Dynamic Reuse of Memory in 2D Convolution Applied to Image Processing

Martin Casabella, Sergio Sulca, Ivan Vignolles, Ariel L. Pola
Department of Research and Development
Fundacion Fulgor
Ernesto Romagosa 518, Córdoba X5016GQN, Argentina
Email: martin.casabella@gmail.com, ser.0090@gmail.com, ivanmvig@gmail.com, arielpola@gmail.com

***Abstract*—This paper presents a hardware architecture for two-dimensional (2D) convolution implementation on a Xilinx Artix 7 FPGA platform when there is not enough RAM memory to instantiate and store a complete image. Not only processing speed is prioritized, but also efficient resource utilization and scalability in design, so as to add as many convolution operations in parallel as needed without requiring any major changes. In the proposed structure, dynamic memory reuse is accomplished, with a linear increase in memory utilization with respect to parallelism level.**
Keywords: FPGA, hardware architecture, convolution, kernel filter, multiplier-accumulator unit, image filtering

1. Introduction

The origin of digital image processing is directly related to the development and evolution of computers. Most image filters that focus, blur, enhance edges and detect edges, among others, use convolution as a mathematical operation. Image processing is a very extensive research area with a large number of applications in multiple fields such as medicine, engineering, navigation, aeronautics, among others. Convolutional neural networks (CNN) [1] have received increasing attention in recent years. This technique uses a fast and efficient convolution implementation.

The implementation of high-speed image processing systems in field programmable gate array (FPGA) has been a very active field. This is mainly due to the ability to take advantage of bit level parallelism, pixel level, neighborhood level and task level to increase computing performance and speed. In addition, FPGAs are reconfigurable, allowing the flexibility that is often desired in neural networks. This combination of parallel processing and flexibility at very high speed, is what makes FPGAs a platform of choice for the development of these areas [2].

There are many examples in the literature of two-dimensional (2D) convolution implementations, where most of the work focuses on high performance, resource reduction and FPGA area efficiency. As shown in [3], block random access memory (BRAM) is one of the most used elements with the highest energy consumption. Therefore, the architectures seek to reduce the use of BRAMs by dividing them into two groups that consider either total storage or partial storage of the image. The first of them increases processing by applying parallelization as well as reusing resources to reduce complexity [4], [5]. On the other hand, architectures with partial storage partition the image into as many parts as convolders are used [6], [7]. The drawback presented by these schemes is the non-modularization of processing, making it difficult to improve the work rate by increasing parallelism.

In this work, we explain the concept of dynamic reuse of BRAM in 2D convolution and its complexity concerning implementation for parallel architectures. Moreover, we propose a modular architecture, where it is easy to appreciate that when the parallelism level increases, the complexity of BRAM increases linearly.

This paper is organized as follows. Section 2 introduces the image processing algorithm. The archicterure design is presented in section 3. In section 4, hardware implementation and experimental results are shown. Finally, conclusions are drawn in section 5.

2. Image Processing Algorithms

Given its local and two dimensional nature, 2-D Convolution is one of the most used image processing algorithms. Inherent parallelism of the image convolution algorithm can be exploited to increase the performance. We focus our study on a more efficient design of 2D convolution technique to reduce the usage of FPGA resources.

2.1. Convolution Operation

Bidimensional convolution is mathematically defined as

$$G(x,y) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} K(i,j) I(x-i, y-j) \qquad (1)$$

where $I(x,y)$ is an image of size $(m \times n)$ pixels, $K(i,j)$ is a set of coefficients called Kernel of size $(k \times k)$ and $G(x,y)$ is convolution result of size $(m-2 \times n-2)$ pixels because we are considered valid convolution [8].

2.2. Change of Dynamic Range

The images used for processing are grayscale because of its implementation simplicity, but the design can be extrapolated to multiple channel images by considering each channel as an independent grayscale image.

The pixels of the grayscale images $I(x,y)$ have a dynamic range between $[0, 255]$ and Kernel values $K(x,y)$ can be negative or positive. This work is part of a Deep Learning project where it is usual to operate with a

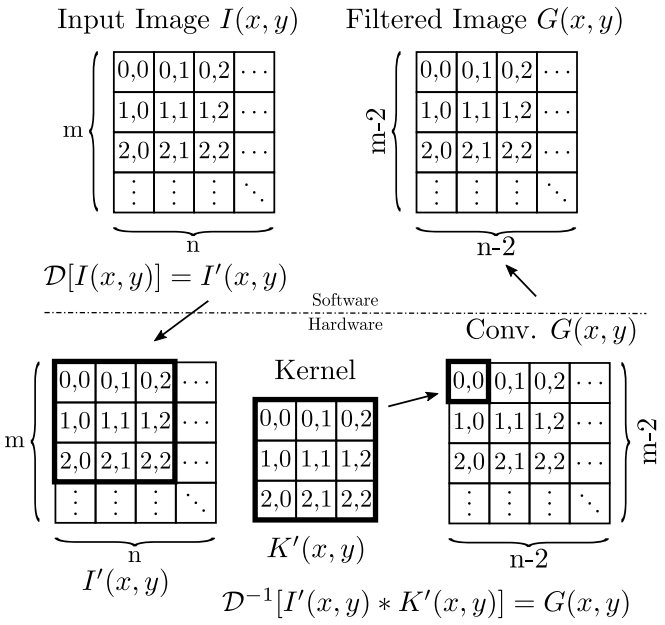

Fig. 1. Transformations of the dynamic range during the processing of the image.

dynamic range centered on zero. Therefore, we apply dynamic range expansion ($\mathcal{D}[.]$) [9] and maximum norm ($\mathcal{M}[.]$) [10] to rearrange $\mathcal{D}[I(x,y)] = I'(x,y)$ between $[0,1]$ and $\mathcal{M}[K(x,y)] = K'(x,y)$ between $[-1,1]$, respectively. Therefore, replacing in (1)

$$G(x,y) = \mathcal{D}^{-1}\left[\sum_{i=0}^{m-1}\sum_{j=0}^{n-1} K'(i,j)I'(x-i,y-j)\right], \quad (2)$$

where $\mathcal{D}^{-1}[.]$ is the dynamic range change between $[0,255]$ in the FPGA. Finally, all the processed blocks are reordered by software. The complete processing is described in Fig. 1.

3. Architecture Design

3.1. Workflow

In this section, the design of the architecture is described in detail. The entire process is divided into three stages, pre-processing, processing and post-processing. The pre-processing consists of capturing the image and applying the described transformation in Section 2 using a Python script. In addition, the script divides the image into batches and sends them through the UART port to the FPGA. A batch consists of contiguous pixel columns whose dimension is described in Section 3.2.2. In the FPGA a microprocessor receives the information and transfers it to the convolution module using a 32 bits general purpose input-output (GPIO) port. This port writes the BRAMs that store filtered data. Due to FPGA size limitations, no other interface was used, such as Ethernet or direct memory access, to read the image directly from a storage device. However, the proposed architecture is indifferent to these methods. The second stage is the processing. The batch is convolved with the kernel inside the module, which is established by the user through the

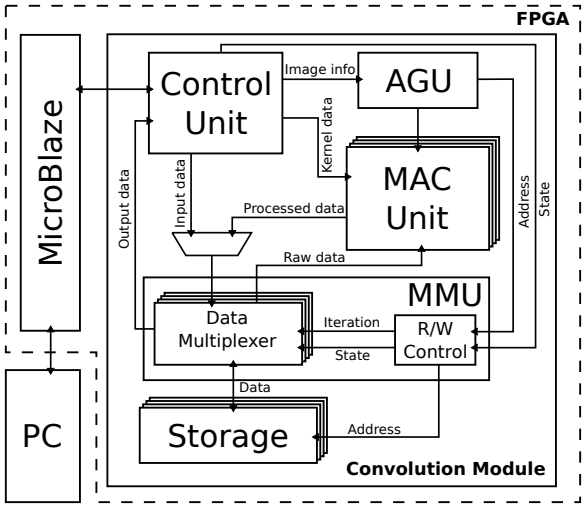

Fig. 2. Simplified block diagram of the convolution module implemented in FPGA.

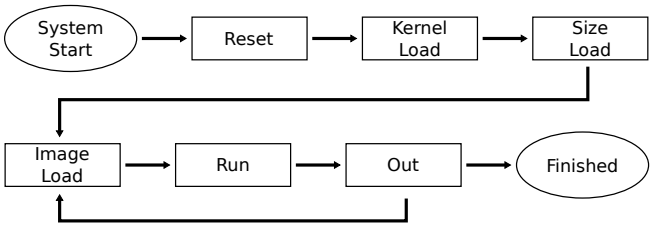

Fig. 3. State diagram that describes the transitions of events for the processing of the image.

GPIO. When the operation ends, a notification is sent to the microprocessor, which gives the order to recover the processed batch from the module and sends it to the central processing unit (CPU). Finally, the post-processing stage combines the batches on the CPU using a Python script.

3.2. Convolution Module

A simplified architecture of the module is shown in Fig. 2. The **control unit** is the one in charge of handling the communication with the processor and the **multiplier-accumulator (MAC) unit** executes the sum of the products of the kernel's coefficients with the pixels. The **address generation unit (AGU)** manages the memory addresses along with the **memory management unit (MMU)**, that decides how the values in memory are read and written. The last module is the **storage** one, which is implemented with a set of columns of the FPGA's block RAM.

3.2.1. States. The convolution module goes through several states during the work cycle, as shown in Fig. 3. states transitions are controlled by coded instructions embedded in the 32-bits input frame, and the module remain in its current state until it receives a valid instruction.

At the beginning the module must be in a **reset** state, waiting to be configured, so it must receive a reset signal. After receiving coded instruction, its is switched to the **KernelLoad** state, in which the kernel coefficients are loaded to the module. The next state is **SizeLoad** state

where the image's height, i.e. number of rows of BRAM, is loaded. The convolution method is explained in the section 3.2.2. The three previous states are executed once throughout the work cycle. The process is repeated for each new image.

The following states define the processing cycle. First, the module goes to the **ImageLoad** state and stores the batch in FPGA's block RAM. Next state is the **Run** state where the processing occurs, and while processing is being made the module can not be interrupted. As a consequence, all instructions are ignored until the batch is completely processed. As soon as processing step ends, a notification is emitted through Control Unit's *EoP* pin and the module waits for **Out** state switch instruction. After this transition, processed batch is sent back to the microprocessor.

3.2.2. Data storage. The kernel coefficients are stored in registers within each MAC unit. To store a batch, the proposed approach is to organize the block RAM in columns. Each block RAM column corresponds to one batch column. Therefore, the batch size is given by the height of the image and the number of memory columns. In addition, the maximum height of the image must be less than or equal to memory column height.

During batch processing, every pixel is read only once, so in order to do a more efficient use of the memory, the pixels that were already used are overwritten with processed pixels. This approach drastically reduces the amount of required memory because it reuses the same memory to store the input batch and the processed batch. In section 3.2.3 additional aspects of the design that reduces even more the memory use are explained.

3.2.3. Data processing. For a $k \times k$ kernel, each MAC unit takes k adjacent memory columns as input. The MAC unit first needs to load k pixels from each columns so as to have $k \times k$ pixels loaded into it to produce the first processed pixel. Then it proceeds to multiply the pixels with the kernel's coefficients and sum every term. Finally, it truncates the result and stores it into memory first position.

Once a pixel is processed, it is stored in the memory while the MAC unit shifts its image register, discarding the oldest k pixels and loading new k pixels, i.e. one from each input column identical to a first input first output (FIFO) structure. This procedure is equivalent to shifting the kernel vertically on the image. All these steps are synchronized in such a way that one processed pixel is generated in every clock cycle.

The AGU manages read and write memory addresses taking into account the latency from the time a pixel is loaded into a MAC unit until the corresponding processed pixel is stored.

Up to this point the analysis was over a single MAC unit, now the approach to work with multiple MAC units in parallel is described.

As mentioned above, k columns are needed as input to produce one processed column in a MAC unit, thus $2 \times k$ columns are needed for two MAC units. However, given the nature of convolution operation, to obtain a contiguous column it is necessary to shift the input columns by one. Hence, there is an overlap between two MAC units inputs

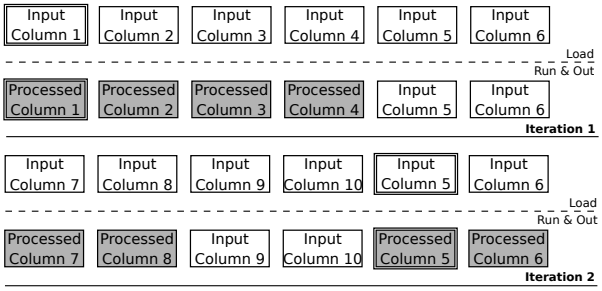

Fig. 4. Memory writing process using circular shift for $N = 4$ MAC units and Kernel $k = 3$.

which produce adjacent columns, so information can be shared. Consequently, despite k input columns are still needed per MAC unit, only $k+1$ different input columns are needed for given two units. . Therefore, for N MAC units, the number of required memory columns is reduced from $N \times k$ to $N + k - 1$, concluding that adding a new MAC unit only adds one new memory column.

As a consequence of explained overlap, there is repeated data between one batch and the next one. In order to reduce data transmission, repeated information is kept in memory and only the missing part of the incoming batch is transmitted. Given N MAC units and a $k \times k$ kernel, a $N + k - 1$ batch width is necessary, but the processed batch has a width of N columns, i.e. one for each MAC unit. So, the last $k - 1$ memory columns are not overwritten and maintain the input data. This $k - 1$ columns are reused as the first columns of the next batch, thus the transmitted batch width is reduced to N, with the exception of the first batch that maintains a width of $N + k - 1$.

Reusing memory columns written by the previous batch results in a circular shift by N places, shifting memory columns possition associated with each MAC unit, in each iteration. From the above it follows that a periodicity in the relation between the memory columns and the MAC unit's inputs must exist, where the period It is the number of iterations necessary to get to the original memory columns - MAC units inputs relation, i.e. when $It(k-1)$ is a multiple of $N + k - 1$. That is, there must be an integer m such that

$$\frac{It}{m} = \frac{N}{k-1} + 1 \tag{3}$$

System logic is implemented in the MMU , which serves as an interface between the memories and the rest of the components, keeping them independent from the parallelism degree and the iteration number. An example of data-flow in memories is shown in Fig. 4. We consider a kernel $k = 3$ and $N = 4$ MAC units, thus 6 memory columns are needed. In the first iteration, memories 1 to 6 are loaded with the new batch, and the result is stored in memories 1 to 4. In the next iteration, data from a new batch is loaded without overwriting memories 5 and 6. After processing step, the data is stored, starting from memory 5 up to the last one and then overwriting previous batch data from the beginning.

The MMU keeps track of the memory positions where the input batch must be stored, the information that must be fed to every MAC unit, the memory position where

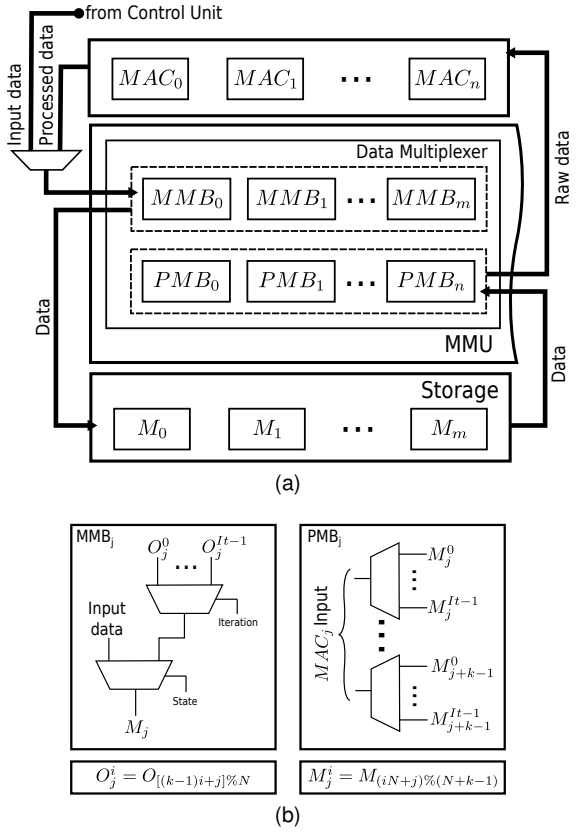

Fig. 5. (a) The data flow between the MACs and the storage. (b) The internal structure of MMB and PMB.

the processed data must be stored and the order in which the processed data must be returned, for each iteration. To do so, it has a finite state machine where each state corresponds to a set of the aforementioned positions. The number of states is given by the iteratiotn number in eq. (3).

The MMU has a set of multiplexers whose select lines are managed by the finite state machine (Fig. 2) to route the incoming and outgoing data. The data multiplexer is conformed by a set of smaller blocks which are classified into two classes according to their function: memory multiplexer blocks (MMB) and processing multiplexer blocks (PMB). The MMB routes the raw data and the data from the MACs to a memory, whereas the PMB routes the data from the memories to a MAC. The information flow between memories→MMU→MACs→MMU→memories is shown in Fig. 5. Both the MMBs and PMBs have a number of inputs proportional to the number of states from the MMU FSM and their multiplexers inputs are defined respectively as

$$O_j^i = O_{[(k-1)i+j]\%N} \tag{4}$$

$$M_j^i = M_{(iN+j)\%(N+k-1)} \tag{5}$$

where the symbol $\%$ represents the module operation and i goes from 0 to $It-1$.

A comparison between the naive (which assigns k independent memory columns to each MAC unit), the shared inputs and the shared inputs with circular shift approaches is shown in Fig. 6. Figure 6a shows the amount of memory required as a function of the parallelism degree

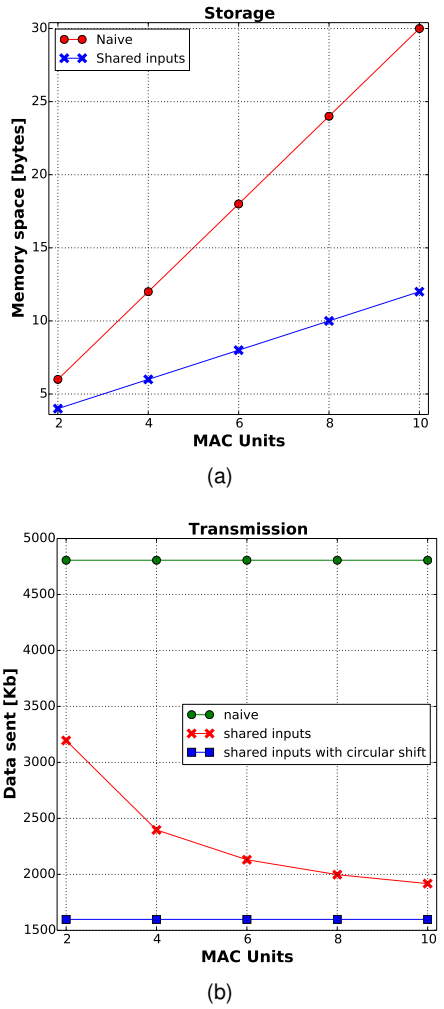

Fig. 6. Comparison between different approaches. (a) Amount of memory required. (b) Amount of data transmitted for an image of 1600×1024px and a 3×3 kernel.

whereas Fig. 6b shows the amount of transferred bytes (considering that every transmitted byte is stored in only one memory register) for processing a 1600×1024 image and a 3×3 kernel.

As shown in Fig. 6b, with the naive approach the transferred data is almost 3 times the image size (because a 3×3 kernel) and it does not get affected by the parallelism degree. The shared inputs approach tends to diminish the redundant data transferred as N increases, that is because the redundant data in a $N + k - 1$ batch is $k - 1$ and $\frac{k-1}{N+k-1}$ tends to zero as N increases. The shared inputs with circular shift approach is the best case scenario where no redundant data is transferred at all.

4. Implementation and Results

The design was implemented in a Xilinx Artix-35T FPGA (xc7a35ticsg324-1L) using Xilinx Vivado Design Suite 2017.4 tools. A 100 MHz reference clock was synthesized. The script for pre-processing and post-processing was written in Python 2.7.

In the first instance the desired kernel coefficients are loaded followed by the corresponding image batch via UART. Although UART is slow with respect to the processing time, it is used because of its simplicity and low

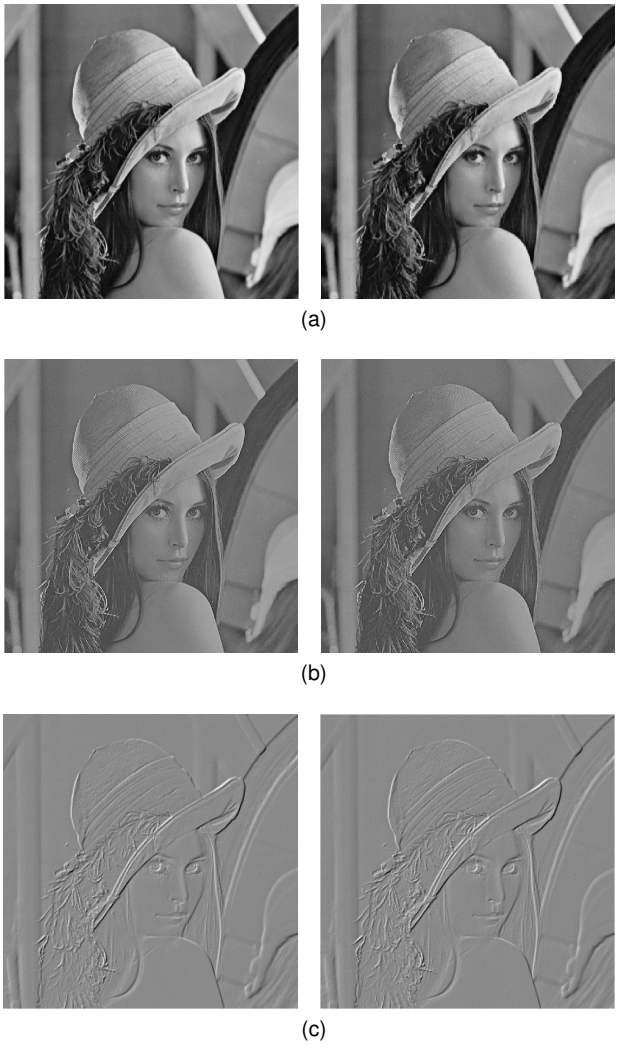

Fig. 7. On the left, an image processed in a CPU using python and on the right the same image processed with the FPGA module. The filters used are (a) Identity Kernel, (b) Sharpening Kernel and (c) Embossing Kernel.

resource consumption compared with other transmission media. As far as data representation is concerned, fixed point arithmetic was used with a resolution of $U(8,0)$ for input and output image and $S(8,7)$ for kernel.

To ensure the proper behavior of the architecture, multiple kernels were coded in python and applied to an image, and then the same kernels were applied using the FPGA module. The kernels used were identity $[0,0,0;0,1,0;0,0,0]$ sharpening $[0,-1,0;-1,5,-1;0,-1,0]$ and embossing $[-2,-1,0;-1,0,1;0,1,2]$. The results are shown in Fig. 7.

The architecture was synthesized for different degrees of parallelism. Table 1 shows module and microprocessor complexity on the FPGA with and without DSP respectively, meassured by resource utilization. As a result, a linear increase is shown according to parallelism. Due to implemented FPGA limitations a parallelism up to 8 was achieved using DSP and up to 24 without DSP but with a more intensive LUT usage.

A comparison between the estimated BRAM utilization and the results obtained from synthesis is shown in Fig. 8. Normalization was made with respect to BRAM

TABLE 1. RESOURCE UTILIZATION TABLE WITH AND WITHOUT DSP.

	With DSP [N](%)			Without DSP [N](%)		
P	**DSP**	**LUT**	**BRAM**	**DSP**	**LUT**	**BRAM**
2	20(22)	1845(9)	10(20)	—	3168(15)	10(20)
4	40(44)	2022(10)	11(22)	—	4627(22)	11(22)
6	60(67)	2175(10)	12(24)	—	6063(29)	12(24)
8	80(89)	2448(12)	13(26)	—	7756(37)	13(26)
10	—	—	—	—	9328(45)	14(28)
12	—	—	—	—	10917(52)	15(30)
24	—	—	—	—	20209(97)	21(42)

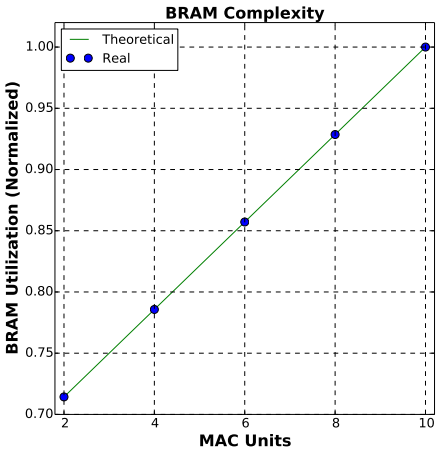

Fig. 8. Normalized curve for BRAM resources utilization.

TABLE 2. THROUGHPUT ACHIEVED IN CONVOLUTION PROCESSING.

Parallelism	Processing Speed [Mp/s]
2	200
4	400
8	800

utilization percentage dividing by the maximun utilization percentage value, and considering that microblaze instance already occupies 18% of BRAM resources. It can be appreciated that the linear behaviour anticipated by theoretical calculations is consistent with the synthesis results.

Considering only convolution operation processing, i.e. not taking into account the time needed to load the image into memory, one pixel per clock cycle (100 Mhz) is obtained per instantiated MAC unit. Thus, throughput increases linearly with parallelism degree. Table 2 shows throughput obtained for different parallelism degrees.

5. Conclusion

Using FPGA we are able to process the filtering at the same time as reading the current image batch. In this paper, we have presented the implementation of two-dimensional convolution on a Xilinx Artix 7 FPGA platform based on resource efficiency and system parallelism. We implemented a whole image processing system taking into account load stage, processing stage, and output stage. Moreover, a relation between instantiated BRAM blocks and MAC units was found in the presented architecture,

which allows our system to work with different parallelism degrees.

In addition, we optimized the use of memory resources implementing an algorithm for memory operations and module synchronization. Performances and results show that resources utilization concerning BRAM resources increase linearly, as desired.

High throughput was achieved in what processing is concerned. On the other hand, the limiting factor that impacted the implemented system the most was UART speed transmission. Another limiting factor was the DSP48A1 slices. Both factors are easily solved by using an FPGA with more slices and resources.

Acknowledgments

The authors would like to thank PhD. Mario R. Hueda and PhD. Damian Morero for supporting their design through Fulgor fundation research program.

References

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

[2] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 161–170. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689060

[3] A. Gupta and V. K. Prasanna, "Energy efficient image convolution on fpga," University of Southern California Los Angeles, USA, agrimgupta92@gmail.com, prasanna@usc.edu, Tech. Rep., 2011.

[4] B. Cope, "Implementation of 2d convolution on fpga, gpu and cpu," UImperial College London, benjamin.cope@imperial.ac.uk, Tech. Rep., 2010.

[5] L. kabbai, A. Sghaier, A. Douik, and M. Machhout, "Fpga implementation of filtered image using 2d gaussian filter," *International Journal of Advanced Computer Science and Applications*, vol. 7, no. 7, pp. 514–520, 2016.

[6] H. Ström, "A parallel fpga implementation of image convolution," Master's thesis, Department of Electrical Engineering, Linköping University, 2016.

[7] D. Ramírez, R. Romero, and F. Santa, "Parallel processing on fpga for image convolution using matlab," Master's thesis, Universidad Distrital Francisco José de Caldas, Bogotá, Colombia, 2015.

[8] A. M. Saxe, P. W. Koh, Z. Chen, M. Bhand, B. Suresh, and A. Y. Ng, "On random weights and unsupervised feature learning," in *ICML*. Omnipress, 2011, pp. 1089–1096.

[9] R. C. González and R. E. Woods, *Digital Image Processing*, 3rd ed. Pearson Prentice Hall, 2008.

[10] W. Rudin, *Principles of Mathematical Analysis*, 3rd ed. McGraw-Hill, 1976.