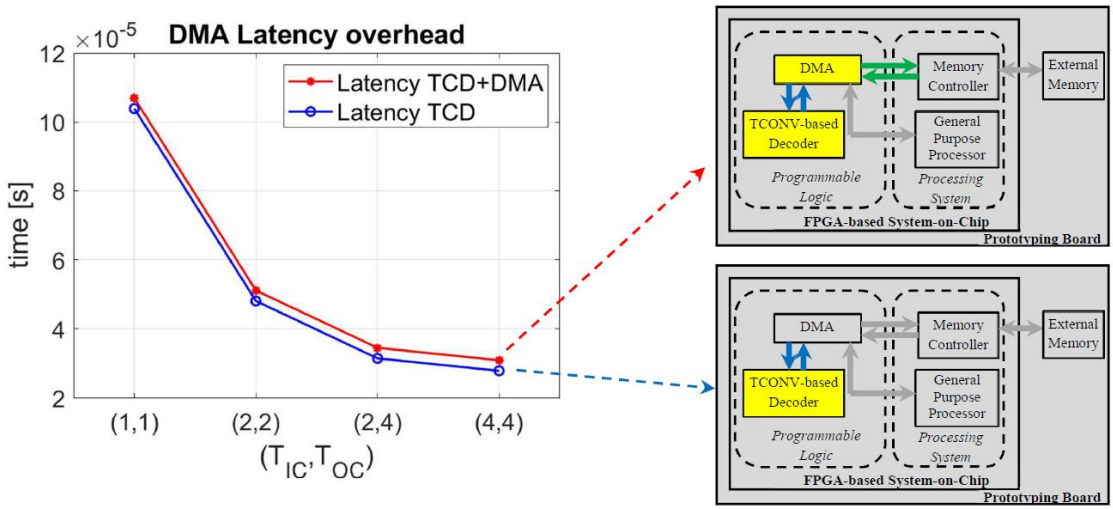


Figure 4.25 Latency variation between the embedded system and the contribution of the TCONV-based decoder only.

4.4. DATAFLOW ACCELERATION OF GENERATIVE ADVERSARIAL NETWORKS

In order to further investigate the behavior of dataflow architectures accommodating TCONV Layers, the generator of the DCGAN network [22] was taken into account. Specifically, considering the accuracy analysis conducted in our previous study [19], we implemented in hardware the architecture for MNIST image generation at 5-bit precision. Algorithm 4.4 illustrates the top-level pseudocode.

Algorithm 4.4: The pseudo-code of the *DCGAN* function

Input: latent vector *inStream*
Output: generated image *outStream*

```
1:  #pragma HLS INTERFACE axis port=inStream
2:  #pragma HLS INTERFACE axis port=outStream
3:  #pragma HLS DATAFLOW
4:  ProjectReshape (inStream, L1Stream);
5:  DataBuffer (L1Stream, L1Stream_buff);
6:  TranspConvLayerReLU (L1Stream_buff, L2Stream);
7:  DataBuffer (L2Stream, L2Stream_buff);
8:  TranspConvLayerReLU (L2Stream_buff, L3Stream);
9:  TranspConvLayer (L3Stream, outStream);
```

The DCGAN function receives, as input, a 1×1×100 latent vector to be modelled as an output image by the whole network. The learned filters are completely stored on-chip, thus avoiding frequent loading from off-chip resources. The *ProjectReshape* method is the first layer that processes the given inputs: it consists of a TCONV Layer at S=1, which manages 256 filters of 4×4×100 weights. The second and the third layers are TCONV Layers equipped with ReLU non-linearity [25], which use 128 and 64 filters, respectively, with K=4 and S=2. Finally, the last TCONV Layer provides the resulting 32×32 output image by processing the input *fmaps* with one 64×4×4 filter.

With respect to the decoder analyzed in the previous Section, the DCGAN requires internal *fmaps'* reuse to comply with layers that use more than one 3D filter. Indeed, due to the streaming behavior of the architecture, pixels are generated once and consumed by the next computing resource as soon as possible. As a consequence, a buffer at the interface between two layers, of which the latter adopts more than one 3D filter, is mandatory. This is the meaning of the *DataBuffer* function reported in Algorithm 4.4 and placed between the layers 1-2 (line 5) and 2-3 (line 7). No extra buffering is needed

between layers 3-4, in that the latter uses just one 3D filter to provide the output image, thus reading the incoming values, from the third layer, once.

4.4.1. CHARACTERIZATION

By using the HLS tool, the C++ parametric template was translated into the RTL abstraction and implemented using the XC7Z045 part for purposes of characterization and comparisons with previous works. Table 4.5 illustrates the results in terms of (a) bit-width; (b) resource occupation in terms of LUTs, FFs, BRAMs and DSPs; (c) achieved clock frequency and throughput as the number of frames-per-second (*fps*); (d) power consumption in Watt; (e) energy-efficiency as the ratio between the throughput and the power.

Three different parallelism configurations were analyzed, to examine the impact of the latter parameter on the accuracy. Specifically, the T_{OC} array reported in Table 4.5 indicates the output parallelism configurations of each layer (equivalently, the input parallelism configurations of the next layer).

Table 4.5 Characterization of the HLS DCGAN model.

T_{OC} **configuration**	**(1,1,1,1)**	**(2,2,2,1)**	**(2,4,2,1)**
Device	XC7Z045	XC7Z045	XC7Z045
Bit-width	5	5	5
LUTs	8.62k	16.88k	22.40k
FFs	10.96k	21.77k	25.08k
BRAMs [Mb]	6.68	6.89	7.03
DSPs	0	1	1
Freq. [MHz]	167	167	167
fps	52	196	240
Power [W]	0.34	0.59	0.74
Energy Efficiency [*fps*/W]	152.94	332.20	324.32

As expected, the higher the parallelism, the better the speed performances. When the baseline configuration (i.e., without parallelism) is compared to the most parallelized counterpart, a speed improvement of 4.6 times is experienced. The slightly higher power dissipation allows the optimized circuit to double the energy-efficiency.

Also in this case, the *#pragma HLS PIPELINE* used to both infer pipelining and parallelism permits an overall better performance at the expense of higher area

occupation. However, while the logic increases up to more than 22k LUTs, the on-chip memory usage is practically steady around 7 Mb. This is due to the fact that BRAMs are heavily exploited to store the needed filters, while the extra amount for provisional data buffering is negligible. In addition, considering the limited bit-width through which both activations and weights are represented, it is worth observing that computations are performed using LUTs, without the requirement of DSPs.

Finally, the configuration T_{OC} = (2,4,2,1) was compared to state-of-the-art counterparts, as reported in Table 4.6. Figure 4.26 also reports the percentage changes with respect to the state-of-the-art. When compared to the architecture presented in [126], the proposed accelerator shows a remarkable 7.3× improvement in terms of energy efficiency, even at 1.7× lower throughput. The power saving is motivated by the fact that the novel engine accommodates all the needed TCONV Layers on chip, as well as the required filter weights, thus limiting the off-chip memory accesses. In addition, the data treatment at 5-bit shrinks the area occupation for computations significantly; indeed, the 16-bit counterpart hugely adopts DSPs, while the referred design practically nullifies their usage. It is also worth underlining that the achieved *fps* of [126] are further improved by the 300 MHz clock frequency, thanks to the high-performance Alveo U200 device [127] took under consideration.

The architecture presented in [70], even implemented using the same FPGA, is 22.9% slower while running at the same frequency. The higher usage of resources to comply with the transformation steps of Winograd-based TCONVs makes the engine 10 times less energy-efficient with respect to the proposed 5-bit competitor.

The high-end Virtex-7 690 T FPGA accommodates the DCGAN accelerator [68] at the 16-bit precision. The latter clearly exhibits the best throughput among the counterparts but at expense of a considerable amount of resources, by requiring 13.26×, 23.47×, 3.56× more LUTs, FFs, on-chip BRAMs, respectively. Furthermore, to meet the 1.21 ms latency, the 64% of the available DSPs is required.

Table 4.6 State-of-the-art comparisons of FPGA-based DCGAN model.

	New	[126]	[70]	[68]
Device	XC7Z045	XCU200	XC7Z045	XC7VX690T
Bit-width	5	16	16	16
LUTs	22.40k	483k	196.7k	297.12k
FFs	25.08k	726k	-	588.62k
BRAMs [Mb]	7.03	77	10.9	25.03
DSPs	1	2176	603	2304
Freq. [MHz]	167	300	167	200
fps	240	400	185	826
Power [W]	0.74	9	5.8	-
***fps*/W**	**324.32**	44.44	31.89	-

Best energy efficiency in **bold**.

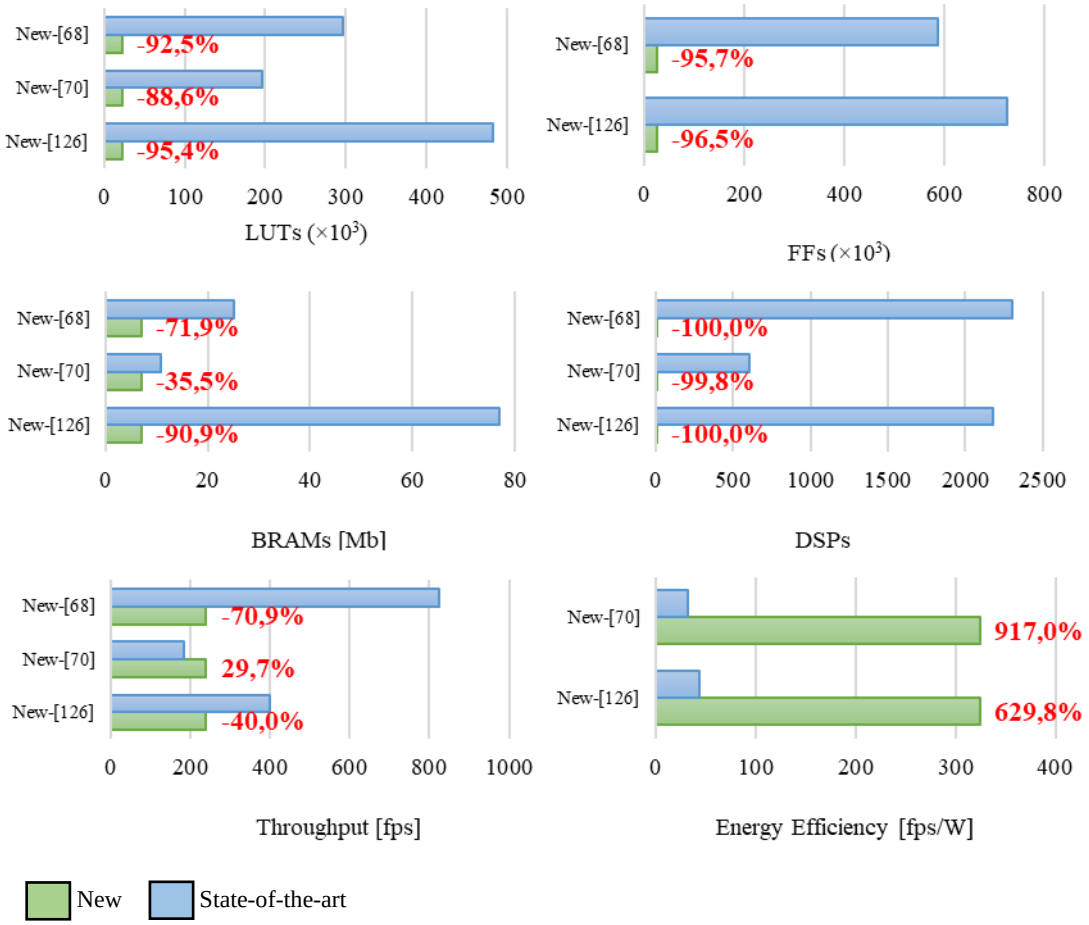


Figure 4.26 Percentage change comparisons: resources, throughput, energy efficiency.

4.5. SUMMARY

In this Chapter, the impact of quantization over QNNs using TCONV Layer was presented. First, a high-level analysis of PyTorch-based models (dealing with image decompression, synthetic objects generation and semantic segmentation) was carried out, showing the range 8-5 bits integer ensures appreciable quality results with respect to 32-bit floating point counterparts. The investigation took under consideration the visual inspection as well as proper analytical metrics for objective evaluation.

Stimulated by the above results, a TCONV based engine using C++ was presented and hardware implemented in FPGA using the High-Level Synthesis. The parametric and platform-independent design allowed a design-space exploration varying the filter size, the up-sampling factor and the size of images, showing limited resource occupation at thousands of frames-per-second. For example, when images are up-sampled with S=4, the equivalent circuit requires the 24.6%, 6.5%, 11.8% of LUTs, FFs, BRAMs, respectively, by processing about 5.6K frames per second.

The proposed architecture was accommodated into dataflow architectures (enabled by C++ pragmas) for image decompression and image generation. The former experiment highlighted that proper usage of HLS *#pragmas* may ensure low latency with relatively limited area occupation. The latter implemented the DCGAN model and demonstrated remarkable speed at the highest energy-efficiency with respect to state-of-the-art counterparts.

Overall, careful HLS design could ensure satisfactory trade-off in terms of area, speed and power consumption.

5. CONCLUSIONS

5.1. SUMMARY AND CONTRIBUTIONS

The pervasiveness of Deep Learning (DL) is a fact in many domains [1, 2, 3, 4]. Image Processing particularly benefits from CNNs that mimic the human visual cortex. However, ever-more increasing effort is needed to offer hardware architectures able to follow the accuracy trend exhibited by complex CNNs, which demand billions of MACs. As highlighted in Chapter 2, among several solutions, FPGAs are showing as one of the most promising candidates to accommodate high-performance CNN models that excel in energy-efficiency [12].

In addition, FPGAs are characterized by the following properties:

- Flexibility, offered by a sea of reconfigurable cells, and able to equip convolutions with run-time reconfigurability of both their own parameters (e.g., filter size), as well as the management of image sizes.
- High-performance Multiply-Accumulations, carried out by dedicated DSPs that are interconnected each other either by high-speed routing interconnections or by fabric tracks to reach the other types of resources.
- On-chip buffering, to limit power-hungry external accesses to off-chip memories.
- Embedded integration, to cooperate with general-purpose processors, by meeting the Hardware-Software Co-Design requirements useful to split complex algorithms by parallel running multiple sections.

These features allow effective implementation of CNNs by either (a) deploying accelerators to be run-time adapted to different configurations, or (b) fitting a complete

model on-chip, based on dataflow processing, and being preliminary subjected to compression to meet the resources constraints [100].

This PhD research has investigated the suitability of particular convolution-based algorithms, namely DCONVs and TCONVs, to be effectively designed on FPGAs. Both low-level languages (i.e., the VHDL) and the use of HLS were adopted to showcase their effectiveness to hardware implement CNNs.

In Chapter 3, the design of VHDL-based circuits [13, 14, 15, 16] has provided meaningful results in terms of resources occupation and achieved clock frequency. The 320 MHz clock frequency was achieved by the hardware implementation of a TCONV Layer, based on the Input-Oriented Mapping strategy, within the XC7VX690T FPGA, thus meeting a throughput close to 1 TOPS, and using only the 5.4% of logic and less than the 50% of the available DSPs resources. As a further example, high energy-efficient super-resolution imaging is carried out by the adaptive CONV/TCONV architecture, reaching up to 518.5 GOPS/W and outperforming the art by up to 2.3 times.

Considering the suitability of HLS to deal with compressed neural networks, Chapter 4 presented the hardware implementation of case-studies dealing with deep quantization [19, 20, 21] and referring to image decompression and image generation. The former has shown that a careful use of *#pragmas* directives may allow the implementation of low-latency circuits, by outperforming baseline configurations by up to 3 orders of magnitude. Energy-efficient image generation is the second key message of the referred chapter, in that the equivalent DCGAN accelerator has shown improvement of at least 7.3 times with respect to state-of-the-art, also motivated by the 5-bit quantization that has allowed the whole parameters to be stored on-chip, without requiring frequent accesses from/to an external memory support.

5.2. FUTURE WORK

The aim of hardware design is not only to provide small, fast and cheap circuits to the community, but also to conceive simple frameworks to infer a generic algorithm with the minimum effort, thus making them suitable to users that may not have specific hardware skills. To this aim, as a future work, we would like to include the HLS of the proposed TCONV Layer within the Xilinx FINN Framework [18]. This full stack project

helps deep learning designers to build a generic FPGA-based accelerator for DNNs' inference by starting from a high-level PyTorch-based model. Accordingly, with no concerns about low-level aspects, an optimized architecture could be available in a limited piece of time, ensuring high-performances at low-power.

LIST OF PUBLICATIONS

- **C. Sestito**, F. Spagnolo, P. Corsonello and S. Perri, "An Efficient Convolution Engine based on the À-trous Spatial Pyramid Pooling," *Proceedings of the 2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors* (ASAP), Virtual, July 2020.

- S. Perri, **C. Sestito**, F. Spagnolo and P. Corsonello, "Efficient Deconvolution Architecture for Heterogeneous Systems-on-Chip", *Journal of Imaging*, vol. 6, no. 85, pp. 1-17, 2020.

- **C. Sestito**, F. Spagnolo, P. Corsonello, S. Perri, "Run-Time Adaptive Hardware Accelerator for Convolutional Neural Networks", *Proceedings of the 16th Conference on PhD Research in Microelectronics and Electronics*, Virtual, 19-22 July 2021.

- **C. Sestito**, F. Spagnolo, S. Perri, "Design of Flexible Hardware Accelerators for Image Convolutions and Transposed Convolutions", *Journal of Imaging*, Vol. 7, no. 210, pp. 1-16, 2021.

- **C. Sestito**, S. Perri, R. Stewart, "Accuracy Evaluation of Transposed Convolution-Based Quantized Neural Networks", *Proceedings of the 2022 IEEE International Joint Conference on Neural Networks (IJCNN)*, Padua, Italy, July 2022.

- **C. Sestito**, S. Perri, R. Stewart, "Design-Space Exploration of Quantized Transposed Convolutional Neural Networks for FPGA-based Systems-on-Chip", *Proceedings of the 20th IEEE International Conference on Pervasive Intelligence and Computing (PiCom 2022)*, Calabria, Italy, September 2022.

- **C. Sestito**, R. Stewart, S. Perri, "High-Level Synthesis of Hardware Accelerators for Deconvolution Engines", *Proceedings of the 15th International Conference on Advances in Circuits, Electronics and Micro-electronics (CENICS 2022)*, Lisbon, Portugal, October 2022.

REFERENCES

[1] A. Voulodimos, N. Doulamis, A. Doulamis and E. Protopapadakis, "Deep Learning for Computer Vision: A Brief Review," *Hindawi Computational Intelligence and Neuroscience*, vol. 2018, pp. 1-13, 2018.

[2] A. B. Nassif, I. Shahin, I. Attili, M. Azzeh and K. Shaalan, "Speech Recognition Using Deep Neural Networks: A Systematic Review," *IEEE Access*, vol. 7, pp. 19143-19165, 2019.

[3] N. Justesen, P. Bontrager, J. Togelius and S. Risi, "Deep Learning for Video Game Playing," *IEEE Transactions on Games*, vol. 12, no. 1, pp. 1-20, 2020.

[4] Z. Wang and A. Majewicz Fey, "Deep learning with convolutional neural network for objective skill evaluation in robot-assisted surgery," *International Journal of Computer Assisted Radiology and Surgery*, vol. 13, pp. 1959-1970, 2018.

[5] Y. LeCun, Y. Bengio and G. Hinton, "Deep Learning," *Nature*, vol. 521, pp. 436-444, 2015.

[6] L. Jiao and J. Zhao, "A Survey on the New Generation of Deep Learning in Image Processing," *IEEE Access*, vol. 7, pp. 172231-172263, 2019.

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *The 3rd International Conference on Learning Representations (ICLR)*, San Diego, CA, USA, 2015.

[8] X. Wang, Y. Han, V. C. M. Leung, D. Niyato, X. Yan and X. Chen, "Convergence of Edge Computing and Deep Learning: A Comprehensive Survey," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 2, pp. 869-904, 2020.

[9] D. Ghimire, D. Kil and S.-h. Kim, "A Survey on Efficient Convolutional Neural Networks and Hardware Acceleration," *Electronics*, vol. 11, no. 945, pp. 1-23, 2022.

[10] Intel Corporation, "FPGA vs. GPU for Deep Learning," [Online]. Available: https://www.intel.it/content/www/it/it/artificial-intelligence/programmable/fpga-gpu.html. [Accessed 3 August 2022].

[11] S. Ma, X. Zhang, C. Jia, Z. Zhao, S. Wang and S. Wang, "Image and Video Compression With Neural Networks: A Review," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 30, no. 6, pp. 1683-1698, 2020.

[12] S. Mittal, "A survey of FPGA-based accelerators for convolutional neural networks," *Neural Computing and Applications*, vol. 32, pp. 1109-1139, 2020.

[13] C. Sestito, F. Spagnolo, P. Corsonello and S. Perri, "An Efficient Convolution Engine based on the À-trous Spatial Pyramid Pooling," in *IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Virtual, 2020.

[14] S. Perri, C. Sestito, F. Spagnolo and P. Corsonello, "Efficient Deconvolution Architecture for Heterogeneous Systems-on-Chip," *Journal of Imaging*, vol. 6, no. 85, pp. 1-17, 2020.

[15] C. Sestito, F. Spagnolo, P. Corsonello and S. Perri, "Run-Time Adaptive Hardware Accelerator for Convolutional Neural Networks," in *IEEE 16th International Conference on PhD Research in Microelectronics and Electronics (PRIME)*, Virtual, 2021.

[16] C. Sestito, F. Spagnolo and S. Perri, "Design of Flexible Hardware Accelerators for Image Convolutions and Transposed Convolutions," *Journal of Imaging*, vol. 7, no. 210, pp. 1-16, 2021.

[17] S. Lahti, P. Sjövall, J. Vanne and T. D. Hämäläinen, "Are We There Yet? A Study on the State of High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898-911, 2019.

[18] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser and K. Vissers, "FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 11, no. 3, pp. 1-23, 2018.

[19] C. Sestito, S. Perri and R. Stewart, "Accuracy Evaluation of Transposed Convolution-Based Quantized Neural Networks," in *IEEE International Joint Conference on Neural Networks (IJCNN)*, Padua, Italy, 2022.

[20] C. Sestito, R. Stewart and S. Perri, "High-Level Synthesis of Hardware Accelerators for Deconvolution Engines," in *The 15th International Conference on Advances in Circuits, Electronics and Micro-electronics (CENICS)*, Lisbon, Portugal, 2022.

[21] C. Sestito, S. Perri and R. Stewart, "Design-Space Exploration of Quantized Transposed Convolutional Neural Networks for FPGA-based Systems-on-Chip," in *The 20th IEEE International Conference on Pervasive Intelligence and Computing (PiCom)*, Calabria, Italy, 2022.

[22] A. Radford, L. Metz and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," in *The 4th International Conference on Learning Representations (ICLR)*, San Juan, Puerto Rico, 2016.

[23] O. Ronneberger, P. Fischer and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," in *The 18th International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, Munich, Germany, 2015.

[24] S. R. Dubey, S. K. Singh and B. B. Chaudhuri, "Activation functions in deep learning: A comprehensive survey and benchmark," *Neurocomputing*, vol. 503, pp. 92-108, 2022.

[25] K. Hara, D. Saito and H. Shouno, "Analysis of Function of Rectified Linear Unit Used in Deep Learning," in *2015 International Joint Conference on Neural Networks (IJCNN)*, Killarney, Ireland, 2015.

[26] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *Proceedings of the 32nd International Conference on Machine Learning (PMLR)*, Lille, France, 2015.

[27] F. Spagnolo, S. Perri and P. Corsonello, "Approximate Down-Sampling Strategy for Power-Constrained Intelligent Systems," *IEEE Access*, vol. 10, pp. 7073-7081, 2022.

[28] F. Spagnolo, S. Perri and P. Corsonello, "Aggressive Approximation of the SoftMax Function for Power-Efficient Hardware Implementations," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 3, pp. 1652-1656, 2022.

[29] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA, 2016.

[30] V. Badrinarayanan, A. Kendall and R. Cipolla, "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmenation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 12, pp. 2481-2495, 2017.

[31] C. Dong, C. C. Loy and X. Tang, "Accelerating the Super-Resolution Convolutional Neural Network," in *Proceedings of The 14th European Conference on Computer Vision (ECCV)*, Amsterdam, The Netherlands, 2016.

[32] F. Yu and V. Koltun, "Multi-Scale Context Aggregation by Dilated Convolutions," in *Proceedings of The 4th International Conference on Learning Representations (ICLR)*, San Juan, Puerto Rico, 2016.

[33] M. Holschneider, R. Kronland-Martinet, J. Morlet and P. Tchamitchian, "A Real-Time Algorithm for Signal Analysis with the Help of the Wavelet Transform," in *Wavelets*, Springer, 1990, pp. 286-297.

[34] L.-C. Chen, G. Papandreou, F. Schroff and H. Adam, "Rethinking Atrous Convolution for Semantic Image Segmentation," 5 December 2017. [Online]. Available: https://arxiv.org/pdf/1706.05587.pdf. [Accessed 6 October 2022].

[35] C. S. Perone, E. Calabrese and J. Cohen-Adad, "Spinal cord gray matter segmentation using deep dilated convolutions," *Scientific Reports*, vol. 8, no. 5966, pp. 1-13, 2018.

[36] Y. Lyu, L. Bai and X. Huang, "ChipNet: Real-Time LiDAR Processing for Drivable Region Segmentation on an FPGA," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 5, pp. 1769-1779, 2019.

[37] V. Sze, Y.-H. Chen, T.-J. Yang and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295-2329, 2017.

[38] Q. Zhang, M. Zhang, T. Chen, Z. Sun, Y. Ma and B. Yu, "Recent advances in convolutional neural network acceleration," *Neurocomputing*, vol. 323, pp. 37-51, 2019.

[39] J. Cong and B. Xiao, "Minimizing Computation in Convolutional Neural Networks," in *Proceedings of The 24th International Conference on Artificial Neural Networks*, Hamburg, Germany, 2014.

[40] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. G. H. Ong, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra and G. Boudoukh, "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Monterey, California, USA, 2017.

[41] L. Crockett, R. A. Elliot, M. A. Enderwitz and R. W. Stewart, The Zynq Book, Glasgow: Strathclyde Academic Media, 2014.

[42] N. Shah, P. Chaudhari and K. Varghese, "Runtime Programmable and Memory Bandwidth Optimized FPGA-Based Coprocessor for Deep Convolutional Neural

Network," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 12, pp. 5922-5934, 2018.

[43] F. Spagnolo, S. Perri, F. Frustaci and P. Corsonello, "Energy-Efficient Architecture for CNNs Inference on Heterogeneous FPGA," *Journal of Low Power Electronics and Applications*, vol. 10, no. 1, pp. 1-17, 2020.

[44] Y. Shen, M. Ferdman and P. Milder, "Maximizing CNN Accelerator Efficiency Through Resource Partitioning," in *Proceedings of The 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, Toronto, ON, Canada, 2017.

[45] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," 2016. [Online]. Available: https://arxiv.org/pdf/1602.07360.pdf. [Accessed 6 September 2022].

[46] A. Yazdanbakhsh, M. Brzozowski, B. Khaleghi, S. Ghodrati, K. Samadi, N. S. Kim and H. Esmaeilzadeh, "FlexiGAN: An End-to-End Solution for FPGA Acceleration of Generative Adversarial Networks," in *Proceedings of The 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Boulder, CO, USA, 2018.

[47] J.-W. Chang, K.-W. Kang and S.-J. Kang, "An Energy-Efficient FPGA-Based Deconvolutional Neural Networks Accelerator for Single Image Super-Resolution," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 30, no. 1, pp. 281-295, 2020.

[48] X. Zhang, S. Das, O. Neopane and K. Kreutz-Delgado, "A Design Methodology for Efficient Implementation of Deconvolutional Neural Networks on an FPGA," 2017. [Online]. Available: https://arxiv.org/pdf/1705.02583.pdf. [Accessed 6 September 2022].

[49] J. Yan, S. Yin, F. Tu, L. Liu and S. Wei, "GNA: Reconfigurable and Efficient Architecture for Generative Network Acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2519-2529, 2018.

[50] Y. Yu, T. Zhao, M. Wang, K. Wang and L. He, "Uni-OPU: An FPGA-Based Uniform Accelerator for Convolutional and Transposed Convolutional Networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 7, pp. 1545-1556, 2020.

[51] P. Getreuer, "Linear Methods for Image Interpolation," [Online]. Available: https://www.ipol.im/pub/art/2011/g_lmii/article.pdf. [Accessed 6 September 2022].

[52] X. Wu, Y. Ma, M. Wang and Z. Wang, "A Flexible and Efficient FPGA Accelerator for Various Large-Scale and Lightweight CNNs," *IEEE Transactions on Circuits and Systems I: Regular Papers,* vol. 69, no. 3, pp. 1185-1198, 2022.

[53] A. Paszke, A. Chaurasia, S. Kim and E. Culurciello, "ENet: A Deep Neural Network Architecture for Real-Time Semantic Segmentation," 2016. [Online]. Available: https://arxiv.org/pdf/1606.02147.pdf. [Accessed 6 September 2022].

[54] J. Long, E. Shelhamer and T. Darrell, "Fully Convolutional Networks for Semantic Segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Boston, MA, USA, 2015.

[55] S. Hussain, M. Javaheripi, P. Neekhara, R. Kastner and F. Koushanfar, "FastWave: Accelerating Autoregressive Convolutional Neural Networks on FPGA," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Westminster, Colorado, USA, 2019.

[56] M. Carreras, G. Deriu and P. Meloni, "Flexible Acceleration of Convolutions on FPGAs: planning NEURAghe 2.0," in *CPS Summer School PhD Workshop*, Alghero, Italy, 2019.

[57] M. Lin, Q. Chen and S. Yan, "Network In Network," 4 March 2014. [Online]. Available: https://arxiv.org/pdf/1312.4400.pdf. [Accessed 6 October 2022].

[58] Xilinx, "Zynq-7000 SoC Data Sheet: Overview," 2 July 2018. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview. [Accessed 7 October 2022].

[59] ARM, "AMBA 4 AXI4, AXI4-Lite, and AXI4-Stream Protocol Assertions User Guide," 23 July 2012. [Online]. Available: https://developer.arm.com/documentation/dui0534/b/. [Accessed 6 October 2022].

[60] Xilinx, "Zynq UltraScale+ MPSoC Data Sheet: Overview (DS891)," [Online]. Available: https://docs.xilinx.com/v/u/en-US/ds891-zynq-ultrascale-plus-overview. [Accessed 10 October 2022].

[61] F. Milletari, N. Navab and S.-A. Ahmadi, "V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation," in *2016 Fourth International Conference on 3D Vision (3DV)*, Stanford, CA, USA, 2016.

[62] C. Chen, M.-Y. Liu, O. Tuzel and J. Xiao, "R-CNN for Small Object Detection," in *13th Asian Conference on Computer Vision*, Taipei, Taiwan, 2016.

[63] C. Tian, Y. Xu, W. Zuo, B. Zhang, L. Fei and C.-W. Lin, "Coarse-to-Fine CNN for Image Super-Resolution," *IEEE Transactions on Multimedia,* vol. 23, pp. 1489-1502, 2021.

[64] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," 12 January 2018. [Online]. Available: https://arxiv.org/pdf/1603.07285.pdf. [Accessed 6 September 2022].

[65] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, S. Biswa and A. A. Bharath, "Generative Adversarial Networks: An Overview," *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 53-65, 2018.

[66] S. Liu, H. Fan, X. Niu, H. Ng, Y. Chu and W. W. Luk, "Optimizing CNN-based Segmentation with Deeply Customized Convolutional and Deconvolutional Architectures on FPGA," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 11, no. 19, pp. 1-22, 2018.

[67] S. Liu and W. Luk, "Towards an Efficient Accelerator for DNN-Based Remote Sensing Image Segmentation on FPGAs," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, Barcelona, Spain, 2019.

[68] D. Wang, J. Shen, M. Wen and C. Zhang, "Efficient Implementation of 2D and 3D Sparse Deconvolutional Neural Networks with a Uniform Architecture on FPGAs," *Electronics*, vol. 8, no. 803, pp. 1-13, 2019.

[69] J.-W. Chang, S. Ahn, K.-W. Kang and S.-J. Kang, "Towards Design Methodology of Efficient Fast Algorithms for Accelerating Generative Adversarial Networks on FPGAs," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Beijing, China, 2020.

[70] X. Di, H.-G. Yang, Y. Jia, Z. Huang and N. Mao, "Exploring Efficient Acceleration Architecture for Winograd-Transformed Transposed Convolution of GANs on FPGAs," *Electronics*, vol. 9, no. 286, pp. 1-21, 2020.

[71] Xilinx, "Vivado Design Suite User Guide (v2019.2)," 30 October 2019. [Online]. Available: https://docs.xilinx.com/v/u/2019.2-English/ug910-vivado-getting-started. [Accessed 10 October 2022].

[72] Xilinx, "AXI DMA v7.1, LogiCORE IP Product Guide (PG021)," [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf. [Accessed 6 October 2022].

[73] Xilinx, "AXI Video Direct Memory Access v6.2, LogiCORE IP Product Guide (PG020)," [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_vdma/v6_2/pg020_axi_vdma.pdf. [Accessed 6 October 2022].

[74] Xilinx, "AXI4-Stream Infrastructure IP Suite v3.0 LogiCORE IP Product Guide (PG085)," [Online]. Available:

https://www.xilinx.com/support/documentation/ip_documentation/axis_infrastructure_ip_suite/v1_1/pg085-axi4stream-infrastructure.pdf. [Accessed 6 October 2022].

[75] S. Liu, C. Zeng, H. Fan, H.-C. Ng, J. Meng, Z. Que, X. Niu and W. Luk, "Memory-Efficient Architecture for Accelerating Generative Networks on FPGA," in *2018 International Conference on Field-Programmable Technology (FPT)*, Naha, Okinawa, Japan, 2018.

[76] Xilinx, "Virtex-7 Product Brief," [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/product-briefs/virtex7-product-brief.pdf. [Accessed 10 October 2022].

[77] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang and H. Yang, "Angel-Eye: A Complete Design Flow for Mapping CNN Onto Embedded FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 35-47, 2018.

[78] F. Spagnolo, S. Perri, F. Frustaci and P. Corsonello, "Reconfigurable Convolution Architecture for Heterogeneous Systems-on-Chip," in *2020 9th Mediterranean Conference on Embedded Computing (MECO)*, Virtual, 2020.

[79] Y. Li, S. Lu, J. Luo, W. Pang and H. Liu, "High-performance Convolutional Neural Network Accelerator Based on Systolic Arrays and Quantization," in *2019 IEEE 4th International Conference on Signal and Image Processing (ICSIP)*, Wuxi, China, 2019.

[80] S. Zhang, J. Cao, Q. Zhang, Q. Zhang, Y. Zhang and Y. Wang, "An FPGA-Based Reconfigurable CNN Accelerator for YOLO," in *2020 IEEE 3rd International Conference on Electronics Technology (ICET)*, Chengdu, China, 2020.

[81] 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)*, Waltham, MA, USA, 2019.

[82] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," [Online]. Available: https://arxiv.org/pdf/1612.08242.pdf. [Accessed 10 October 2022].

[83] K. Chatfield, K. Simonyan, A. Vedaldi and A. Zisserman, "Return of the Devil in the Details: Delving Deep into Convolutional Nets," 5 November 2014. [Online]. Available: https://arxiv.org/pdf/1405.3531.pdf. [Accessed 10 October 2022].

[84] W. Mao, J. Lin and Z. Wang, "F-DNA: Fast Convolution Architecture for Deconvolutional Network Acceleration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 8, pp. 1867-1880, 2020.

[85] Z. Tang, G. Luo and M. Jiang, "FTConv: FPGA Acceleration for Transposed Convolution Layers in Deep Neural Networks," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Seaside, CA, USA, 2019.

[86] B. Shi, Z. Tang, G. Luo and M. Jiang, "Winograd-Based Real-Time Super-Resolution System on FPGA," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, Tianjin, China, 2019.

[87] K. He, X. Zhang, S. Ren and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," in *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, Santiago, Chile, 2015.

[88] Xilinx, "Kintex-7 FPGA Family Product Brief," [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/product-briefs/kintex7-product-brief.pdf. [Accessed 10 October 2022].

[89] S. Lee, S. Joo, H. K. Ahn and S.-O. Jung, "CNN Acceleration With Hardware-Efficient Dataflow for Super-Resolution," *IEEE Access*, vol. 8, pp. 187754-187765, 2020.

[90] M. Bevilacqua, A. Roumy, C. Guillemot and M. L. Alberi-Morel, "Low-complexity single-image super-resolution based on nonnegative neighbor embedding," in *Proceedings of the 23rd British Machine Vision Conference (BMVC)*, Surrey, UK, 2012.

[91] R. Zeyde, M. Elad and M. Protter, "On Single Image Scale-Up Using Sparse-Representations," in *International Conference on Curves and Surfaces*, Avignon, France, 2010.

[92] D. Martin, C. Fowlkes, D. Tal and J. Malik, "A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics," in *Proceedings Eighth IEEE International Conference on Computer Vision (ICCV)*, Vancouver, BC, Canada, 2001.

[93] Z. Wang, A. Bovik, H. Sheikh and E. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600-612, 2004.

[94] W. Shi, J. Cao, Q. Zhang, Y. Li and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637-646, 2016.

[95] E. Wang, J. J. Davis, R. Zhao, H.-C. Ng, X. Niu, W. Luk, P. Y. Cheung and G. A. Constantinides, "Deep Neural Network Approximation for Custom Hardware: Where We've Been, Where We're Going," *ACM Computing Surveys*, vol. 52, no. 2, pp. 1-39, 2020.

[96] V. Radu, K. Kaszyk, Y. Wen, J. Turner, J. Cano, E. J. Crowley, B. Franke, A. Storkey and M. O'Boyle, "Performance Aware Convolutional Neural Network Channel Pruning for Embedded GPUs," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*, Orlando, FL, USA, 2019.

[97] Y. Tian, D. Krishnan and P. Isola, "Contrastive Representation Distillation," in *Proceedings of 8th International Conference on Learning Representations (ICLR)*, Virtual, 2020.

[98] J. Su, N. J. Fraser, G. Gambardella, M. Blott, G. Durelli, D. B. Thomas , P. H. Leong and P. Y. Cheung, "Accuracy to Throughput Trade-Offs for Reduced Precision Neural Networks on Reconfigurable Logic," in *Proceedings of The 14th International Symposium on Applied Reconfigurable Computing (ARC)*, Santorini, Greece, 2018.

[99] V. Rybalkin, A. Pappalardo, M. M. Ghaffar, G. Gambardella, N. Wehn and M. Blott, "FINN-L: Library Extensions and Design Trade-Off Analysis for Variable Precision LSTM Networks on FPGAs," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, Dublin, Ireland, 2018.

[100] R. Stewart, A. Nowlan, P. Bacchus, Q. Ducasse and E. Komendantskaya, "Optimising Hardware Accelerated Neural Networks with Quantisation and a Knowledge Distillation Evolutionary Algorithm," *Electronics*, vol. 10, no. 4, pp. 1-21, 2021.

[101] D. Wan, F. Shen, L. Liu, F. Zhu, L. Huang, M. Yu, H. T. Shen and L. Shao, "Deep quantization generative networks," *Pattern Recognition*, vol. 105, pp. 1-12, 2020.

[102] J. S. Choi, J. Kim and J. H. Ko, "A Weight-Sharing Autoencoder with Dynamic Quantization for Efficient Feature Compression," in *2021 International Conference on Information and Communication Technology Convergence (ICTC)*, Jeju Island, Korea, 2021.

[103] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford and X. Chen, "Improved Techniques for Training GANs," in *Proceedings Conference on Neuarl Information Processing Systems (NIPS)*, Barcelona, Spain, 2016.

[104] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler and S. Hochreiter, "GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium," in *Proceedings of Conference on Neural Information Processing Systems (NIPS)*, Long Beach, CA, USA, 2017.

[105] Y. LeCun, C. Cortes and C. J. Burges, "The MNIST Database of handwritten digits," [Online]. Available: http://yann.lecun.com/exdb/mnist/.

[106] H. Xiao, K. Rasul and R. Vollgraf, "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms," 15 September 2017. [Online]. Available: https://arxiv.org/pdf/1708.07747.pdf. [Accessed 24 October 2022].

[107] A. Krizhevsky and G. Hinton, "Learning Multiple Layers of Features from Tiny Images," 8 April 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf. [Accessed 24 October 2022].

[108] Z. Liu, P. Luo, X. Wang and X. Tang, "Deep Learning Face Attributes in the Wild," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Santiago, Chile, 2015.

[109] O. M. Parkhi, A. Vedaldi, A. Zisserman and C. Jawahar, "Cats and dogs," in *2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Rhode Island, 2012.

[110] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler , R. Benenson, U. Franke, S. Roth and B. Schiele, "The Cityscapes Dataset for Semantic Urban Scene Understanding," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA, 2016.

[111] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv and Y. Bengio, "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations," *Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869-6898, 2017.

[112] The Linux Foundation, "PyTorch," [Online]. Available: https://pytorch.org/. [Accessed 24 October 2022].

[113] A. Pappalardo, "Xilinx Brevitas," [Online]. Available: https://doi.org/10.5281/zenodo.3333552. [Accessed 24 October 2022].

[114] J. Xu, Z. Li, B. Du, M. Zhang and J. Liu, "Reluplex made more practical: Leaky ReLU," in *2020 IEEE Symposium on Computers and Communications (ISCC)*, Rennes, France, 2015.

[115] 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 (CVPR)*, Boston, MA, USA, 2015.

[116] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Miami, FL, USA, 2009.

[117] NVIDIA Corp., "NVIDIA® Tesla® T4," [Online]. Available: https://www.nvidia.com/en-us/data-center/tesla-t4. [Accessed 24 October 2022].

[118] Google, "Google Colaboratory," [Online]. Available: https://colab.research.google.com/. [Accessed 24 October 2022].

[119] NVIDIA Corp., "NVIDIA® Tesla® K80," 24 October 2022. [Online]. Available: https://www.nvidia.com/en-gb/data-center/tesla-k80/.

[120] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE,* vol. 86, no. 11, pp. 2278-2324, 1998.

[121] I. Goodfellow, "NIPS 2016 Tutorial: Generative Adversarial Networks," 3 April 2017. [Online]. Available: https://arxiv.org/pdf/1701.00160.pdf. [Accessed 24 October 2022].

[122] S. Barratt and R. Sharma, "A Note on the Inception Score," 21 June 2018. [Online]. Available: https://arxiv.org/pdf/1801.01973.pdf. [Accessed 24 October 2022].

[123] E. Marrazzo, F. Spagnolo and S. Perri, "Runtime Reconfigurable Hardware Accelerator for Energy-Efficient Transposed Convolutions," in *2022 17th Conference on Ph.D Research in Microelectronics and Electronics (PRIME)*, Villasimius, Sardegna, Italy, 2022.

[124] Digilent, "Pynq-Z1 Reference Manual," [Online]. Available: https://digilent.com/reference/programmable-logic/pynq-z1/reference-manual. [Accessed 24 October 2022].

[125] Xilinx, "Integrated Logic Analyzer v6.2 LogiCORE IP Product Guide (PG172)," [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg172-ila. [Accessed 24 October 2022].

[126] Y. Meng, S. Kuppannagari, R. Kannan and V. Prasanna, "How to Avoid Zero-Spacing in Fractionally-Strided Convolution? A Hardware-Algorithm Co-Design Methodology," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, Bengaluru, India, 2021.

[127] Xilinx, "Alveo U200 and U250 Product Brief," [Online]. Available: https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/alveo-product-brief.pdf. [Accessed 24 October 2022].

[128] Xilinx, Inc, "Vivado Design Suite User Guide: High-Level Synthesis," 13 Jan 2020. [Online]. Available: https://docs.xilinx.com/v/u/2019.2-English/ug902-vivado-high-level-synthesis. [Accessed 11 Aug 2022].