Journal of Signal Processing Systems (2023) 95:1245–1263
https://doi.org/10.1007/s11265-023-01883-7

FPGA Design of Transposed Convolutions for Deep Learning Using High-Level Synthesis

Cristian Sestito[1,2] · **Stefania Perri**[3] · **Robert Stewart**[4]

Received: 30 November 2022 / Revised: 28 April 2023 / Accepted: 13 July 2023 / Published online: 4 August 2023

Abstract
Deep Learning (DL) is pervasive across a wide variety of domains. Convolutional Neural Networks (CNNs) are often used for image processing DL applications. Modern CNN models are growing to meet the needs of more sophisticated tasks, e.g. using Transposed Convolutions (TCONVs) for image decompression and image generation. Such state-of-the-art DL models often target GPU-based high-performance architectures, due to the high computational and hardware resource needs of TCONV layers. To avoid prohibitive GPU energy costs, CNNs are increasingly deployed to decentralized embedded autonomous devices, such as Field Programmable Gate Arrays (FPGAs). However, this poses challenges for designing efficient hardware implementations of TCONV layers. This paper presents a parameterized design and implementation of a new TCONV module, which is synthesizable onto FPGAs. It is implemented using the High-Level Synthesis (HLS), through a C++ template to parameterize its functional and non-functional properties. These parameters allow kernel sizes, image sizes, quantization and parallelism to be varied by users. With a systematic exploration in this design space, we find an optimal instance of this TCONV module that achieves 6.25 Giga Outputs per Second (*Gout/s*) using just 1.53 W of power. We then use our TCONV layer in two neural networks for image decompression and image generation. Image decompression achieves a speed throughput of more than 30K frames-per-second (*fps*) using only the 16% of resources on average, image generation achieves an energy efficiency of 324 *fps*/W and outperforms comparable state-of-the-art models by at least 7.3×.

Keywords Transposed Convolution · Deep Learning · FPGA · High-Level Synthesis · Quantization · Parallelism

1 Introduction

Convolutional Neural Networks (CNNs) have gained widespread adoption in several applications, such as image processing [1], speech recognition [2] and robotics [3].

Input data, in the form of multi-dimensional arrays, are managed through a sequence of computing layers that modulate the space representation to handle the complexity of information. While down-sampling layers progressively compress data to extract relevant features, up-sampling stages act in the opposite way, by predicting new informative content to be arranged within a wider space.

Among several up-sampling layers, Transposed Convolutions (TCONVs) work well with image processing tasks. Indeed, they exploit learnable filters to produce high-resolution images starting from low-resolution representations. They are used for image generation through adversarial learning, where Generative Adversarial Networks (GANs) [4] build new images similar to those belonging to the training dataset. The detection of fake

✉ Cristian Sestito
cristian.sestito@unical.it

✉ Robert Stewart
R.Stewart@hw.ac.uk

Stefania Perri
stefania.perri@unical.it

[1] Department of Informatics, Modeling, Electronics and System Engineering, University of Calabria, 87036 Rende, Italy

[2] Current Affiliation: Centre for Electronics Frontiers, School of Engineering, The University of Edinburgh, Edinburgh EH9 3BF, UK

[3] Department of Mechanical, Energy and Management Engineering, University of Calabria, 87036 Rende, Italy

[4] Department of Computer Science, Heriot-Watt University, Edinburgh EH14 4AS, UK

images in social media is an application scenario [5]. TCONVs are also suitable to implement pixel-level classification, or semantic segmentation, to highlight different objects within an image [6]. For instance, medical image segmentation makes use of TCONV-based decoders to analyze optic discs, retinal vessels and lungs [7]. Super-resolution imaging [8] is another scenario that benefits from such up-sampling layers to deal with virtual and augmented reality through smart head-mounted displays [9].

However, the performance of TCONVs are often impacted by high computational complexity. Indeed, they require input data pre-processing that results in more computations compared with conventional Convolutions (CONVs) [10]. Low-latency applications are extremely susceptible to this detrimental effect, thus demanding highly-parallelizable devices. Mainstream Graphics Processing Units (GPUs) effectively meet the parallelism constraint, but at the cost of a higher power dissipation [11]. Field Programmable Gate Arrays (FPGAs), other than offering a reasonable trade-off in terms of speed and power, also provide a flexible substrate suitable to not only deploy parallel CNNs, but to also infer either alternative algorithmic strategies or compression techniques (e.g., data quantization) to further improve the overall efficiency with respect to GPUs. FPGA can address the complexity issue by skipping redundant computations [12], or revisiting the conventional data pre-processing [13, 14]. The FPGA implementation of a 16-bit TCONV-based GANs [12], where redundant computations are skipped, outperformed the energy-efficiency on GPU by a ~ 3× factor. More aggressive quantization may boost such improvement, as shown in image classification CNNs implemented on FPGAs [15, 16]. However, the impact of deep quantization over up-sampling models using TCONVs is still under-explored.

Motivated by all these preliminary considerations, we have recently investigated the joint impact of parallelism and quantization over a simple Transposed Convolutional Neural Network (TCNN), implemented on FPGA [17], that deals with image decompression. The High-Level Synthesis (HLS) paradigm has been adopted, since the latter allows (1) the architecture to be platform-independent, and (2) parametric C++ templates to be used to investigate different configurations with minimal top-level modifications. We have performed an extensive design-space exploration, by either varying the input and output parallelism (i.e., the number of input images and output images processed in parallel) or the data bit-width, to determine the optimum configuration, by using the commercial XC7Z020 FPGA. We have observed that the architecture is able to decompress MNIST [18] and Fashion-MNIST [19] with only the ~ 2.5% accuracy loss when moving from 8 to 4 bits. Parallelism provides a 3.5× speed-up, whilst only requiring less than the 10% of Look-Up Tables (LUTs).

This work extends the previous investigations and provides further lines of research. Specifically, the new contributions can be summarized as follows:

- An extended review about state-of-the-art TCNNs deployed on FPGAs (Section 2).
- A detailed presentation of the C++ TCONV layer template, suitable to be implemented within dataflow TCNNs. The impact of the specific TCONV parameters over the resources utilization and the speed throughput are presented, as well as comparisons with some state-of-the-art competitors (Section 3). When implemented within the XC7Z100 FPGA device, the architecture provides 6.25 Giga Outputs per Second (*Gout/s*), whilst dissipating only 1.53 W, using ~ 11.7k LUTs and 4.52 Mb of on-chip memory.
- A high-level design space exploration of the TCONV decoder from [17], using C++ *#pragmas* to control dataflow and data parallelism, memory resources, pipelining and loop unrolling. In particular, the trade-off between resources utilization and throughput is discussed (Section 4). The analysis, carried out in the range 8–4 bits, shows an improvement of 3 orders of magnitude in the number of frames-per-second (*fps*), with a 16% average resources utilization at most.
- The characterization of a more complex TCNN, dealing with image generation through the Deep Convolutional Generative Adversarial Network (DCGAN) paradigm [20]. The impact of parallelism over the resources utilization and throughput is presented, as well as comparisons with state-of-the-art DCGAN accelerators (Section 5). The characterization at 5-bits highlighted an energy efficiency of 324.32 *fps*/W, which outperforms the art by at least ~ 7.3×.

Finally, conclusions are drawn in Section 6.

2 Background and Related Works

Several CNNs take advantage of TCONV layers, including architectures tailored for image generation and models conceived to either decompress data or provide super-resolution images. The former includes the well-known DCGAN [20] and consists of two networks, namely the discriminator and the generator. Training aims at strengthening the generator to build realistic images similar to the actual dataset under examination. While the discriminator consists of consecutive Convolution (CONV) layers that progressively down-sample the informative content for binary classification, the generator stacks multiple TCONV layers to build an image starting from a latent vector. Conversely, models for image decompression and super-resolution usually rely on an

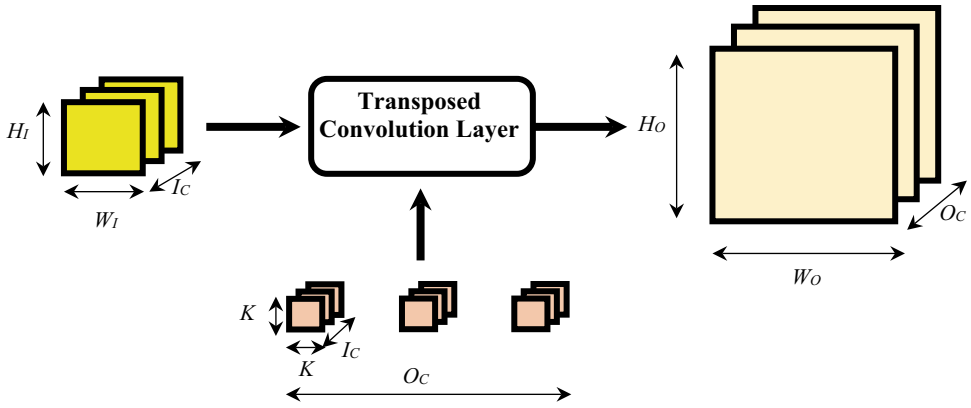


Figure 1 Example of TCONV layer with $I_C=3$ and $O_C=3$.

encoder-decoder architecture. For instance, the Fast Super-Resolution CNN (FSRCNN) [8] uses an encoder, made of seven CONVs layers, to extract meaningful features from inputs. The decoder adopts just a single TCONV layer to provide the final high-resolution image.

The generic TCONV layer receives a set of I_C input feature maps (*ifmaps*), each consisting of $H_I \times W_I$ activations, and a set of O_C 3-D filters, each consisting of I_C kernels of $K \times K$ weights. As a result, the layer provides a set of high-resolution O_C *output feature maps* (*ofmaps*), each $H_O \times W_O$-sized. The *ofmaps* sizes are proportional to those related to the *ifmaps* by a factor S, known as stride or up-sampling factor. Each 3-D filter performs a 3-D TCONV processing the I_C *ifmaps*, thus generating one of the O_C *ofmaps*. Optionally, O_C biases can be finally summed up to the *ofmaps* activations. By a top-level viewpoint, with no reference to the specific algorithm adopted, as depicted in Fig. 1, the generic TCONV layer practically matches the exoskeleton of conventional CONV layers. However, in order to meet the up-sampling behavior, the needed computations may significantly exceed those required by CONV layers. Indeed, the last TCONV layer of the FSRCNN may require up to ~6.75 more Multiply-Accumulations (MACs) with respect to the CONV layers [10]. This is due to the fact that, conventionally, TCONVs can be treated as direct CONVs over a dilated representation of the generic *ifmap*. To better understand this point, let us consider a 4×4 *ifmap* that is subjected to a filter having just one 2×2 kernel (Fig. 2); we also suppose $S=2$. Firstly, the *ifmap* is up-sampled, by interleaving S–1 zeros between adjacent activations. Then, a direct CONV is performed between the dilated *ifmap* and the 2×2 kernel. As a result, each output activation is given by 2×2 MACs. In total, considering that 64 activations are generated, this TCONV costs 256 MACs. In the case of a conventional CONV, the overall cost would have been ~86% lower.

High computational complexity makes it difficult to achieve low latency performance, typically required by real-time systems. As a result, efforts to accelerate TCNNs on dedicated hardware, such as FPGAs, have been undertaken in recent years [10, 12–14, 21–25, 27, 28, 30]. Different hardware-oriented algorithmic strategies have been investigated, with the aim to trade-off the resources utilization, the throughput and the power dissipation. The dilation of *ifmaps*, through zeros insertion, is the primitive way to infer the up-sampling capability to conventional CONV engines. This strategy was effectively managed in [12] through the FlexiGAN framework, which skips the redundant computations exhibited by the zeros insertion. This is accomplished by adding extra control logic to coach the computing core about the actual patterns to be managed (i.e., the patterns of non-zero values). As an alternative approach, the multi-channel-multi-kernel parallel algorithm was presented in [21], which rearranges $K \times K$ TCONVs into K^2 separate 1×1 CONVs to avoid patterns with zeros.

With the aim of transforming TCONVs into CONVs, the algorithm proposed in [10] decomposes wide filters in multiple sub-filters, to be then processed by as many CONV engines. Starting from the observation that specific TCONVs with zeros insertion exhibit $S \times S$ regular patterns of actual computations, the generic filter can be split into $S \times S$ smaller filters, each consisting of a different number of weights, before being supplied to the CONV engine. However, the different number of weights of each pattern results

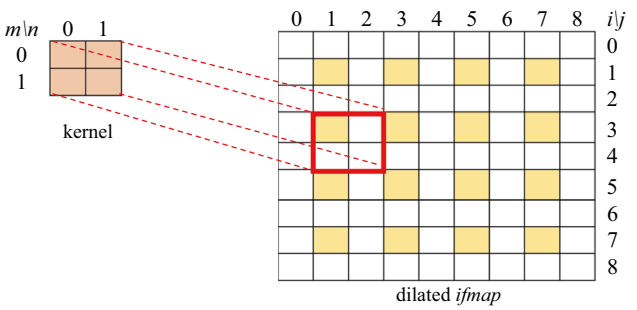


Figure 2 Example of a TCONV between a 4×4 *ifmap* and a 2×2 kernel. The yellow cells within the dilated *ifmap* represent the actual input activations.

in computational imbalance. The latter was addressed by the iterative filters sub-splitting proposed in [22], where the achieved efficacy was evaluated over both the FSRCNN [8] and the DCGAN [20] models. Both the above approaches [10, 22] need to pre-process filters off-chip, by making challenging the run-time adaptability to different configurations. In order to avoid preliminary transformations, as demonstrated in [13], *ifmaps'* activations can be properly re-arranged at run-time. The equivalent reconfigurable circuit was made capable to support different filter sizes and benchmarked using the FSRCNN [8]. Finally, filters management was also taken into account in Uni-OPU [23] to both addresses zero-insertion in TCONV and offer adaptability to the nearest-neighboring up-sampling method that, in turn, dilates the *ifmaps* by replicating pixels instead of inserting zeros.

Alternative strategies strove to either leverage the Winograd transformation [29] or to adopt hybrid computations to alleviate the high computational complexity. The former strategy was examined in [24], where element-wise multiplication manipulation was used to model multiplications as simpler additions and shift operations. Conversely, the architecture in [25] adopted a mixed approach in which part of TCONVs are replaced by average computations to minimize the overall MACs. The key benefit of such an approach is to improve the throughput noticeably, at the cost of lower accuracy.

The Input-Oriented Mapping (IOM) strategy [26] uses a completely different algorithm that produces the same outputs of the previous approaches, but avoids both zeros insertion and sub-filters management. The $H_I \times W_I$ activations of the generic *ifmap* are multiplied by the $K \times K$ kernel weights, thus providing $H_I \times W_I$ provisional output windows. When $K > S$, adjacent windows overlap, by sharing K–S columns and rows. The overlapping areas are summed up to provide the actual results. The architecture presented in [14] firstly adopted the IOM on FPGAs, with reverse looping to avoid accumulations for overlaps, by taking into account the input coordinates for each iteration. Conversely, the reconfigurable engine proposed in [27] dealt with overlaps by carefully using the on-chip Digital Signal Processing (DSP) slices to boost the speed performance at limited power dissipation. Results reported in [28] demonstrated the suitability of both the IOM and neural network compression to manage 2-D and 3-D GANs on FPGAs. They also shown that filters pruning allows low-weights connections to be removed and, accordingly, the computational efficiency to be improved. Authors in [30] dealt with semantic segmentation by proposing a parameterizable architecture to comply with different neural networks. They also compared the FPGA results with the equivalent software on GPU. While the latter dissipates at least 147 W on the NVIDIA Titan X (Pascal) GPU, the FPGA accelerator used only 9.6 W.

In this work, we further extend the investigations about the FPGA implementation of IOM-based TCONV layers, by proposing a platform-independent HLS template, able to be adapted either to very low bit-width or to high parallelism, in accordance with the constraint of the specific deep learning task to be performed. The synthesis tool is supplied by HLS *#pragmas* to balance the trade-off between resources utilization, speed throughput and power dissipation, in order to achieve the highest energy efficiency.

3 Design and Characterization of the HLS-Based Transposed Convolution Layer

The generic TCONV layer, used within the TCNNs discussed in this work, is described at the C++ abstraction. The template is made parameterizable in terms of (a) the bit-width N, (b) the TCONV parameters (i.e., the kernel size K, the stride S, the *ifmaps* sizes H_I, W_I), and (c) the input and output parallelism factors (i.e., T_{IC} and T_{OC}). These parameters allow either characterizing different configurations onto the same device or implementing a specific architecture on different devices.

3.1 The Proposed Design

The algorithm for the proposed TCONV layer is reported in Listing 1. The equivalent architecture is made able to process T_{IC} *ifmaps* in parallel, using $T_{OC} \times T_{IC}$ filters and T_{OC} biases at the same time. While the activations are the actual inputs, all the weights and biases are preliminarily stored on-chip. At the completion of the computations, the circuit provides T_{OC} *ofmaps* in parallel. Figure 3 illustrates an example of top-level block diagram when $T_{IC} = 2$ and $T_{OC} = 2$. Considering the output parallelism factor, line 1 highlights that O_C/T_{OC} steps are required to generate all the *ofmaps*. Referring to Listing 1, for each iteration of the loop in line 1, the circuit takes $I_C/T_{IC} + 1$ steps (line 2). The first I_C/T_{IC} steps are needed to compute the 3-D TCONVs between the T_{IC} *ifmaps* and the $T_{OC} \times T_{IC}$ filters, to provide a group of T_{OC} *ofmaps*. The extra step is used to (a) add the T_{OC} biases, and (b) move the actual outputs out, after being temporarily stored within on-chip memories.

Listing 1 The pseudocode of the Transposed Convolution Layer

```
Inputs: Stream of T_IC ifmaps,
        O_C×I_C×K×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 and biases;
5:        for hi=0 to H_I−1 do
6:          for wi=0 to W_I−1 do
7:            #pragma HLS PIPELINE II=1
8:            Perform 3-D TCONVs using the IOM method
9:          end for
10:       end for
11:     else
12:       for iter=0 to H_O*W_O−1 do
13:         #pragma HLS PIPELINE II=1
14:         Biases accumulations and outputs movement
15:       end for
16:     end if
17:   end for
18: end for
```

Listing 2 details the 3-D TCONV that makes use of the IOM strategy carried out on line 8 in Listing 1. In what follows, the reported lines refer to Listing 2, unless otherwise stated. The *#pragma HLS PIPELINE* (line 3) ensures that the underlining loops are parallel performed in a pipeline fashion. For each clock cycle, as stated by the *Initiation Interval* (II = 1), a stream of T_{IC} activations is read and saved in the array *inAct* (line 4). The T_{IC} activations are multiplied by the respective $T_{OC} \times T_{IC} \times K \times K$ weights of the array *filt*, and subjected to columns overlap. We refer to weights to indicate the trained parameters of the generic neural network.

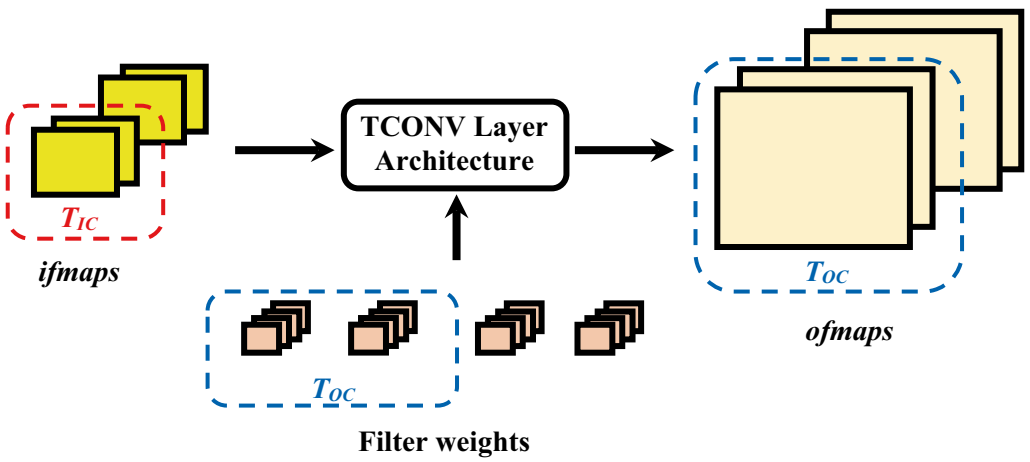


Figure 3 Parallel inputs and outputs of the TCONV Layer when $T_{IC} = 2$ and $T_{OC} = 2$.

Listing 2 Detail of the *Input-Oriented Mapping* 3-D TCONVs

```
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 = inStream.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)*filt(toc,tic,kr,kc);
12:       else
13:        outCol(toc,tic,kr,kc) = inAct(tic*N+N–1:tic*N)*filt(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)*filt(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
```

As a result, $T_{OC} \times T_{IC}$ windows of $K \times K$ provisional results are generated each cycle. Within each window, the locations having row index $kr < S$ and column index $kc < S$ are final, while the remaining cells must be overlapped to those belonging to subsequent windows, either in column- or row-sense. Firstly, column overlaps are managed in lines 9–20. There, while the *outCol* array collects the provisional results, the *ColBuff* array is responsible to temporarily store the products to be overlapped. The column overlap takes place by accumulating the products belonging to the first $K–S$ columns of the current windows with those belonging to the last $K–S$ columns of the windows

computed in the previous clock cycle (line 13). Accordingly, *ColBuff* saves the current products having column index $kc \geq S$ (lines 18–20). The processing of the very first T_{IC} activations does not entail accumulations, as stated by the body of lines 10–11.

Lines 21–36 report the more complex management of row overlaps. In this case, while the *outRow* array stores the new computations, the buffer *RowBuff* stores the provisional results to be overlapped. Specifically, the first *K*–*S* rows of the current windows are accumulated to the last *K*–*S* rows of the windows previously computed, and which share the same column indices. The latter were computed W_I clock cycles before according to the raster-order alignment. In addition, considering that the column overlap implies that only the first *S* outputs of each row are valid for each clock cycle, row overlap also takes care of the latter consideration (lines 21–22). Accordingly, *RowBuff* saves the current outputs having row index $kr \geq S$ and column index $kc < S$ (lines 32–36). The processing of the T_{IC} activations belonging to the first row of the generic *ifmap* does not entail accumulations, as stated by the body of lines 23–24.

To better explain the behavior of the IOM strategy, Fig. 4 illustrates the example in which an *ifmap* having $H_I = W_I = 2$ is processed by a filter with $K = 3$ and the stride $S = 2$. The input activation I_{00} is multiplied by the weights *Wij,* with $i = 0, \ldots, 2$ and $j = 0, \ldots, 2$. As a result, the provisional output

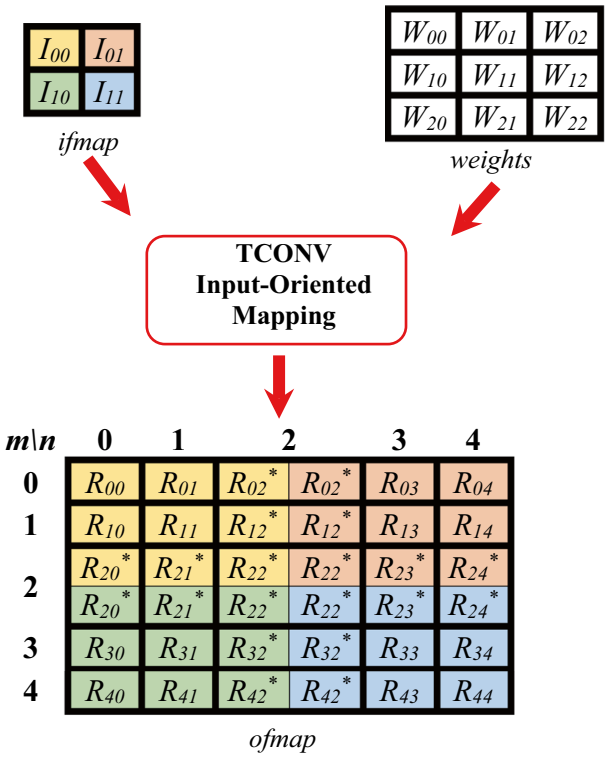


Figure 4 Example of IOM-based TCONV between a 2×2 *ifmap* and a 3×3 weights kernel.

window, associated to the array *outCol*, and having activations *Rmn*, with $m = 0, \ldots, 2$ and $n = 0, \ldots, 2$, is placed within the output space. While the activations having $m = 0$, 1 and $n = 0$, 1 are final, the remaining activations must be accumulated either along the columns or along the rows with the results provided by the subsequent products. All these activations are followed by an asterisk, which means that they concur to compose the final results to be placed in the generic (*m.n*) position. Specifically, the activations with $n = 2$ are stored within the *ColBuff* array for column overlap during the next cycle. Conversely, activations with $m = 2$ and $n = 0$, 1 are temporarily stored within the *RowBuff* array for subsequent row overlap. The activation R_{22} will be accepted by *RowBuff* during the next clock cycle, in that it must be preliminary subjected to column overlap.

This process is repeated for the input activation I_{01}, which either fully generates or contributes to the output activations *Rmn*, with $m = 2, \ldots, 4$ and $n = 0, \ldots, 2$. In this step, column overlap definitely provides the final activations R_{02} and R_{12}. Conversely, R_{22} is stored within the *RowBuff* array for row overlap purposes.

The input activation I_{10} belongs to the second row of the *ifmap*. Starting from this point, row overlaps will be also executed. After having generated the activations *Rmn*, with $m = 0, \ldots, 2$ and $n = 2, \ldots, 4$, those having $m = 2$ are accumulated to the temporary results provided two cycles before, which share the same row index and have $n = 0$, 1. The provisional result R_{22} is stored within *ColBuff* to be subjected to its final column overlap during the next cycle. Finally, the input activation I_{11} generates the final results *Rmn*, with $m = 2, \ldots, 4$ and $n = 2, \ldots, 4$.

In order to comply with 3-D TCONV, the $T_{OC} \times T_{IC}$ windows are summed up in a pixel-wise manner (lines 37–41) to generate the provisional T_{OC} *ofmaps*. Obviously, if no parallelism is exploited, no further computations are required (lines 37–38). The process is equally repeated when the subsequent group of T_{IC} *ifmaps* is processed, thus generating a new group of provisional T_{OC} *ofmaps*, which must be accumulated to that generated during the previous cycle. To manage this, an on-chip buffer, namely *outBuff*, is exploited. By a circuital point of view, the latter can be thought as T_{OC} banks of *S* simple dual-port memories, each being $H_I \times W_I$ wide. During the writing phase, *outBuff* receives $T_{OC} \times S \times S$ activations per cycle, while during the read phase it provides the data as a stream of T_{OC} activations per cycle, in order to well comply with the raster-order policy of either a subsequent TCONV layer or an external memory support.

The writing phase works as follows: according to the IOM strategy, the T_{IC} ifmaps are able to generate T_{OC} *ofmaps* with $S \times S$ valid activations per clock cycle. The generic memory bank stores the $S \times S$ results, into the *S* memories. Taking into account that the latter consists of *S* rows of *S* adjacent activations, the *S* memories of the bank store *S* activations in each cell.

To properly access each memory cell, the read phase makes use of a control logic that manages two pointers, namely *buff_idx* and *cell_idx* that indicates, respectively, which memory of the bank and which specific cell are under analysis. Without loss of generality, the example reported in Fig. 5 shows the behavior of *outBuff* during the reading phase. There, $T_{OC}=1$, $S=2$ and $H_I=W_I=2$. In other words, the *outBuff* consists of $T_{OC}=1$ memory bank of $S=2$ memories (i.e., *Memory#0* and *Memory#1*). Each memory has $H_I \times W_I=4$ cells, each able to accommodate $S=2$ activations. The numbers indicated in each cell refer to the spatial position of the activations within the *ofmap* space. The control logic follows that numbering strategy to furnish $T_{OC}=1$ activations per clock cycle. During the first cycle, $buff\_idx=0$ and $cell\_idx=0$. This state is preserved for two cycles, in order to get the first two activations stored within the *Memory#0*. During the third and fourth cycles, $cell\_idx=1$ and the activations R_{02}, R_{03} are read. Afterwards, *cell_idx* is set again to 0, while $buff\_idx=1$. Accordingly, the activations R_{10} and R_{11} are read. Thus, $cell\_idx=1$ to read the activations R_{12} and R_{13}. At this point, in order to read the activations R_{20} and R_{21}, $buff\_idx=0$ and $cell\_idx=2$. And so on for the remaining activations, following a ping-pong way.

3.2 Parametric Analysis

In order to be compliant with real FPGA-based acceleration architectures, the proposed TCONV layer is equipped with the streaming interface of the fourth generation Advanced eXtensible Interface (AXI4-Stream) [31]. To this aim, the *#pragma HLS INTERFACE axis* is used.

As a first set of experiments, the proposed template implements several circuit configurations, in order to examine the trend of both resources utilization and speed throughput when the specific TCONV parameters are varied (i.e., the kernel size K, the stride S and the *ifmaps* sizes H_I, W_I). The characterization is carried out using

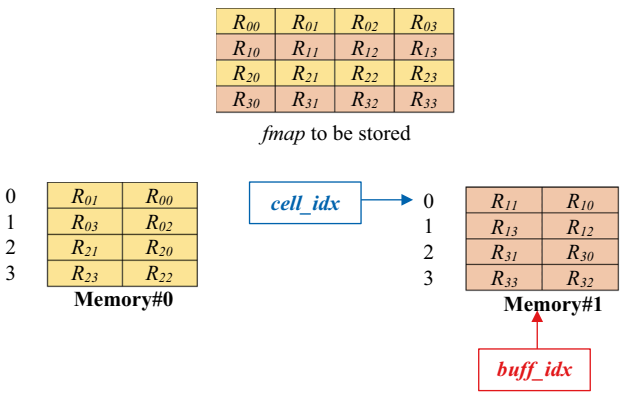


Figure 5 Example of *outBuff* management when $H_I=W_I=2$ and $S=2$.

the Xilinx Vivado Design Suite (v2019.2) and referring to the XC7Z020 device at the 100 MHz clock frequency. The resources utilization is evaluated in terms of Look-Up Tables (LUTs), Flip-Flops (FFs), on-chip Block Random Access Memories (BRAMs) and Digital Signal Processing slices (DSPs). The frames-per-second (*fps*) metric is considered for the evaluation of the speed throughput. Figure 6a-c illustrate the obtained trends. For each plot, the horizontal axis refers to the specific parameter under evaluation, while the vertical axis reports the resources utilization as well as the achieved throughput.

The impact of the stride S is reported in Fig. 6a. The variation of S is typically adopted in super-resolution imaging, which relies on wide filters to process the *ifmaps* [8]. Accordingly, K is set to 9 and left fixed for all the configurations. The other parameters are fixed to $N=8$, $H_I=W_I=32$, $T_{IC}=2$, $T_{OC}=2$. While LUTs, FFs, and DSPs are practically unaffected by S, the quantity of BRAMs grows with the latter. This is justified by the fact that the sizes of the generated *ofmaps* are proportional to S and, in turn, they are temporarily buffered in on-chip memories to comply with 3-D TCONV accumulations. For example, varying S from 2 to 4 leads to a 3.6× increase of memory. As stated in Section 3.1, the on-chip buffers are made able to write data as a stream of $T_{OC} \times S \times S$ activations per cycle. However, in order to satisfy the raster-order policy of contiguous TCONV layers, the stored activations have to be moved out as a stream of T_{OC} values per cycle. The latter directly impacts on the latency that becomes proportional to S. As a result, the higher S, the higher the latency and the lower the frame rate.

The impact of the kernel size K is reported in Fig. 6b. There, while K is varied between 3 and 9, the other parameters are fixed to $N=8$, $H_I=W_I=32$, $S=2$, $T_{IC}=2$, $T_{OC}=2$. The computing resources (i.e., LUTs and DSPs) are strongly influenced by K. This is due to the use of *#pragma HLS PIPELINE* that completely unrolls the computational loops of the IOM to meet the $II=1$ constraint. The higher the filter size, the higher the number of parallel hardware replicas. Conversely, the *fps* is practically independent of K. Indeed, the negligible ~5.1% loss noted when K is moved from 3 to 9 is due to the low amount of extra cycles to load wider 9×9 kernels.

Finally, the impact of the input *fmap* sizes H_I, W_I is reported in Fig. 6c, where $H_I=W_I$ by supposing square *fmaps*, as usually happens in CNNs. The sizes range from 8 to 128, which is usual in image generation. The other parameters are fixed to $N=8$, $K=4$, $S=2$, $T_{IC}=2$, $T_{OC}=2$. As expected, the throughput is strongly influenced by H_I and W_I. Indeed, wider *ifmaps* require more clock cycles to be processed by the TCONV layer. On-chip BRAMs are also influenced by the referred sizes, especially when H_I, $W_I \geq 32$. Indeed, at the parity of S, the wider the *ifmaps*, the wider the *ofmaps* to be temporarily stored on-chip.

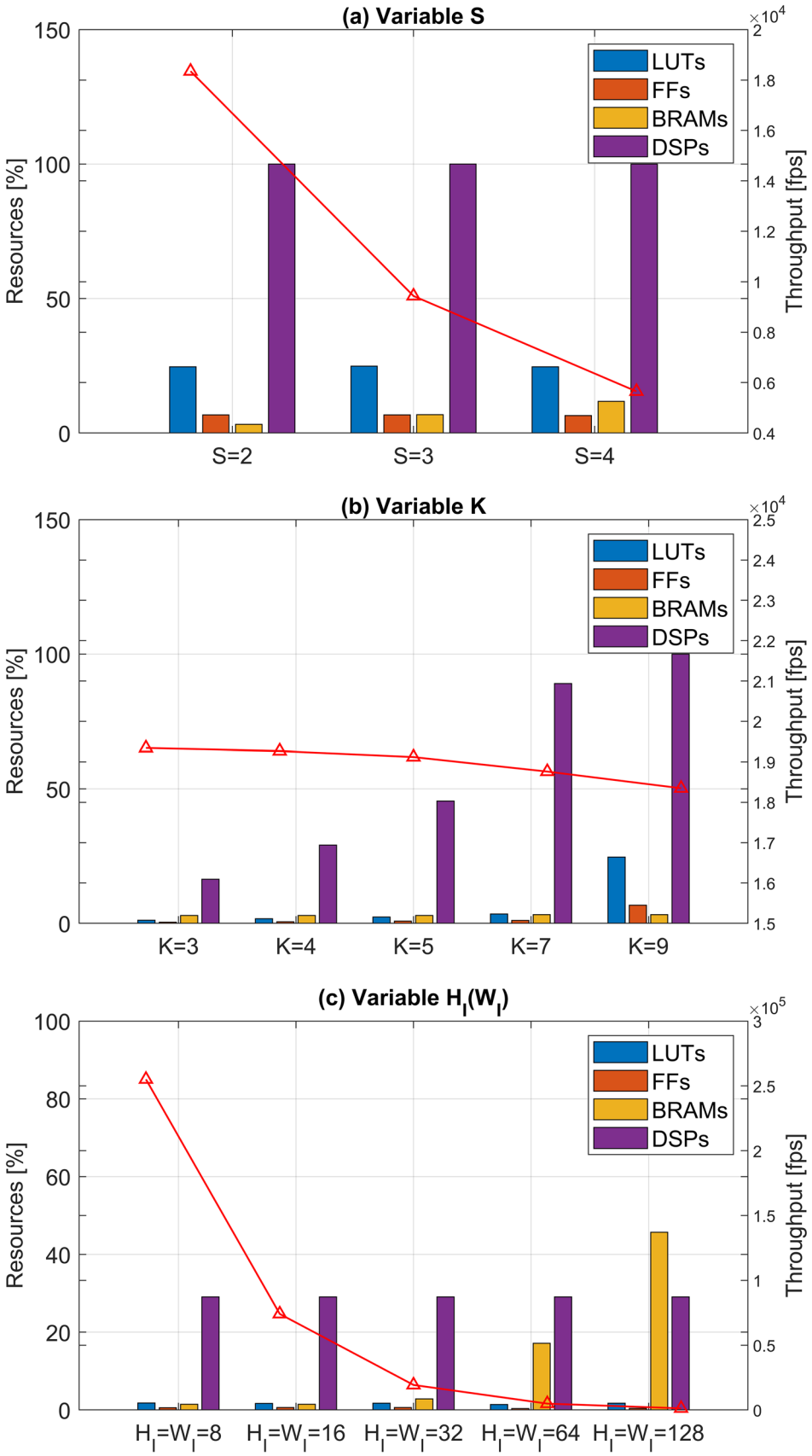


Figure 6 Parametric analysis of the TCONV layer (**a**) varying S, (**b**) varying K, (**c**) varying H_I (W_I).

Table 1 Summary of the TCONV layer parametric analysis.

Fixed Parameters	Variable Parameter	Resources				Throughput [*fps*]
		LUTs	FFs	BRAMs (18 Kb)	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

Table 1 summarizes all the performed experiments, by reporting (a) the parameters of each configuration, (b) the resources utilization (i.e., LUTs, FFs, 18 Kb BRAMs, DSPs), and (c) the throughput in terms of *fps*.

3.3 Comparison with State-of-the-Art Competitors

In this Section, the proposed TCONV layer is compared to several state-of-the-art FPGA-based architectures [23, 25, 27] at a parity of bit-width N, kernel size K and stride S, as well as the used device. Table 2, other than reporting the referred parameters, provides information about (a) the output image sizes ($H_O \times W_O$); (b) the output parallelism, given by the number of images generated in parallel (T_{OC}); (c) the resources utilization (i.e., LUTs, FFs, BRAMs, DSPs used); (d) the clock frequency; (e) the throughput in terms of Giga Outputs per Second (*Gout/s*); (f) the power consumption and the energy efficiency (i.e., the ratio between the throughput and the power). Power consumption was estimated by analyzing the post-implementation results through the power analysis tool available within Vivado.

The circuit having $K=3$ is implemented within the XC7Z020 FPGA device to be compared with the direct competitor [25]. It can be seen that, due to the higher parallelism, the proposed solution exhibits twice the throughput but at the expense of a ~ 36% lower energy efficiency. Indeed, the hybrid computational approach adopted in [25] to replace many TCONVs with simpler averages leads to a power dissipation of only ~ 9 mW and to a DSP slices utilization ~ 5× lower than the proposed architecture.

Table 2 Characterization of the HLS TCONV Layer and state-of-the-art comparisons[a].

	New	New	New	[25]	[27]	[27]	[23]
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
$H_O \times W_O$	64×64	64×64	64×64	NA[b]	64×64	64×64	NA
T_{OC}	8	4	8	NA	2	2	64
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

[a]Best performance in **bold** at the parity of device and *K, S*

[b]*NA* Not Available

The circuit having $K=5$ is implemented within both the XC7Z020 and the XC7Z100 FPGA devices and directly compared to the reconfigurable architectures proposed in [27]. Both the accelerators adopt the IOM strategy for TCONV computations. However, in [27] VHDL templates are used to improve speed performances. When the XC7Z020 device is referred to, the novel architecture exhibits a ~ 25% higher throughput, due to the doubled parallelism, even running at a ~ 37.5% lower frequency. Results obtained for the XC7Z100 part show that the proposed design is ~ 1.14× more energy-efficient than [27], while running ~ 1.5× slower. Finally, when compared to the accelerator Uni-OPU [23], the proposed solution is only ~ 5.8% less energy-efficient, but it uses ~ 89.8%, ~ 95.8%, ~ 59.7%, ~ 74% less LUTs, FFs, DSPs and BRAMs, respectively.

Finally, in order to get a more insightful understanding about the impact of power over the time, we also retrieved the energy values for all the implemented circuits. Considering that all of them take the same number of clock cycles to complete the task, the reported energy depends on the clock frequency and the power consumption. The XC7Z020 implementation with $K=3$ dissipates ~ 19 μJ at 125 MHz. At the same frequency, the $K=5$ configuration shows a ~ 24.2% increment due to the slightly higher power (because of the higher usage of computing resources to manage wider filters). As expected, the high-end XC7Z100 implementation led to the highest contribution of 62.8 μJ to meet the doubled parallelism. When comparing the XC7Z020 and the XC7Z100 implementations at $K=5$, while the power dissipation shows a 4.25× increase, the energy ratio is only 2.7×. This because the XC7Z100 implementation benefits from a 50% higher clock frequency.

4 Characterization of a Decoder Through Design-Space Exploration

This Section evaluates the suitability of the TCONV layer model introduced in Section 3 to be accommodated within dataflow architectures, consisting of stacked layers that exchange informative content on-chip, by reducing the off-chip memory accesses to send and retrieve the intermediate results. The evaluation has a twofold aim: (1) examine the suitability of careful high-level synthesis, through the effective use of *#pragmas*; (2) determine the impact of the bit-width N, and parallelism configurations, over the implemented architectures.

4.1 The HLS Decoder Template

The top-level model of the TCNN for image decompression, namely decoder, is reported in Listing 3. Two TCONV layers compose the network, with the former also equipped with the Rectified Linear Unit (ReLU) non-linearity [32]. The first layer is supplied by four 8×8 *ifmaps*, and 16 filters of $4\times3\times3$ weights. The stride $S=2$. Accordingly, sixteen 16×16 *fmaps* are generated and provided to the second layer that, in turn, uses 1 filter of $16\times3\times3$ weights. Finally, a 32×32 *ofmap* is provided. The *#pragma HLS DATAFLOW* is adopted to infer task-level parallelism, thus allowing the overlap of the computations of both the layers.

Listing 3 The pseudocode of the TCNN for image decompression

```
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: TCONV Layer + ReLU;
5: TCONV Layer;
```

4.2 Evaluation of *#pragmas*, Bit-Width and Parallelism

To evaluate the effectiveness of *#pragmas*, as well as the influence of bit-width and parallelism, an extensive design-space exploration is carried out, using the XC7Z020 FPGA device at $f = 100$ MHz. Two types of HLS designs are considered:

- Baseline designs, containing the minimum number of *#pragmas* to ensure the correct behavior of the synthesized circuits.
- Optimized designs, containing additional *#pragmas*, to allow the circuits to meet hardware-oriented features, including pipelining and parallelism.

Table 3 summarizes the used *#pragmas*, by clarifying their behavior in hardware, and taking into account both the baseline designs and the optimized counterparts.

Results in terms of resources utilization are shown in Fig. 7a–d. Each plot refers to a specific type of resource (i.e., LUTs, DSPs, FFs, BRAMs, respectively). The labels reported in the *Configurations* axis must be interpreted as (*type of implementation-*T_{IC},T_{OC}), where type of implementation can be '*base*' for the baseline designs and '*opt*' for the optimized versions. Bars of different colors refer to different bit-widths.

LUTs in Fig. 7a are mainly exploited for computations and control. To understand the impact of *#pragmas*, we consider the configurations having $(T_{IC},T_{OC}) = (2,4)$ as an example. The optimized versions use more resources. This is justified by the use of *#pragma HLS PIPELINE* that inherits the *#pragma HLS UNROLL* to allow loops bodies to perform the computations in parallel. For instance, when $N = 6$, the optimized version adopts ~ 2.5× more LUTs.

DSPs are used, in conjunction with LUTs, for computation purposes. Accordingly, the optimized versions, relying in loop unrolling and pipelining, ask for more DSPs, as highlighted in Fig. 7b. For example, fixed $N = 8$, the configuration (*opt-2,2*) requires 14.5× more DSPs than the configuration (*base-2,2*). As a further consideration, it is worth noting that N differently affects DSPs and LUTs utilization trends. As an example, taking into account the optimized designs only, while LUTs utilization progressively increases varying N from 8 to 6, for DSPs a downward trend occurs. Indeed, while the amount of occupied LUTs grow by a 2.1× factor, the amount of utilized DSPs decrease up to 95%. Below 6 bits, the DSPs utilization is steady, while the LUTs utilization progressively decreases. This means that, while the synthesizer infers multiplications and additions using DSPs predominantly in the range 8–6 bits, it relies on LUTs for lower bit-widths.

Figure 7c shows that, given that FFs mainly equip with pipelining the combinatorial paths, their utilization follows the LUTs trend. Conversely, BRAMs are exploited to buffer weights on-chip and to temporary store *ofmaps*

Table 3 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 $T_{OC} \times S \times S$ 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 ($II = 1$) the *ifmaps*' activations. Used to manage the timing of *ofmaps*' activations to be read from the output buffer.
HLS UNROLL It transforms loops by creating several copies of the body to infer parallelism.	**Not Used**	Used to unroll loops for *ifmaps* acquisition, IOM computations and buffering.
HLS INTERFACE It specifies which interfaces must be used by I/O ports.	Used to equip both input and output ports of the TCONV layer 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 timing of data 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 [33].	

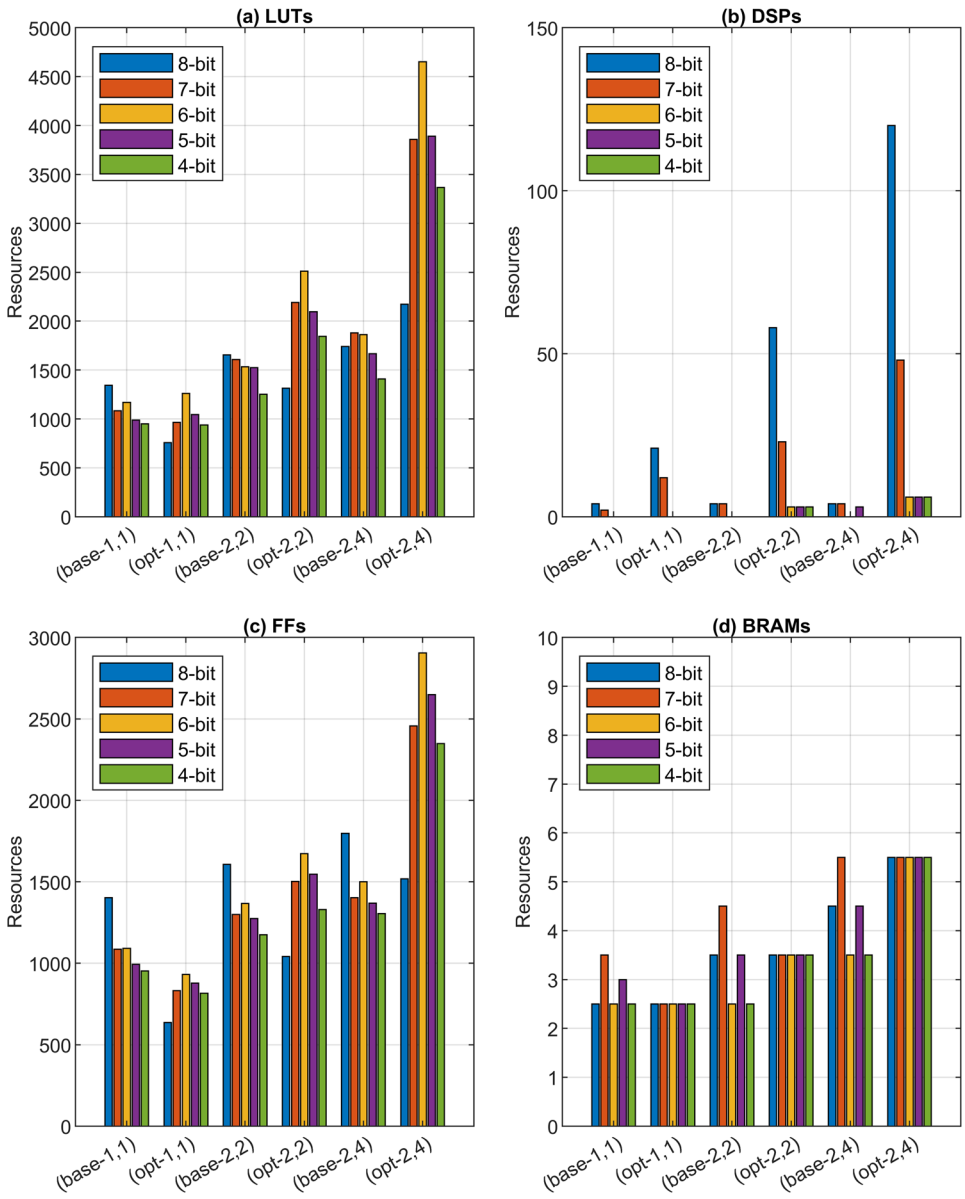


Figure 7 Resources utilization of different configuration of the TCONV-based decoder: **a** LUTs, **b** DSPs, **c** FFs, **d** BRAMs.

before being delivered either to the subsequent TCONV layer or as final outputs. While the optimized configurations make use of proper *#pragmas* to instruct the synthesizer to split and place memory arrays into on-chip RAMs (i.e., *#pragma HLS ARRAY_PARTITION* and *#pragma HLS RESOURCE*), the baseline designs completely leverage the synthesizer to arrange data storage. Let us compare the configurations (*base-1,1*) and (*opt-1,1*) at various N. As visible in Fig. 7d, at a fixed configuration in terms of input and output parallelism, the baseline designs have an irregular trend, whereas the optimized configurations have a constant trend, meaning that they use the same number of BRAMs for each bit-width. This is due to the above mentioned *#pragmas* that instruct the synthesizer to leave BRAMs implementation independent of the bit-width. Finally, as expected, for both the baseline and the optimized configurations, higher (T_{IC},T_{OC}) pairs mean higher memory requirements.

Figure 8 illustrates the frame rate variations versus N by means of a heat map. The optimized designs improve considerably the performance of the decoder, in a range of 2 to 3 orders of magnitude. While a higher parallelism obviously leads to a higher throughput, the latter is independent of the bit-width. Indeed, the *#pragma HLS PIPELINE* (thus, *#pragma HLS UNROLL*) makes the circuit able to perform parallel computations independently of the data word-length.

Finally, in order to provide a trade-off picture of the whole set of experiments, Fig. 9 plots the average resources utilization versus the throughput. The average

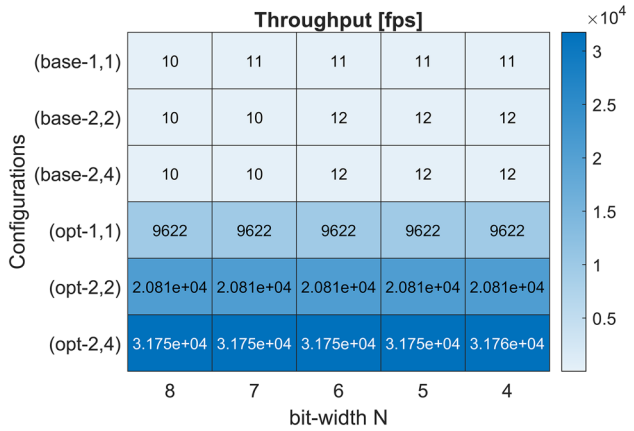


Figure 8 Throughput trend of different configurations of the TCONV-based decoder.

resources utilization is the average of the percentages of each type of resource (i.e., LUTs utilization, FFs utilization, BRAMs utilization, DSPs utilization). For the sake of clear visualization, both the axes use the logarithmic scale. Each point is labeled to represent a specific configuration: indicating the design type (i.e., baseline as '*b*', optimized as '*o*'), the bit-width N and the T_{OC} factor. For example, *o62* represents the optimized design with $N=6$ and output parallelism factor $T_{OC}=2$. The green points refer to the configurations that satisfy the minimum accuracy threshold for both MNIST and Fashion-MNIST datasets, while red points fail in meeting these accuracy thresholds, based on our previous study [17].

Two main regions can be identified within the plot of Fig. 9: on the left, the baseline designs, whereas on the right the optimized counterparts. For what concerning the *fps*, a proper usage of *#pragmas* to infer parallelism leads to an overall 3 orders of magnitude improvement. However, higher throughput means higher resources requirements. Despite this, even considering the worst-case configuration *o84*, the average utilization is limited to 16%. Figure 10 magnifies the baseline designs points. While the *fps* does not significantly changes, a more noticeable variation occurs in terms of the average percentage of resources utilization. As expected, the higher the data bit-width N the higher the resources occupation. This because wider bit-widths reflect on wider computing units (i.e., multipliers and adders), thus increasing the quantity of LUTs and FFs. Furthermore, as expected, higher T_{OC} lead to higher resources occupation. Finally, 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.

5 Characterization of the Generator of the DCGAN Architecture

In order to further investigate the suitability of the proposed TCONV layer for dataflow architectures, we consider the DCGAN network. Specifically, considering the quality results over the MNIST dataset [34], we propose an FPGA accelerator dealing with 5-bit weights and activations. The chosen quantization ensures all the weights

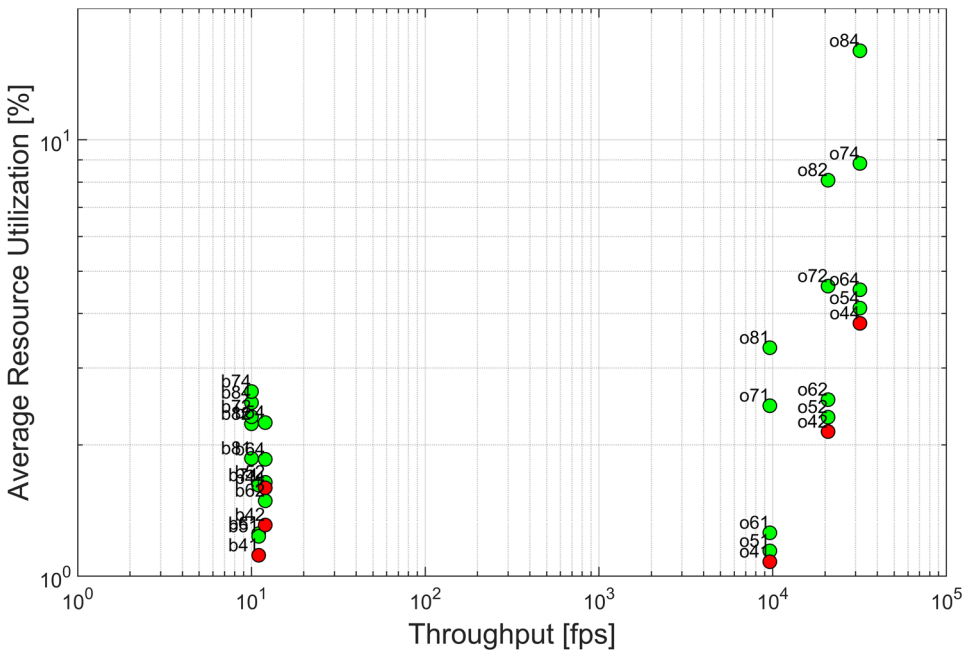


Figure 9 Trade-off analysis of different configurations of the TCONV-based decoder. Each point represents a specific configuration: the first letter indicates the design type (i.e., baseline configuration as 'b', optimized configuration as 'o'); the second value is the bit-width N; the third value is the output parallelism (i.e., the T_{OC} factor).

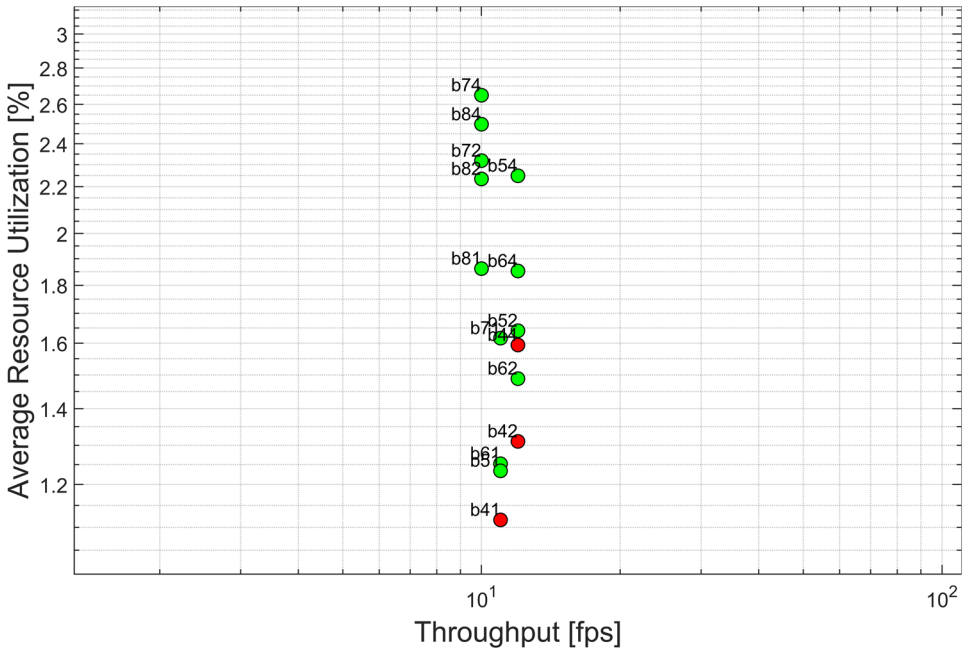


Figure 10 Detail of trade-off analysis of the baseline configurations TCONV-based decoder. Each point represents a specific configuration: the first letter indicates the design type (i.e., baseline configuration 'b'); the second value is the bit-width N; the third value is the output parallelism (i.e., the T_{OC} factor).

to be preliminary stored on-chip and the off-chip memory accesses to retrieve data to be reduced.

5.1 The HLS DCGAN Template

Listing 4 shows the top-level pseudocode of the C++ model. The first *Project and Reshape* layer is supplied by a $1 \times 1 \times 100$ activations and $256 \times 100 \times 4 \times 4$ weights, and performs TCONVs having $K = 4$ and $S = 1$. The resulting 256 4×4 *fmaps* are supplied to the second layer, that executes TCONVs having $K = 4$ and $S = 2$. The same for the third and the fourth layers, which progressively up-sample *fmaps* from 8×8 to 32×32. The second and the third layers are also equipped with ReLU non-linearity [32].

Listing 4 The pseudocode of the generator of the DCGAN model

```
Input: latent vector inStream
Output: generated image outStream
#pragma HLS INTERFACE axis port=inStream
#pragma HLS INTERFACE axis port=outStream
#pragma HLS DATAFLOW
Project and Reshape;
DataBuffer;
TCONV Layer + ReLU;
DataBuffer;
TCONV Layer + ReLU;
TCONV Layer;
```

With respect to the decoder analyzed in Section 4, the DCGAN requires reusing the *fmaps* to manage internal layers with more than one 3-D filter. Indeed, all the 3-D filters need the same activations to generate as many *ofmaps*. However, due to the streaming behavior of the architecture, the activations provided by the current layer are generated once and consumed by the next layer as soon as possible. As a consequence, a buffer at the interface is mandatory. This is the meaning of the *DataBuffer* function reported in Listing 4 and placed between the Layers 1–2 (line 5) and 2–3 (line 7). No extra buffering is needed between the Layers 3 and 4, in that the latter uses just one 3-D filter to provide the output image.

5.2 Characterization and State-of-the-Art Comparisons

The architecture is implemented within the XC7Z045 FPGA and characterized over different parallelism configurations. Table 4 reports the resources utilization, the achieved clock frequency, the throughput (*fps*), the power dissipation, the energy dissipation, and the energy efficiency. Power was estimated considering the post-implementation results and using the power analysis tool. Energy was computed considering the power results and the latency to process one image through the entire DCGAN architecture. The configurations array refers to the parallelism factor T_{OC} of each layer. For example, the configuration (2,2,2,1) indicates that the first, the second, and the third layers exhibit an output parallelism $T_{OC}=2$, while the last layer has $T_{OC}=1$.

Obviously, the higher the overall parallelism, the better the throughput. An improvement of ~4.6× is achieved with the most parallelized configuration (2,4,2,1) compared with the sequential (1,1,1,1) configuration, whilst power dissipation grows from 0.34W to 0.74W due to increased BRAM use. In addition, $N=5$ practically nullifies the use of DSPs for computations, thus removing a not-negligible source of power consumption.

Parallelism positively impacts the energy dissipation. Indeed, a ~2.15× reduction is evident when moving from the baseline (1,1,1,1) configuration to the most parallelized counterpart (2,4,2,1). Considering the slight increase in power, the referred result is mainly due to the higher speed performance (i.e., the *fps*).

Finally, the configuration (2,4,2,1) is compared to some state-of-the-art counterparts, as reported in Table 5. With respect to the architecture presented in [21], the proposed accelerator shows a 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 new design takes just 1 DSP. It is also worth underlining that the counterpart [21] takes advantage of the high-performance Alveo U200 device: it achieves the 300 MHz clock frequency, thus improving the *fps*.

In comparison with [24], at a parity of the FPGA device used and of the clock frequency, the architecture presented here is ~29.7% faster and exhibits an energy efficiency more than 10× higher. This is the direct consequence of the extra logic exploited in [24] to meet the transformations steps of the Winograd-based TCONV algorithm, as well as the high level of parallelism.

Finally, Table 5 shows that the 16-bit DCGAN accelerator [28] accommodated within the high-end XC7VX690T FPGA device reaches the best throughput but at expense of a considerable amount of resources: it occupies ~13.26×, ~23.47×, ~3.56× more LUTs, FFs, on-chip BRAMs, respectively, than the new design.

Table 4 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 [mJ]	6.46	3.01	3.03
Energy Efficiency [*fps*/W]	152.94	332.20	324.32

Table 5 State-of-the-art comparisons of FPGA-based DCGAN models.

	New	[21]	[24]	[28]
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[a]	826
Power [W]	0.74	9	5.8	-
Energy [mJ]	3.03	22.50	31.32	-
***fps*/W**	324.32	44.44	31.89	-

[a]Retrieved from the GOPS reported in the paper

6 Conclusions

This paper presented the design of an HLS-based TCONV layer, based on the IOM strategy, and suitable for dataflow-based neural network architectures. The latter consist of stacked layers that exchange data internally, by reducing the data movement from/to external memory resources. The accelerator was conceived at the C++ abstraction, using a parametric template to be adapted at different configurations using proper bit-widths, kernel sizes, strides, image sizes, parallelism factors. For purposes of characterization, the proposed engine was preliminarily examined as a standalone unit and then integrated into two neural networks: a decoder for image decompression, and the generator of the DCGAN network.

The standalone TCONV layer showed competitiveness when compared the state-of-the-art, being able to provide up to 6.25 *Gout/s* and dissipating only 1.53 W, using ~ 11.7k LUTs resources and 4.52 Mb of on-chip memory.

When used within the decoder architecture, a systematic design-space exploration was conducted to investigate the positive influence of *#pragmas* to infer hardware optimizations, including pipelining and parallelism. The evaluation ranges in the interval 8–4 bits and showed an improvement of 3 orders of magnitude in *fps* throughput, with an average resources utilization of at most 16%. Finally, the characterization of the DCGAN at 5 bits highlighted a noticeable throughput of 240 *fps*, with an energy efficiency of 324.32 *fps*/W, which outperforms the state-of-the-art counterparts by a factor of at least ~ 7.3×.

Funding Open access funding provided by Università della Calabria within the CRUI-CARE Agreement. This work was supported by POR Calabria FSE/FESR 2014-2020 – International Mobility of PhD students and research grants/type A Researchers" – Actions 10.5.6 and 10.5.12 actuated by Regione Calabria, Italy, and by Ministero dell'Università e della Ricerca under ICSC National Research Centre for High Performance Computing, Big Data and Quantum Computing within the Next Generation EU program.

Data Availability Statement The dataset generated during the current study is available from the authors on request.

Declarations

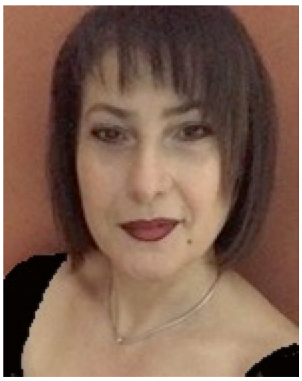
Competing Interests The authors declare no conflicts of interest.

References

1. Voulodimos, A., Doulamis, N., Doulamis, A., & Protopapadakis, E. (2018). Deep Learning For Computer Vision: A Brief Review. *Computational Intelligence and Neuroscience, 2018*, 1–13. https://doi.org/10.1155/2018/7068349
2. Nassif, A. B., Shahin, I., Attili, I., Azzeh, M., & Shaalan, K. (2019). Speech recognition using deep neural networks: A systematic review. *IEEE Access, 7*, 19143–19165. https://doi.org/10.1109/ACCESS.2019.2896880
3. Wang, Z., & Majewicz Fey, A. (2018). Deep learning with convolutional neural network for objective skill evaluation in robot-assisted surgery. *International Journal of Computer Assisted Radiology and Surgery, 13*(12), 1959–1970. https://doi.org/10.1007/s11548-018-1860-1
4. Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., & Bharath, A. A. (2018). Generative adversarial networks: An overview. *IEEE Signal Processing Magazine, 35*(1), 53–65. https://doi.org/10.1109/MSP.2017.2765202
5. Kumar, M., & Sharma, H. K. (2023). A GAN-Based Model of Deepfake Detection in Social Media. *Procedia Computer Science, 218*, 2153–2162. https://doi.org/10.1016/j.procs.2023.01.191
6. Im, D., Han, D., Choi, S., Kang, S., & Yoo, H. J. (2020). DT-CNN: An energy-efficient dilated and transposed convolutional neural network processor for region of interest based image segmentation. *IEEE Transactions on Circuits and Systems I: Regular Papers, 67*(10), 3471–3483. https://doi.org/10.1109/TCSI.2020.2991189
7. Gu, Z., Cheng, J., Fu, H., Zhou, K., Hao, H., Zhao, Y., Zhang, T., Gao, S., & Liu, J. (2019). Ce-net: Context encoder network for 2d medical image segmentation. *IEEE Transactions on Medical Imaging, 38*(10), 2281–2292. https://doi.org/10.1109/TMI.2019.2903562
8. Dong, C., Loy, C. C., & Tang, X. (2016). Accelerating the super-resolution convolutional neural network. In *European Conference on Computer Vision (ECCV)* (pp. 391–407). Springer, Cham. https://doi.org/10.1007/978-3-319-46475-6_25
9. Spagnolo, F., Corsonello, P., Frustaci, F., & Perri, S. (2023). Design of a Low-power Super-Resolution Architecture for Virtual Reality Wearable Devices. *IEEE Sensors Journal, 23*(8), 9009–9016. https://doi.org/10.1109/JSEN.2023.3256524
10. Chang, J. W., Kang, K. W., & Kang, S. J. (2020). An energy-efficient FPGA-based deconvolutional neural networks accelerator for single image super-resolution. *IEEE Transactions on Circuits and Systems for Video Technology, 30*(1), 281–295. https://doi.org/10.1109/TCSVT.2018.2888898
11. Nurvitadhi, E., Venkatesh, G., Sim, J., Marr, D., Huang, R., Ong Gee Hock, J., Liew, Y. T., Srivatsan, K., Moss, D., Subhaschandra, S., & Boudoukh, G. (2017). 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 (FPGA)* (pp. 5–14). ACM. https://doi.org/10.1145/3020078.3021740
12. Yazdanbakhsh, A., Brzozowski, M., Khaleghi, B., Ghodrati, S., Samadi, K., Kim, N. S., & Esmaeilzadeh, H. (2018). FlexiGAN: An end-to-end solution for FPGA acceleration of generative adversarial networks. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (pp. 65–72). IEEE. https://doi.org/10.1109/FCCM.2018.00019

13. Sestito, C., Spagnolo, F., & Perri, S. (2021). Design of Flexible Hardware Accelerators for Image Convolutions and Transposed Convolutions. *Journal of Imaging*, *7*(10):210, 1–16. https://doi.org/10.3390/jimaging7100210
14. Zhang, X., Das, S., Neopane, O., & Kreutz-Delgado, K. (2017). A Design Methodology for Efficient Implementation of Deconvolutional Neural Networks on an FPGA. *arXiv preprint* arXiv:1705.02583.
15. Blott, M., Preußer, T. B., Fraser, N. J., Gambardella, G., & O'brien, K., Umuroglu, Y., Leeser, M., & Vissers, K. (2018). FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems (TRETS), 11*(3), 1–23. https://doi.org/10.1145/3242897
16. Stewart, R., Nowlan, A., Bacchus, P., Ducasse, Q., & Komendantskaya, E. (2021). Optimising hardware accelerated neural networks with quantisation and a knowledge distillation evolutionary algorithm. *Electronics*, *10*(4):396, 1–21. https://doi.org/10.3390/electronics10040396
17. Sestito, C., Perri, S., & Stewart, R. (2022). Design-Space Exploration of Quantized Transposed Convolutional Neural Networks for FPGA-based Systems-on-Chip. *In 2022 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)* (pp. 1–6). IEEE. https://doi.org/10.1109/DASC/PiCom/CBDCom/Cy55231.2022.9927825
18. LeCun, Y., Cortes, C., & Burges, C. J. (1998). *The MNIST database of handwritten digits*. Retrieved from http://yann.lecun.com/exdb/mnist/
19. Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. arXiv preprint arXiv:1708.07747. https://doi.org/10.48550/arXiv.1708.07747
20. Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. arXiv preprint arXiv:1511.06434. https://doi.org/10.48550/arXiv.1511.06434
21. Meng, Y., Kuppannagari, S., Kannan, R., & Prasanna, V. (2021, December). 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)* (pp. 81–90). IEEE. https://doi.org/10.1109/HiPC53243.2021.00022
22. Mao, W., Lin, J., & Wang, Z. (2020). F-DNA: Fast convolution architecture for deconvolutional network acceleration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *28*(8), 1867–1880. https://doi.org/10.1109/TVLSI.2020.3000519
23. Yu, Y., Zhao, T., Wang, M., Wang, K., & He, L. (2020). Uni-OPU: An FPGA-based uniform accelerator for convolutional and transposed convolutional networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *28*(7), 1545–1556. https://doi.org/10.1109/TVLSI.2020.2995741
24. Di, X., Yang, H. G., Jia, Y., Huang, Z., & Mao, N. (2020). Exploring efficient acceleration architecture for Winograd-transformed transposed convolution of GANs on FPGAs. *Electronics*, *9*(2):286, 1–21. https://doi.org/10.3390/electronics9020286
25. Marrazzo, E., Spagnolo, F., & Perri, S. (2022). Runtime Reconfigurable Hardware Accelerator for Energy-Efficient Transposed Convolutions. In *2022 17th Conference on Ph. D Research in Microelectronics and Electronics (PRIME)* (pp. 141–144). IEEE. https://doi.org/10.1109/PRIME55000.2022.9816800
26. Yan, J., Yin, S., Tu, F., Liu, L., & Wei, S. (2018). GNA: Reconfigurable and efficient architecture for generative network acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 37*(11), 2519–2529. https://doi.org/10.1109/TCAD.2018.2857258
27. Perri, S., Sestito, C., Spagnolo, F., & Corsonello, P. (2020). Efficient deconvolution architecture for heterogeneous systems-on-chip. *Journal of Imaging*, *6*(9):85, 1–17. https://doi.org/10.3390/jimaging6090085
28. Wang, D., Shen, J., Wen, M., & Zhang, C. (2019). Efficient implementation of 2D and 3D sparse deconvolutional neural networks with a uniform architecture on FPGAs. *Electronics*, *8*(7):803, 1–13. https://doi.org/10.3390/electronics8070803
29. Lavin, A., & Gray, S. (2016). Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 4013–4021). IEEE. https://doi.org/10.1109/CVPR.2016.435
30. Liu, S., Fan, H., Niu, X., Ng, H. C., Chu, Y., & Luk, W. (2018). Optimizing CNN-based segmentation with deeply customized convolutional and deconvolutional architectures on FPGA. *ACM Transactions on Reconfigurable Technology and Systems (TRETS), 11*(3), 1–22. https://doi.org/10.1145/3242900
31. ARM. (2012). *AMBA 4 AXI4, AXI4-Lite, and AXI4-Stream Protocol Assertions User Guide*. Retrieved from https://developer.arm.com/documentation/dui0534/b/
32. Hara, K., Saito, D., & Shouno, H. (2015). Analysis of function of rectified linear unit used in deep learning. In *2015 International Joint Conference on Neural Networks (IJCNN)* (pp. 1–8). IEEE. https://doi.org/10.1109/IJCNN.2015.7280578
33. AMD Xilinx. (2020). *Vivado Design Suite User Guide: High-Level Synthesis. UG902 (v2019.2)*. Retrieved from https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2019_2/ug902-vivado-high-level-synthesis.pdf
34. Sestito, C., Perri, S., & Stewart, R. (2022). Accuracy Evaluation of Transposed Convolution-Based Quantized Neural Networks. In *2022 International Joint Conference on Neural Networks (IJCNN)* (pp. 1–8). IEEE. https://doi.org/10.1109/IJCNN55064.2022.9892671

Cristian Sestito was born in Chiaravalle Centrale, Italy, on February 7, 1994. He received his Master Degree in Electronic Engineering and the Ph.D. degree in Information and Communication Technologies from the University of Calabria, Italy, in 2019 and 2023, respectively. In 2021-2022, he was also a Visiting Scholar at Heriot-Watt University, Edinburgh. Currently, he is a Research Associate in Embedded System Design at Centre for Electronics Frontiers, The University of Edinburgh. His research interests include the design of efficient FPGA-based architectures for image processing tasks, the hardware acceleration of Deep Learning applications and the investigation of compression techniques for Deep Learning, such as data quantization.



Stefania Perri was born in Cosenza, Italy, on April 6, 1971. She received her Master degree in Computer Science Engineering from the University of Calabria, Italy, in 1996 and the Ph.D. degree in Electronics Engineering from the University Mediterranea of Reggio Calabria, Italy, in 2000. In 1996, she joined the Department of Electronics, Computer Sciences and Systems of the University of Calabria as Researcher Associate. In 2002, she was appointed as Assistant Professor of Electronics with the Department of Electronics, Computer Science and Systems of the University of Calabria, Italy. In the summer 2004, she was a Visiting Researcher at the Department of Electrical and Computer Engineering of the University of Rochester, NY, USA, where in 2005 she was appointed as Adjunct Assistant Professor for four years. In 2010, she was appointed as Associate Professor of Electronics at the Department of Electronics, Computer Sciences and Systems of the University of Calabria. In 2017, she joined the Department of Mechanical, Energy and Management Engineering of the University of Calabria. Her current research interests include QCA-based circuits, high-performance embedded systems, low-power design, VLSI circuits for image processing and multimedia, reconfigurable computing, and VLSI design. She is coauthor of more than 140 technical papers and holds two patents in these fields. She serves on technical committees of several VLSI conferences and as a peer reviewer for several VLSI journals. She is an Associate Editor of the Journal of Low PowerElectronics and Applications and Sensors.

Robert Stewart is an Associate Professor at Heriot-Watt University, Edinburgh. He received his MEng in Software Engineering in 2010, and his PhD in Computer Science in 2013, both at Heriot-Watt University. He was a post-doctoral researcher then Research Fellow from 2013, before being appointed to a lectureship position in 2018. His research interests are at the intersection between high level programming models and low level system architectures. This includes generating deep learning hardware accelerators from high-level specifications, fault tolerant runtime systems for scalable functional languages, DSLs for FPGAs, and recently, designing processor architectures specialised for parallel functional languages. He is the co-author of more than 40 peer reviewed papers. He serves on the Programme Committee and as a reviewer for multiple programming language and parallel computing conferences and journals.