University of Calabria

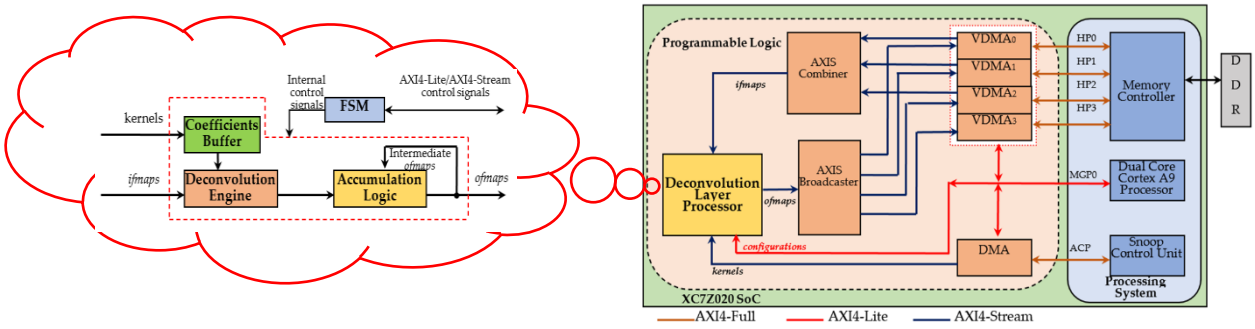
Department of Informatics, Modeling,
Electronics and System Engineering

Rende, Cosenza, Italy

XILINX
| UNIVERSITY PROGRAM

Project report (extract)

An Efficient FPGA-based Deconvolver for Deep Learning Applications



Cristian Sestito, *PhD student*

Supervisor: Stefania Perri, *Associate Professor*

Team number: **xohw20-191**

PhD Category

YouTube link: https://youtu.be/CCn8W2tfqO8

Contents

1. Introduction

Nowadays, Convolutional Neural Networks (CNNs) are proving effective in different image processing tasks. In particular, beside the mainstream convolutional layers (CONVs) able to interpolate the main features of inputs, deconvolutional layers (DECONVs) are gaining interest in different scenarios, such as semantic image segmentation [1], super-resolution images [2], object generation [3], by exploiting the deconvolution algorithm. DECONVs read a set of N_C $H \times W$ input feature maps (*ifmaps*) and N_F $N_C \times K \times K$ kernels and generate N_F output feature maps (*ofmaps*). Deconvolutions are performed between each *ifmap* and each $K \times K$ kernel. Finally, the N_C intermediate *ofmaps* provide the generic *ofmap* by means of a pixel-wise addition. This process is equally repeated N_F times.

Deconvolutions, better known as *transposed convolutions* or *fractionally-strided convolutions* [4] aim at extrapolating new features from inputs, by means of an upsampling process. Basically, they provide high-resolution images from low-resolution ones. Despite the two process seem to be diametrically opposed, they can be managed similarly. Transposed convolutions, indeed, are direct convolutions whose workspace is a wider *ifmap*, got by using zero padding and striding over the original input. For the sake of clarity, let us consider the transposed convolution between a 3×3 *ifmap* and a 3×3 kernel. Let also suppose a stride $S = 2$ and a padding $P = 1$. According to [4], the extended *ifmap* is characterized by pixels spaced from each other by $S - 1$ locations and zeroed borders, whose size is $K–P–1$. The resulting high-resolution *ofmap* sizes are $S \times (H–1) + K – 2P$ and $S \times (W–1) + K – 2P$. In such a case, the 3×3 *ifmap* provides a 5×5 output, as depicted in **Figure 1** [4].

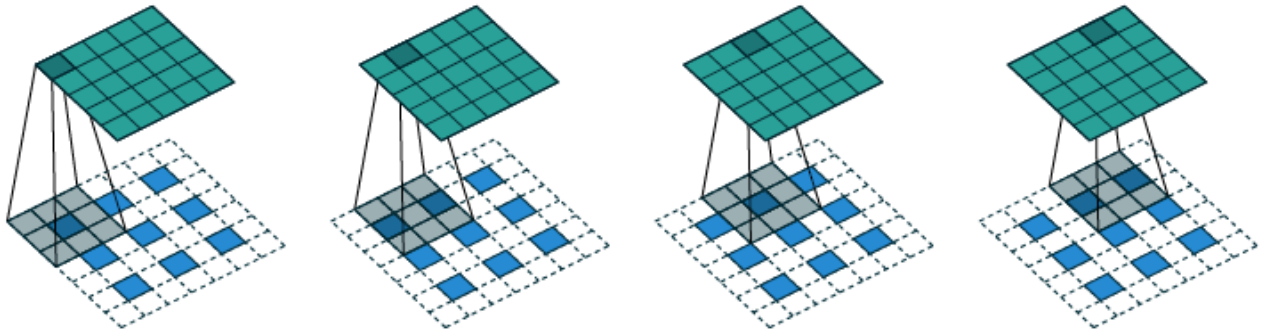


Figure 1: An example of transposed convolution

However, this strategy requires more operations than those needed for direct convolutions. Since Deep Learning applications generally exhibit real-time performances, the above discussed process could be harmful when executed by Central Processing Units (CPUs). Although Graphics Processing Units (GPUs) efficiently address this issue in terms of performances, they are power-hungry devices and, hence, not suitable when dealing with smart nodes, as happens in Internet of Things (IoT) applications. Conversely, Field-Programmable Gate Array (FPGAs) are promising devices able to

reach a valuable trade-off between performances and power consumption and intensively exploited for accelerating CNNs [5]. Even better, FPGA-based Systems-on-Chip (SoCs) can effectively merge the power of a hard processor and a reconfigurable logic to implement advanced embedded systems. Such devices adopt fast Digital Signal Processing (DSP) slices to perform convolutions. Each DSPs, in fact, implements a Multiply-Accumulation (MAC), that is the essence of the discussed algorithm. Nevertheless, the traditional transposed convolution is inefficient since it requires too many useless MACs due to the huge padding and striding applied and, therefore, exhibiting unbalanced workloads among the DSPs.

To address this issue, a novel strategy has been recently proposed [6]. It completely revisits the deconvolution process by exploiting a 4-step algorithm, as follows:

(1) An input pixel, with coordinates (h,w), is multiplied by the respective $K \times K$ kernel and the resulting block is arranged within a $K \times K$ area of the *ofmap*, which starts at the position ($h \times S$,$w \times S$).

(2) Resulting blocks, belonging to adjacent input pixels, can overlap. If it occurs, the overlapped elements must be summed up. For each couple of pixels, the size of overlapped rows/columns is equal to $K–S$.

(3) Steps (1) and (2) are repeated for all the pixels.

(4) The borders of the resulting *ofmap* can be cropped of size P.

It is worth pointing out that this strategy provides $S \times S$ results per cycle, rather than just one as happens in typical convolutions. **Figure 2** shows an example, when $H = W = 3$, $K = 3$, $S = 2$, $P = 1$.

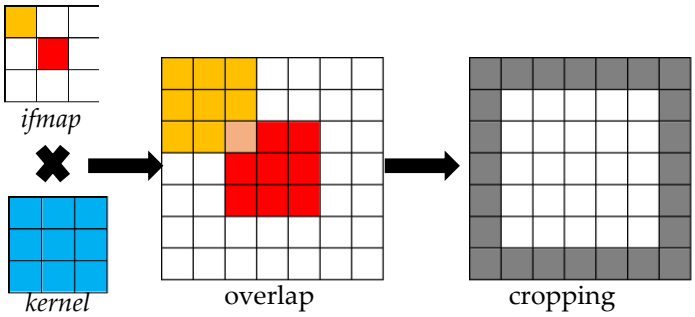


Figure 2: An example of the optimized deconvolution algorithm

This work presents a custom hardware accelerator able to perform efficient deconvolutions by exploiting the latter algorithm. The proposed Deconvolution Layer Processor (DLP) also complies with the Advanced eXtensible Interface (AXI) [7] protocol to be easily integrated within virtually any heterogenous FPGA-based SoC. As a case study, the novel engine has been employed to accelerate Deep Convolutional Generative Adversarial Networks (DCGANs) [8] by using the Xilinx Zynq XC7Z020 device available within the prototyping ZedBoard development kit [9].

2. Design

This section describes the Deconvolution Layer Processor (DLP) and how it was integrated within the complete embedded system. A concise description of the control software is also provided.

2.1. Hardware

The DLP, whose top-level architecture is depicted in **Figure 3**, consists of three main components: the Deconvolution Engine (DE), the Coefficients Buffer (CB) and the Accumulation Logic (AL). The DE is the computing core that process T_N *ifmaps* and T_M kernels in parallel. The latter are previously stored within the CB to permit continuous processing steps without breaks. Finally, the AL exploits fast adder trees, implemented within DSP slices, to accumulate provisional results that are collected within local Block RAMs until the *ofmaps* generation process is completed. A Finite State Machine (FSM) orchestrates the operations and provides proper AXI4 controls to make the DLP suitable for heterogenous SoCs.

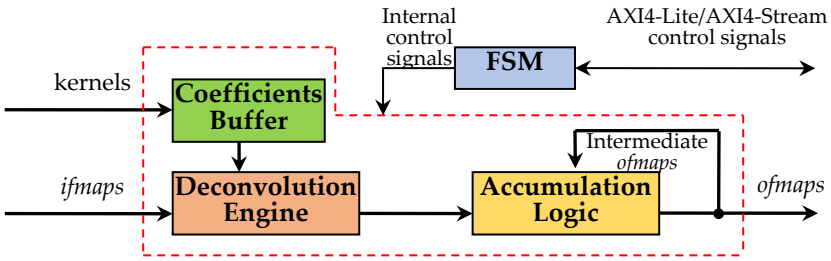


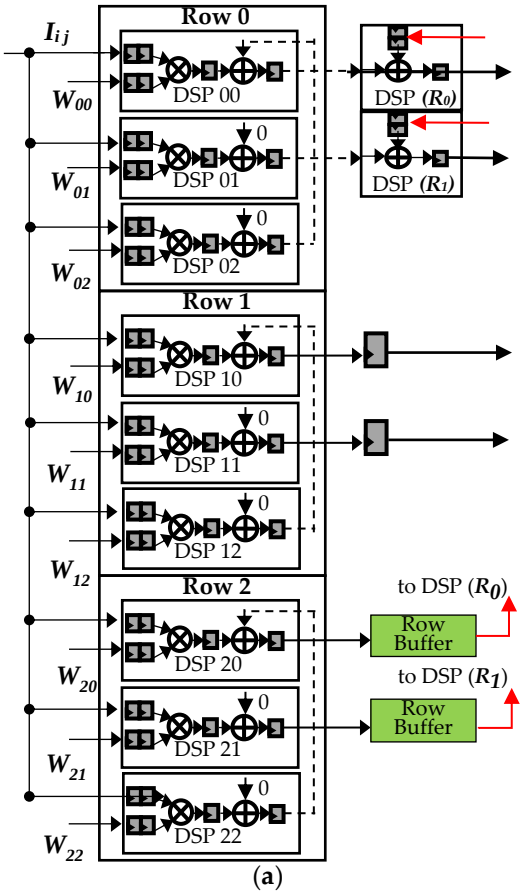
Figure 3: The Deconvolution Layer Processor

2.1.1. The Deconvolution Engine

The DE consists of $T_M \times T_N$ Deconvolution Units (DUs) that execute parallel transposed convolutions. Each DU reads an input pixel per clock cycle and a $K \times K$ kernel per *ifmap*. $K \times K$ DSPs multiply the current pixel by the kernel. Moreover, these slices are employed to properly comply with the overlapping regions, as dictated by step (2) of the algorithm. More precisely, such DSPs were exploited to fulfill the column overlap, i.e. the overlap exhibited by adjacent input pixels belonging to the same input row. The Xilinx DSP48E1 slices [10] can efficiently manage this situation, thanks to fast dedicated routing resources that allow to link stacked DSPs belonging to the same tile. On the contrary, extra $S \times (K–S)$ DSPs are needed to handle the overlap between adjacent pixels belonging to the same column, i.e. the row overlap. Such a scenario also requires $S \times (K–S)$ First-In-First-Out (FIFO) Row Buffers to store the overlapping results related to the previous row processing step. Their depth is related to the W size, according to the expression (1):

$$FIFO\_DEPTH = \left\lceil \frac{S \times (W - 1) + K}{S} \right\rceil \quad (1)$$

In order to design a run-time reconfigurable FIFO, the SRLC32 primitive [11] was chosen, that consists of a 32-bit dynamic-length shift register implemented within just a Look-Up Table (LUT). For the sake of simplicity, **Figure 4a** depicts the DU when $K = 3$ and $S = 2$. Nine DSPs are arranged within 3 Row blocks. This configuration exhibits one column (**Figure 4b**) and one row (**Figure 4c**) of overlap between adjacent windows, i.e. the 1st and 3rd columns and the 1st and 3rd rows, respectively. This DU furnishes 4 output pixels for each cycle, after an initial latency that depends on the DSP pipeline stages.



$I(i)(j)\times W_{00}$	$I(i)(j)\times W_{01}$	$I(i)(j)\times W_{02} + I(i)(j+1)\times W_{00}$
$I(i)(j)\times W_{10}$	$I(i)(j)\times W_{11}$	$I(i)(j)\times W_{12} + I(i)(j+1)\times W_{10}$
$I(i)(j)\times W_{20}$	$I(i)(j)\times W_{21}$	$I(i)(j)\times W_{22} + I(i)(j+1)\times W_{20}$

(b)

$I(i)(j)\times W_{00}$	$I(i)(j)\times W_{01}$	$I(i)(j)\times W_{02}$
$I(i)(j)\times W_{10}$	$I(i)(j)\times W_{11}$	$I(i)(j)\times W_{12}$
$I(i)(j)\times W_{20} + I(i+1)(j)\times W_{00}$	$I(i)(j)\times W_{21} + I(i+1)(j)\times W_{01}$	$I(i)(j)\times W_{22} + I(i+1)(j)\times W_{02}$

(c)

Figure 4: The Deconvolution Unit: (a) architecture; (b) column overlap; (c) row overlap

The DE also manages border pixels carefully. More precisely, the DE must stop the stream in two main moments:

(a) At the end of each line, the DE must wait $\left\lceil \frac{S\times(W-1)+K}{S} \right\rceil - W$ clock cycles before reading the next line.

(b) After the *ifmap* reading, the DE must wait $\left\lceil \frac{S\times(H-1)+K}{S} \right\rceil - H$ lines of latency.

Both of (a) and (b) are managed by the FSM, which controls the *AXI Stream Slave Ready* signal and manages multiplexing logic that furnishes zeros instead of valid pixels.

2.1.2. The Coefficients Buffer

The CB, depicted in **Figure 5a**, mainly consists of a register file able to store $K \times K \times T_N \times T_M$ N-bit coefficients. It reads $T_N \times N$-bit words that represent homologous coefficients related to the T_N *fmaps* parallel processed. Therefore, all of them are correctly stored within just $K \times K \times T_M$ clock cycles. Then, the Separate & Route logic furnishes the kernels to the DE.

This module complies with the AXI4-Stream interface efficiently, to avoid extra latencies among the processing steps. For the sake of clarity, **Figure 5b** illustrates a timing diagram. Basically, for each step, the CB reads the kernels needed for the next step. Meanwhile, the DE processes current *ifmaps* by using the coefficients stored during the previous one and captured by each DU at the begin of the current step (*Kernels capture*).

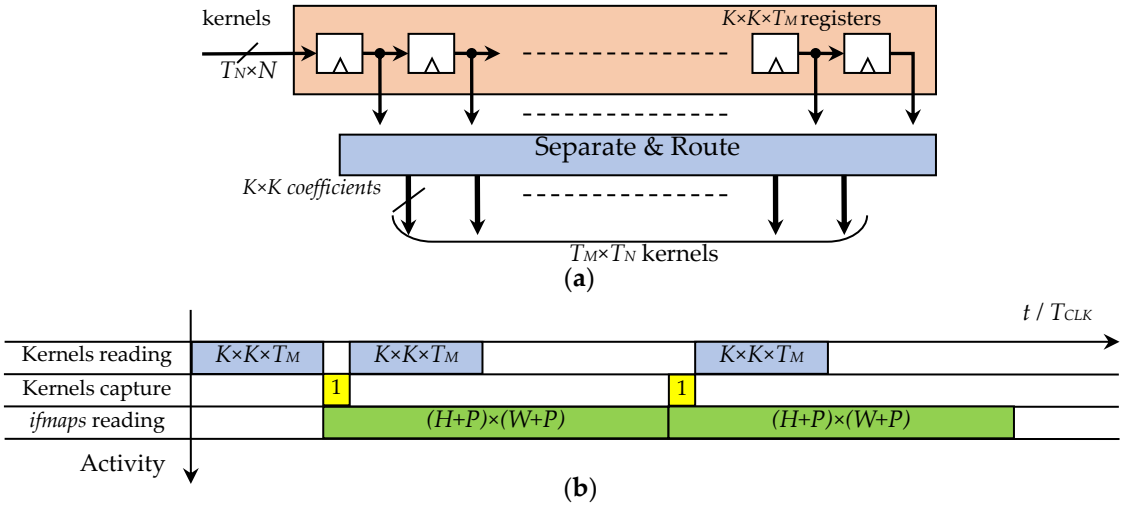


Figure 5: The Coefficients Buffer: (a) architecture; (b) example timing

2.1.3. The Accumulation Logic

The AL performs pixel-wise additions to provide the *ofmaps*, as depicted in **Figure 6**. In such a module, sums are also executed by DSP slices to improve performances.

The DE furnish its results, hereby indicated as grouped DU outputs (*grDUouts*), to $S \times S \times T_M$ adder trees. The Route module provides the correct *grDUout* to each adder tree, consisting of $T_N - 1$ adders. The provisional results are accumulated within Block RAMs (BRAMs) [12] configured as Simple Dual Port memories. In fact, the write port receives the current results, whereas the read port provides the accumulations of the previous processing steps to $S \times S \times T_M$ extra adders to progressively compose the *ofmaps*. It is worth highlighting that during the last computing step, the write ports of the BRAMs receive zeroed words, through a bank of Multiplexers, to prepare the local memory space for the next filtering phase. Finally, the Quantize & Group module performs N-bit quantization and arranges results to comply with the $S \times S$ throughput of the algorithm.

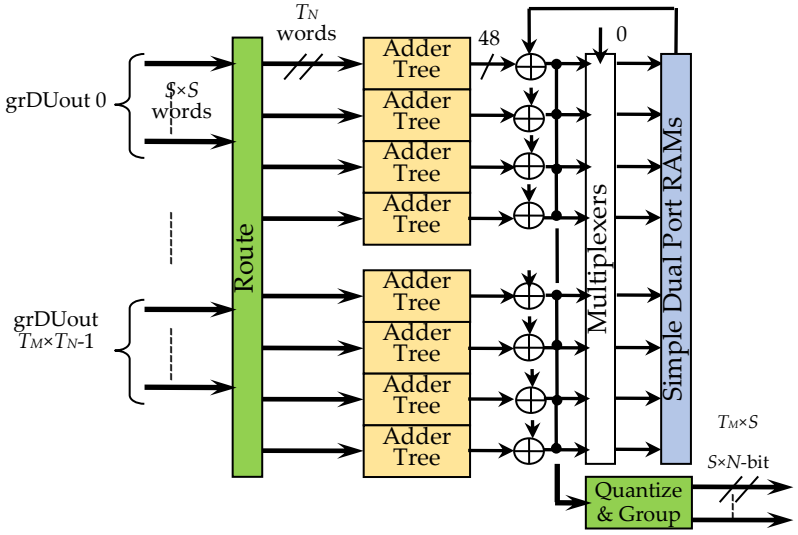


Figure 6: The Accumulation Logic in the case of S = 2 and T_M = 2

Figure 7 clarifies the *ofmap* structure if S = 2. For each cycle, the AL provides four parallel results per *ofmap*, that can be arranged within 2 couples: the former refers to column-wise adjacent pixels belonging to the *i*-th row, whereas the latter represents nearby pixels belonging to the (*i*+1)-th row.

1	1	2	2	3	3
1	1	2	2	3	3
4	4	5	5	6	6
4	4	5	5	6	6
7	7	8	8	9	9
7	7	8	8	9	9

Figure 7: The *ofmap* space

2.1.4. The embedded system

The proposed DLP was embedded within the Zynq-7020 SoC [13]. **Figure 8** depicts the whole system, which essentially consists of the Processing System (PS) and the Programmable Logic (PL). The former configures the modules within the PL by using the Master General Port 0 (MGP0) through the AXI4-Lite protocol. The latter hosts the DLP and DMA/VDMAs IPs [14,15] to fulfill the data movement from/to the external DDR memory. In particular, the DMA exploits the 64-bit Accelerator Coherency Port (ACP) to move coefficients from the DDR to the DLP. Conversely, four VDMAs manage the *fmaps* transfer by using 32-bit bidirectional High-Performance Ports (HP0÷HP3).

The DLP was configured to support $K = 5$, $S = 2$, $T_M = 2$, $T_N = 3$ and 32×32 *ifmaps*. For each clock cycle, three VDMAs send 16-bit data to the DLP, by means of the AXI Stream Combiner [16] which synchronizes them within 48-bit words. Conversely, the eight 16-bit parallel results provided by the DLP, merged within 128-bit word, are provided to the VDMAs as 32-bit data by using the AXI Stream Broadcaster [16]. It is worth underlining that *fmaps* are arranged in raster-order within the DDR, thus being ready for subsequent deconvolution layers without needing of data reorganization or complex memory management.

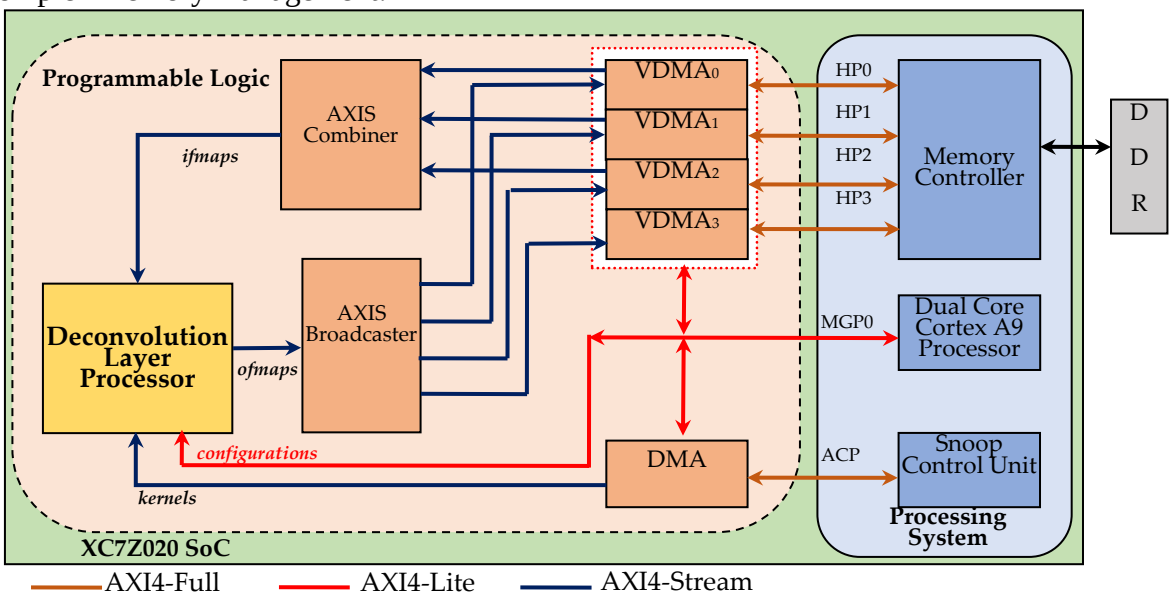


Figure 8: The proposed embedded system

2.2. Software

The PS runs a C routine to take care of configuring IP and manage data transfers properly. The operations flow is as follows:

1) The PS write test data (coefficients and *fmaps*) within proper DDR areas, paying attention not to access to dedicated locations (e.g. areas in which the C instructions are written).
2) Therefore, it resets all the IP involved in computation and transfers (i.e. the DLP, the DMA and the VDMAs).
3) After that, the PS writes to the DLP's configuration registers by exploiting the AXI4-Lite interface. Refer to section 2.3.1. for details.

a) `Xil_Out32(XPAR_DECONV_LAYER_PROC_V1_0_0_BASEADDR,0x3C)`

This line configures the dynamic size of each SRLC32 slice and indicates if the current processing step is intermediate or if it is the last.

b) `Xil_Out32(XPAR_DECONV_LAYER_PROC_V1_0_0_BASEADDR + 0x4,0x22C0FC82)`

This line configures the latencies related to the FSM unit.

c) `Xil_Out32(XPAR_DECONV_LAYER_PROC_V1_0_0_BASEADDR + 0x8,1155)`

This line indicates the number of 32-bit words managed by each VDMA, decreased by 1 since the count starts from zero.

Hence, the dedicated accelerator is ready to execute its operations.

4) The DMA, responsible for coefficients transfers, is configured by these lines of code:

a) `Xil_Out32(XPAR_AXI_DMA_0_BASEADDR + 0x0, 0x1) // MM2S_DMACR`
b) `Xil_Out32(XPAR_AXI_DMA_0_BASEADDR + 0x18, 0x01000000) // MM2S_SA`
c) `Xil_Out32(XPAR_AXI_DMA_0_BASEADDR + 0x28, 400); // MM2S_LENGTH`

The former runs the MM2S channel, whereas (b) points to the coefficients location within the DDR. Finally, the latter starts data acquisition, by indicating how many bytes must be written into the CB within the DLPE.

5) The PS configures the S2MM ports of the VDMA, by executing the following lines for each of them:

a) `Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR+0x30,0x11011);// S2MM_VDMACR`
b) `Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR+0xAC,0x01400000);//S2MM_START_ADDR`
c) `Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR+0xA8,34*4*2);//S2MM_FRMDLY_STRIDE`
d) `Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR+0xA4,34*4);//S2MM_HSIZE`
e) `Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR+0xA0,34);//S2MM_VSIZE`

The instruction (a) runs the port and configures it in park mode to process just a frame and then stop. (b) indicates the frame buffer location in which the *ofmap* storing begins. Instruction (c) and (d) are related to the frame spatial features: the former indicates the address pointer for each new line; the latter expresses the number of bytes per line. Finally, (e) indicates the number of lines per frame and it triggers data acquisition.

6) Similarly, the PS configures the MM2S ports.

a) `Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR+0x0,0x11011);//MM2S_VDMACR`
b) `Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR+0x5C,0x01100000);//MM2S_START_ADDR`
c) `Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR+0x58,32*32*2);//MM2S_FRMDLY_STRIDE`
d) `Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR+0x54,32*32*2);//MM2S_HSIZE`
e) `Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR+0x50,1);//MM2S_VSIZE`

7) At the completion of the operations, the PS reads DDR location in order to verify the correctness of stored data.

2.2.1. Data organization

In order to clarify the VDMAs configuration instructions, details about the *ifmap* and *ofmap* data structure are here provided.

a) Each *ifmap* is read as a line of H×W 16-bit words. According to the test setup, H = W = 32. Hence, HSIZE = 32×32×2 = 2048 bytes per line. Furthermore, since each VDMA reads adjacent locations continuously, STRIDE = HSIZE. Conversely, VSIZE refers to the number of processing steps. Therefore, if just an *ifmap* is sent, VSIZE = 1.

b) Each *ofmap* is written into the DDR by means of 2 VDMAs. The former arranges the even rows, whereas the latter the odd ones. In such case, the DLP provides 68×68 *ofmaps*. Since each VDMA sends two adjacent pixels at a time, each output line will consist of thirty-four 32-bit words, that are 136 bytes (HSIZE). Moreover, according to the rows interleaving among VDMAs, the STRIDE doubles the HSIZE value. Finally, VSIZE refers to the line written by each VDMA, that are the half of the total (34).

2.3. Design Reuse

The design of the DLP is highly reusable since it was written by exploiting VHDL parametric constructs. As an example, it is worth spending few words about the Deconvolution Unit (DU), which is the key innovation. As previously explained, its design depends on the kernel size *K*, the stride *S* and the vertical size *Wo* of the *ofmap*. While *K* and *S* are static parameters and have to be configured before the synthesis, the latter can be furnished run-time to dynamically adapt the engine to features of different sizes. Therefore, it complies well with uniform kernel size DCNN models, such as the DCGAN [8]. **Figure 9** depicts a sketch of the code of the DU, to provide the reader with a practical example. More precisely, it refers to the DSPs' generation within the Row Block.

```
-- DSPs instantiation
-- Notes for T parameter: T = 0 => dedicated DSPs' fast routing; T = 1 => fabric routing
DSP_g: for i in 0 to K-1 generate
    DSP_g_ovlp: if S < K generate
        DSP_g_1a: if (i < (K-S) and i < S and T = 0) generate
            DSP1ax: dsp_PCIN_PCOUT port map(clk=>clk,rst=>rst,cea=>cei,ceb=>cew,A=>I_int,B=>W_int(i),C=>C_int(i),P=>Q(i));
        end generate DSP_g_1a;
        DSP_g_1aa: if (i < (K-S) and i < S and T = 1) generate
            DSP1aax: dsp_PCIN_P port map(clk=>clk,rst=>rst,cea=>cei,ceb=>cew,A=>I_int,B=>W_int(i),C=>C_int(i),P=>Q(i));
        end generate DSP_g_1aa;
        DSP_g_1b: if (i < (K-S) and i >= S) generate
            DSP1bx: dsp_PCIN_PCOUT port map(clk=>clk,rst=>rst,cea=>cei,ceb=>cew,A=>I_int,B=>W_int(i),C=>C_int(i),P=>P_reg(i-S));
        end generate DSP_g_1b;
        DSP_g_1c: if (i >= (K-S) and i < S and T = 0) generate
            DSP1cx: dsp_C0_PCOUT port map(clk=>clk,rst=>rst,cea=>cei,ceb=>cew,A=>I_int,B=>W_int(i),C=>C_int(i),P=>Q(i));
        end generate DSP_g_1c;
        DSP_g_1cc: if (i >= (K-S) and i < S and T = 1) generate
            DSP1ccx: dsp_C0_P port map(clk=>clk,rst=>rst,cea=>cei,ceb=>cew,A=>I_int,B=>W_int(i),C=>C_int(i),P=>Q(i));
        end generate DSP_g_1cc;
        DSP_g_1d: if (i >= (K-S) and i >= S) generate
            DSP1dx: dsp_C0_PCOUT port map(clk=>clk,rst=>rst,cea=>cei,ceb=>cew,A=>I_int,B=>W_int(i),C=>C_int(i),P=>P_reg(i-S));
        end generate DSP_g_1d;
    end generate DSP_g_ovlp;
    DSP_g_noovlp: if S = K generate
        DSP2x: dsp_C0_P port map(clk=>clk,rst=>rst,cea=>cei,ceb=>cew,A=>I_int,B=>W_int(i),C=>C_int(i),P=>Q(i));
    end generate DSP_g_noovlp;
end generate DSP_g;
```

Figure 9: Sketch code of the Deconvolution Unit

2.3.1. Run-time reconfigurability

The DLP exploits the AXI-Lite interface for run-time reconfigurability. In particular, the VHDL code provided by Xilinx was used to manage the protocol properly. Three 32-bit control registers were implemented, as follows.

- Register CTRL0 manages the Last Channel (LC) bit to control the multiplexing logic within the AL, and the dynamic depth (SR_DEPTH) of Row Buffers within each DU to adapt the DLP to different *fmap* sizes.
  a) LC = 0 (intermediate step); 1 (last step).
  b) SR_DEPTH = FIFO_DEPTH – 4.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
																										SR_DEPTH					**LC**

- Register CTRL1 manages specific latencies related to the FSM:
  a) MDPTH = (H+P)×(W+P) – 2, that refers to the padded *ifmap* area.
  b) WL = W – 1, that refers to the *ifmap* W size.
  c) Wext_L = (W+P) – 2, that refers to the *ifmap* padded W size.
  d) Pad_L = P+P×(W+P) – 1, that refers to the lines of latencies at the end of each *ifmap*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
		Pad_L							**Wext_L**						**WL**						**MDPTH**										

- Register CTRL2 manages the latency related to the AXI Stream Master Interface to complete the data transmission: NDATA = (H+P)×(W+P) – 1.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
																					NDATA										

According to the test case, SR_DEPTH = 30, MDPTH = 1154, WL = 31, Wext_L= 32, Pad_L = 69, NDATA = 1555. These values justify the hex words indicated in the step 3 in section 2.2.

Please refer to the VHDL_template_notes.txt within the doc sub-directory for details to proper configure the DLP_Top_Level unit to easily adapt the project to different application tasks.

3. Results

The DLP was configured to accelerate the 5th layer of the DCGAN models, that supports K = 5, S =2, 32×32 *ifmaps*. Managing a valuable parallelism was the main challenge during the design phase, considering the limitations of the Zynq-7020 both in terms of resources and performances.

The bottleneck related to the preliminary resource planning lies in DSP slices, since the must deal with multiplications, row/column overlap and accumulations. More precisely, it was demonstrated that the number of DSPs per DU is $K\times K+S\times(K-S)$, and extra $S\times S$ DSPs are needed for accumulations. Therefore, the maximum parallelism achievable, according to the XC7Z020 datasheet [13], can be express by the (2):

$$\max_{DSP} parallelism = \left\lfloor \frac{220}{K \times K + S \times (K - S) + S \times S} \right\rfloor \quad (2)$$

In such a case, $parallelism \leq 6$. Therefore, since $parallelism = T_M \times T_N$, the combination $T_M = 2$, $T_N = 3$ was chosen.

According to this configuration, the bandwidth requirements within the ES were analyzed. It is worth underlining that the main limitation is referred to the DDR, whose bandwidth is 4.16 GB/s [17]. Therefore, taking into account the scenario depicted in **Figure 8**, the maximum running frequency is dictated to 118 MHz. Therefore, the ES runs at the 111 MHz clock frequency.

Resource Utilization	**LUTs**	12,830	(24.1 %)
	FFs	17,728	(16.7 %)
	BRAMs [Mb]	1.21	(24.6 %)
	DSPs	210	(95.5 %)
Actual Clock Frequency [MHz]		111	
Throughput [GOPS]		40	
Density Efficiency [GOPS/DSP]		0.2	
Power [W]		1.8	

Table 1: Embedded system characterization

Table 1 reports the main characteristic of the proposed architecture. The system exploits a limited amount of resources, except for DSPs intensively used to sustain the referred parallelism. Throughput, expressed in terms of Giga Operations Per Second (GOPS), is related to the MACs executed by the DSP slices. The average power consumption of 1.8 W, directly provided by VIVADO, includes both PL and PS contributions. Furthermore, it is worth underlining that the Worst Negative Slack (WNS) is consistent with the estimation provided. Indeed, WNS = 0.544 ns that leads to the 118 MHz frequency

4. References

[1] Alberto Garcia-Garcia, Sergio Orts-Escolano, Sergiu Oprea, Victor Villena-Martinez, and Jose Garcia Rodriguez, "A review on deep learning techniques applied to semantic segmentation," *arXiv preprint arXiv:1704.06857, 2017.*

[2] C. Dong, C. C. Loy, K. He and X. Tang, "Image super-resolution using deep convolutional networks", *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 38, no. 2, pp. 295-307, Feb. 2015.

[3] I. Goodfellow et al., "Generative adversarial nets", *Advances in neural information processing systems*, pp. 2672-2680, 2014.

[4] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv:1603.07285*, 2016.

[5] S. Mittal, "A survey of FPGA-based accelerators for convolutional neural networks," *Neural Comput. Appl.*, vol. 32, no.4, pp. 1–31, 2018.

[6] X. Zhang, S. Das, O. Neopane and K. Kreutz-Delgado, A design methodology for efficient implementation of deconvolutional neural networks on an FPGA, *arXiv:1705.02583*,2017

[7] "AMBA® 4 AXI4TM, AXI4-LiteTM, and AXI4-StreamTM Protocol Assertions User Guide." [Online]. Available: http://infocenter.arm.com/help/index.jsp.

[8] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv:1511.06434*, 2015

[9] ZedBoard (Zynq™ Evaluation and Development) Hardware User's Guide, Version 1.1. August 2012. Available online: https://www.xilinx.com/products/boards-and-kits/1-elhabt.html

[10] Xilinx and Inc, "7 Series DSP48E1 Slice User Guide (UG479)," 2018. Available online: https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

[11] Xilinx and Inc, "7 Series FPGAs Configurable Logic Block User Guide (UG474)," 2016. Available online : https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

[12] Xilinx and Inc, "Block Memory Generator LogiCORE IP, PG058(v8.3)," 2017. Available online:https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_3/pg058-blk-mem-gen.pdf

[13] Xilinx and Inc, "Zynq-7000 SoC First Generation Architecture, DS190 (v1.11.1)," 2018. Available online: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[14] Xilinx and Inc, “AXI DMA LogiCORE IP, PG021 (v7.1),” 2018. [Online]. Available online: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf

[15] Xilinx and Inc, “AXI Video Direct Memory Access v6.2 LogiCORE IP Product Guide Vivado Design Suite.” Available online: https://www.xilinx.com/support/documentation/ip_documentation/axi_vdma/v6_2/pg020_axi_vdma.pdf

[16] Xilinx and Inc, “AXI4-Stream Infrastructure IP Suite v3.0 LogiCORE IP Product Guide Vivado Design Suite.” Available online: https://www.xilinx.com/support/documentation/ip_documentation/axis_infrastructure_ip_suite/v1_1/pg085-axi4stream-infrastructure.pdf

[17] Xilinx Zynq-7000 External Memory Interfaces. Available online: https://www.xilinx.com/products/technology/memory.html#externalMemory