High-Level Synthesis of Hardware Accelerators for Deconvolution Engines

Cristian Sestito
Department of Informatics, Modeling, Electronics and System Engineering
University of Calabria
Arcavacata di Rende, Italy
e-mail: cristian.sestito@unical.it

Robert Stewart
Department of Computer Science
Heriot-Watt University
Edinburgh, United Kingdom
e-mail: R.Stewart@hw.ac.uk

Stefania Perri
Department of Mechanical, Energy and Management Engineering
University of Calabria
Arcavacata di Rende, Italy
e-mail: s.perri@unical.it

***Abstract*—Convolutional and Deconvolutional Neural Networks are widespread in several modern computer vision applications, such as high-resolution imaging, object classification and generation, image segmentation and many others. While several efficient hardware architectures are known in literature to accelerate the convolution task, the design of accelerators for deconvolution is still an open problem. The few existing deconvolution engines are customized to exploit in the best possible way specific hardware resources, thus suffering from platform-dependency that certainly allows maximizing speed performances and power-resource efficiency, but, on the other hand makes these designs unsuitable for the high-level synthesis approach. This paper presents a deconvolution structure described in the C++ high-level language and then synthesized at the register-transfer level of abstraction. Results demonstrate that, when characterized within the Xilinx XC7VX980tffg1930-1 device, the described architecture can up-sample a 256×256 input image to the 1024×1024 resolution using less than 3000 LUTs, 1028 18Kb BRAMs and ~640 FFs. The reached 121 MHz running frequency guarantees a frame rate higher than 50 fps to be achieved.**

Keywords-Hardware accelerators; High-Level Synthesis; Deconvolution; Multiply Accumulations; FPGAs.

I. INTRODUCTION

Modern deep learning applications [1]-[3], including image segmentation, object generation and high-resolution imaging, exploit both Convolutional and Deconvolutional Neural Networks (CNNs and DCNNs). The former progressively down-sample the digital images received as input to extract relevant features, whereas the latter elaborate the input images to extrapolate new features. As it is well known, Convolution (CONV) and Deconvolution (DECONV) are nothing more than Multiply Accumulations (MACs) performed on the pixels of the received images and the kernel coefficients of $k \times k$ filters. However, despite to their similarity, while CONV has been extensively used in several CNNs, such as AlexNet [4], GoogleNet [5], ResNet [6], VGG16 [7], just to cite some of the most popular models, DECONV has received a great deal of attention only recently: it is an efficient approach to furnish high-resolution images and, therefore, it has become the basic operation of generative neural networks [8][9].

Generally speaking, a DECONV engine receives a low-resolution $H \times W$ image and a $k \times k$ filter and produces a high-resolution $Ho \times Wo$ output image. Several approaches can be exploited to perform such an operation, each having its own pros and cons. As shown in [10], DECONVs can be computed by executing classical CONVs. In order to do this, with S and P being the adopted stride and padding, respectively, the input image is preliminarily strided, by interleaving $S-1$ zeros between each pair of adjacent pixels, and padded by inserting P zeros on the borders. The image obtained in this way is processed through a classical CONV, which is a benefit in terms of design efforts, given that engines designed for CONV can be utilized also to perform DECONV. However, inserted zeros cause useless zeroed MACs and lead to unbalanced workloads. Moreover, the input reorganization, required to stride and pad the input images, limit the achievable speed performances.

As an alternative, the technique proposed in [11] directly multiplies each input pixel by the filter coefficients, thus computing a block of $k \times k$ products. In this way, the blocks of products related to adjacent pixels are overlapped and, to perform DECONV correctly, up to $k-S$ overlapping rows and columns must be properly managed, which increases both the computational complexity and the delay.

The designs presented in [12]-[18] improve the above approach to implement efficient hardware DECONV engines within FPGA-based Systems-on-Chip (SoCs) able to accelerate the complex segmentation and the super-resolution imaging tasks.

As deeply discussed in [19], also the Winograd algorithm can be exploited to perform DECONV. The main benefit of this solution is the very high speed achieved, but, as a drawback, input images and filters must be preliminarily transformed in the Winograd domain, which introduces significant resources and power overheads.

All the previously cited state-of-the-art papers present efficient DECONV engines customized to exploit in the most efficient way the hardware resources available within a specific FPGA device. If on the one hand this choice allows speed performances to be maximized, limiting the power dissipation and the hardware resources requirements, on the other hand it introduces specific realization platform-dependency, which makes such designs unsuitable for the High-Level Synthesis (HLS). The latter allows describing complex tasks, like those performed by DCNNs, in a high-level language (e.g., C/C++) letting the software tool automatically provide the description at the Register-Transfer Level (RTL) of abstraction. The HLS design approach offers a precious aid to the users who: 1) must

 ISBN: 978-1-68558-009-4

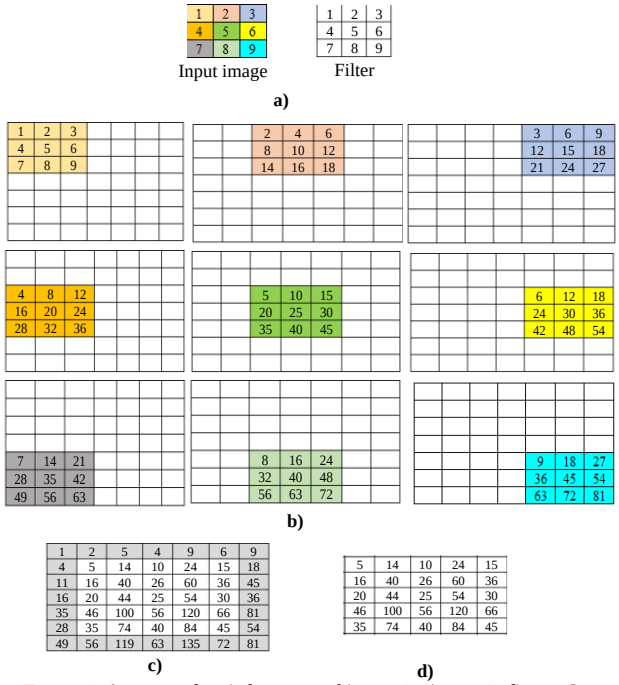

Figure 1. An example: a) the inputs; b) step 1; c) step 2; d) step 3.

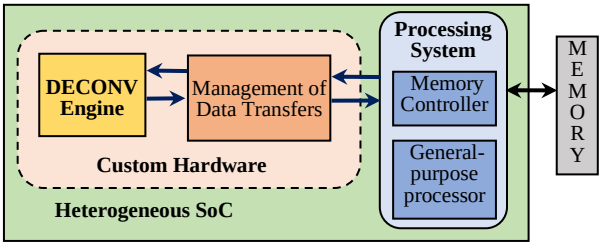

Figure 2. Typical structure of a heterogeneous SoC.

```
1: for (unsigned int i = 0; i < k; i++) {
2:     for (unsigned int j = 0; j < k; j++) {
3:         #pragma HLS PIPELINE II=1
4:         filt[i][j]=filter.read();
5: for (unsigned int r = 0; r < H; r++) {
6:     for (unsigned int c = 0; c < W; c++) {
7:         #pragma HLS PIPELINE II=1
8:         Pix=InIm.read();
9:         for (unsigned int i = 0; i < k; i++) {
10:            for (unsigned int j = 0; j < k; j++) {
11:                // Multiply the generic pixel by the filter
12:                Prods[i][j]=Pix*filt[i][j];
13:                // Store the products to be reused for the column overlap
14:                if (j >= S)
15:                  CBuff[i][j−S]=Prods[i][j];
16                 // Sum up overlapped columns
17:                if (j<k−S)
18:                  if (c==0)
19:                      SumCol [i][j]=Prods [i][j];
20:                  else SumCol[i][j]=Prods [i][j]+CBuff[i][j];
21:                else SumCol[i][j]=Prods[i][j];
22:               // Store the results to be reused for the row overlap
23:                if (i>=S) {
24:                  if (j<S)
25:                    RBuff[i-S][j][c]=SumCol[i][j];
26:                }
27:               //Sum up overlapping rows
28:                if (i < k-S) {
29:                  if (j < S) {
30:                    if (r == 0)
31:                        SumRow[i][j]=SumCol [i][j];
32:                    else SumRow[i][j]=SumCol[i][j]+RBuff[i][j][c];
33:                  }
34:                }
35:                else SumRow [i][j]=SumCol [i][j];
36:                // Map the results to the output space
37:                for (unsigned int i = 0; i < S; i++) {
38:                    for (unsigned int j = 0; j < S; j++) {
39:                      OBuff[c+i*W+r*S*H].range(16*j+15,16*j)=SumRow[i][j];
40:                    }
41:                }
42:            }
43:        }
44:    }
45: }
```

Figure 3. The synthesizable C++ code describing the DECONV task.

comply with limited realization time; 2) are not familiar with hardware designs at a low-level of abstraction; 3) desire platform-independent portable design descriptions. Indeed, HLS tools can access sets of libraries providing several classes of synthesizable functions that can be exploited to describe complex tasks. Moreover, proper directives and pragmas can be used within the description code to architecturally constrain the synthesis result. Stimulated by these considerations, this paper presents the design of a DECONV engine based on the HLS approach.

The rest of the paper is organized as follows: Section II reviews the adopted DECONV method; Section III details the synthesizable C++ code, written, verified and synthesized with the Xilinx Vivado HLS Tool, and presents post-synthesis results; future works are briefly described in Section IV; finally, conclusions are drawn in Section V.

II. THE ADOPTED DECONV METHOD

The proposed DECONV engine implements the Input-Oriented-Mapping (IOM) strategy [11]. It performs the generic computation within three steps: 1) multiply each input pixel by the filter coefficients, thus providing a block of $k \times k$ products; 2) sum up the products belonging to the $k–S$ rows (columns) overlapped with adjacent blocks; 3) crop the borders of the output image to modulate its size to $Ho \times Wo$, as given in (1), where P_I and P_O are the input and output padding, respectively.

$$H_O = (H-1) \times S + k - 2P_I + P_O \quad (1a)$$

$$W_O = (W-1) \times S + k - 2P_I + P_O \quad (1b)$$

To better explain how the referred method runs, let us examine the example reported Figure 1. It refers to the case in which H=W=3, k=3, S=2, P_I=1, P_O=0. Figure 1b shows how the 3×3 blocks of products obtained by the step 1 (i.e., multiplying each input pixel by the filter) should be arranged into the output space. In this case, adjacent bocks have only 1 overlapping row (column), therefore the accumulations performed in the step 2 lead to the 7×7 provisional image of Figure 1c. Since the size of the output image obtained by (1) is H_O=W_O=5, the gray borders are cropped in the step 3, thus finally producing the output image reported in Figure 1d.

III. THE SYNTHESIZABLE C++ CODE AND POST-SYNTHESIS RESULTS

The synthesizable C++ routine purposely written to exploit the HLS design approach has been organized assuming that the DECONV engine is the computational core of a custom hardware module exploited within a typical

heterogeneous System-On-Chip (SoC) structured as schematized in Figure 2. In such an architecture, data to be processed and produced results are stored in the external memory. As usually happens, read and write memory accesses are managed by the memory controller that communicates directly with the modules responsible for the management of data transfers, like Direct Memory Access modules (DMAs), Central DMAs (CDMAs) or Video DMAs (VDMAs).

From Figure 3, it can be seen that the engine processes the streams *filter* and *InIm* that collect the $k \times k$ filter coefficients and the $H \times W$ pixels of the input image, respectively (lines 1-8). As explained above, the generic pixel *Pix* is multiplied by the filter coefficients, thus providing the block of products *Prods* (lines 9-12). In order to properly manage the overlapping columns between adjacent blocks of products, the 2D array *CBuff* is exploited to provisionally store the overlapping products that must be summed up (lines 13-21) taking into account where the currently processed pixel is located within the input image. To correctly treat also the overlapping rows between adjacent blocks of products, the 3D array *RBuff* is also used. Given that the input image is fed in the raster scan order, the 3D data structure is needed to store: the results obtained by the previous sum of overlapping columns; the results obtained by the current sum of the overlapping rows; and the products that are being computed on the next incoming pixel (lines 22-35). Finally, the results are stored in the output buffer *OBuff* (lines 36-45).

It is worth noting that, in order to architecturally constrain the synthesis result, the C++ code reported in Figure 3 uses the directive *#pragma HLS PIPELINE II=1* several times to introduce pipelining with an Initiation Interval (*II*) equal to 1. The latter ensures that a new input is read at each clock cycle, thus allowing the incoming data and the produced results to be continuously streamed-in and streamed-out.

The above C++ code has been successfully simulated and synthesized using the Vivado HLS 2019.2 CAD tool. Several functional tests have been performed referring to 8-bit unsigned input images and 8-bit signed filters with different image and kernel sizes.

TABLE I. POST-SYNTHESIS RESULTS

Chip		**XC7Z020-clg484-1**					
k	***S***	***H*×*W*, *H*$_O$×*W*$_O$**	***Tclk* [ns]**	***fps***	**#BRAMs**	**#LUTs**	**#FFs**
3	2	64×64, 128×128	7.81	4878	18	1648	741
		128×128, 256×256	7.81	1219	66	1674	756
		256×256, 512×512	7.81	304	258	1729	771
5	2	64×64, 128×128	7.81	4878	20	2256	1105
		128×128, 256×256	7.81	1219	68	2282	1122
		256×256, 512×512	7.81	304	260	2307	1139
5	4	64×64, 256×256	7.85	840	68	2862	795
		128×128, 512×512	7.85	210	260	2887	817
Chip		**XC7VX980tffg1930-1**					
5	4	256×256, 1024×1024	8.24	53	1028	2917	641
7	4	256×256, 1024×1024	8.01	37	1036	5132	1230

Some post-synthesis results obtained with the XC7Z020-clg484-1 and the XC7VX980tffg1930-1 devices for various image and filter sizes and strides are summarized in Table 1. The latter shows how the speed performances, achieved in terms of clock period (*Tclk*) and number of frames produced per second (*fps*), and the hardware resources requirements, represented in terms of occupied Lookup Tables (LUTs), Flip-Flops (FFs) and on chip 18Kb Blocks RAM (BRAMs), change with *k*, *S*, $H \times W$ and $H_O \times W_O$.

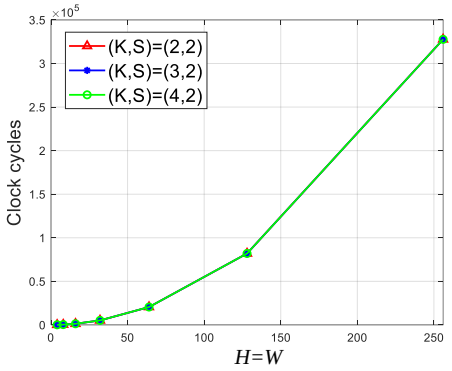

Figure 4. Number of clock cycles versus the input image size.

Obtained results clearly demonstrate that, while the stride *S* and the output image size $H_O \times W_O$ directly affect the amount of utilized BRAMs, the filter size $k \times k$ impacts on the amount of occupied LUTs and FFs. It can also be observed that the achieved frame rate strictly depends on $H \times W$, which determine how many clock cycles are required to process all the input pixels. Figure 4 plots the number of clock cycles required at various input image size when the stride is set to 2 and the filter size varies from 2 to 4. As expected, the number of clock cycles varies with the image size.

From Table 1, it can also be seen that, due to the limited amount of available BRAMs, the XC7Z020 chip is unsuitable to host the DECONV engine when 256×256 images must be up-sampled to the 1024×1024 resolution (i.e., *S*=4). For this reason, a different platform has been chosen to synthesize and characterize the proposed architecture in this operating condition. Obtained results confirm the behavior previously discussed.

IV. FUTURE WORKS

It is worth noting that the design presented in the previous Section is the preliminary version of a DECONV engine, which is intended to be used within DCNNs to implement DECONV Layers (DCLs). This means that the deconvolution operation is being performed on *M* input images (named *ifmaps*) using *N* different $M \times k \times k$ filters, thus furnishing *N* output images (named *ofmaps*), each obtained by accumulating *M* intermediate *ofmaps* in a pixel-wise manner.

Taking this into account, for future works, the architecture above described and characterized will be improved to employ a proper accumulation logic as schematized in Figure 5. Moreover, an adequate level of parallelism will be introduced to process multiple *ifmaps* contemporaneously. Generally speaking, a DCL can be made able to perform multiple deconvolutions in parallel, thus producing O_M intermediate *ofmaps* contemporaneously. A

 ISBN: 978-1-68558-009-4

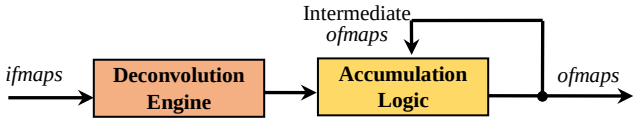

Figure 5. A possible DECONV layer architecture.

certain parallelism may be exploited also at the pixel-level to process multiple pixels of the same *ifmap* at the same time. It is expected that this capability will be introduced by exploiting the Single Instruction Multiple Data (SIMD) paradigm.

Finally, on the basis of the desired behavior other directives and pragmas will be used to use available resources more efficiently, for example including the Digital Signal Processors (DSPs). Obviously, this will further improve the achieve speed performances.

V. CONCLUSION

This paper presented a deconvolution engine designed using the high-level synthesis approach. In contrast to state-of-the-art designs proposed in literature, the description proposed here avoids specific realization platform-dependency, thus being suitable to be implemented efficiently in different realization platforms. The synthesizable C++ description here described has been characterized at different input and output image sizes, referring to various stride and kernel sizes. Some post-synthesis results have been presented referring to the XC7Z020 low-end device. Then, due to the increasing demand of on-chip memory resources, with the output image being up-sampled to the 1024×1024 resolution, a more expensive chip has been required. Due to its platform independency, the presented code can be synthesized also within different devices families. For future works, the proposed deconvolution engine can be improved to be used within DCNNs and to introduce proper level of parallelism at both frame- and pixel-level.

ACKNOWLEDGMENTS

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; The Engineering and Physical Research Council: HAFLANG (EP/W009447/1); Border Patrol (EP/N028201/1); Serious Coding (EP/T017511/1).

REFERENCES

[1] I. J. Goodfellow *et al.*, "Generative adversarial nets," in Proc. of the 27th International Conference on Neural Information Processing Systems—Volume 2, Montreal, QC, Canada, 8–13 Dec. 2014, pp. 2672–2680.

[2] A. Garcia-Garcia, S. Orts-Escolano, S. Oprea, V. Villena-Martinez, and J. Garcia Rodriguez, "A review on deep learning techniques for image and video semantic segmentation," Appl. Soft Comput., vol. 70, pp. 41–65, 2018.

[3] 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, 2015.

[4] A. Krizhevsky, I. Sutskever, and G. Hinton, ''ImageNet classification with deep convolutional neural networks,'' in Proc. Neural Inf. Process. Syst. Conf. (NIPS), 2012, pp. 1097–1105.

[5] C. Szegedy *et al.*, "Going deeper with convolutions," in Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR), Boston (MA), USA, 2015, pp. 1-9.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR), Boston (MA), USA, 2015, pp. 770-778.

[7] K. Simonyan, and A. Zisserman, "Very Deep Convolutional Networks For Large-Scale Image Recognition," in Proc. Int. Conf. on Learning Representations (ICLR), San Diego (CA), USA, 2015, pp. 1-14.

[8] A. Radford. L. Metz and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," in Proc. 4th Int. Conf. on Learning Representations (ICLR 2016), San Juan, Puerto Rico, May 2016.

[9] Y. Yu, T. Zhao, M. Wang, K. Wang, and L. He, "Uni-OPU: An FPGA-Based Uniform Accelerator for Convolutional and Transposed Convolutional Networks," IEEE Trans. VLSI Syst., vol. 28, no. 7, pp. 1545–1556, 2020.

[10] V. Dumoulin, and F. Visin, "A Guide to Convolution Arithmetic for Deep Learning," [Online; Retrieved: Jul, 2022] Available: https://arxiv.org/abs/1603.07285.

[11] D. Wang, J. Shen, M. Wen, and C. Zhang, "Efficient Implementation of 2D and 3D Sparse Deconvolutional Neural Networks with a Uniform Architecture on FPGAs," Electronics, vol. 8, no. 7, pp. 1–13, 2019.

[12] S. Liu, H. Fan, X. Niu, H. C. Ng, Y. Chu, and W. Luk, " Optimizing CNN-based Segmentation with Deeply Customized Convolutional and Deconvolutional Architectures on FPGA," ACM Trans. Rec. Technol. Syst., vol. 11, no. 3, pp. 1–22, 2018.

[13] S. Liu, C. Zeng, H. Fan, H. C. Ng, J. Meng, and W. Luk, "Memory-Efficient Architecture for Accelerating Generative Networks on FPGAs," in Proc. of the IEEE International Conference on Field Programmable Technology, Naha, Okinawa, Japan, 10–14 Dec. 2018, pp. 33–40.

[14] S. Liu, and W. Luk, "Towards an Efficient Accelerator for DNN-Based Remote Sensing Image Segmentation on FPGAs," in Proc. of the 29th International Conference on Field Programmable Logic and Applications, Barcelona, Spain, 9–13 September, 2019; pp. 187–193.

[15] J. W. Chang, and S. J. Kang, "Optimizing FPGA-based convolutional neural networks accelerator for image super-resolution," in Proc. of the 23rd Asia and South Pacific Design Automation Conference, Jeju, South Korea, 22–25 January 2018, pp. 343–348.

[16] J. W. Chang, K. W. Kang, and S. J. Kang, "An Energy-Efficient FPGA-Based Deconvolutional Neural Networks Accelerator for Single Image Super-Resolution," IEEE Trans. Circ. Sys. Video Technol., vol. 30, no. 1, pp. 281–295, 2020.

[17] S. Perri, C. Sestito, F. Spagnolo, and P. Corsonello, "Efficient Deconvolution Architecture for Heterogeneous Systems-on-Chip," Journal of Imaging, vol. 6, no. 9, pp. 1-17, 2020.

[18] C. Sestito, F. Spagnolo, and S. Perri, "Design of Flexible Hardware Accelerators for Image Convolutions and Transposed Convolutions," Journal of Imaging, vol. 7, no. 10, pp. 1-16, 2021.

[19] X. Di, H. G. Yang, Y. Jia, Z. Huang, and N. Mao, "Exploring Efficient Acceleration Architecture for Winograd-Transformed Transposed Convolution of GANs on FPGAs," Electronics, vol. 9, no. 2, pp. 1–21, 2020.

 ISBN: 978-1-68558-009-4