

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

Martin Casabella, Sergio Sulca, Ivan Vignolles, Ariel L. Pola

Escuela de Ingeniería en Computación

Facultad de Ciencias Exactas Físicas y Naturales

Universidad Nacional de Córdoba

Email: martin.casabella@gmail.com, ser.0090@gmail.com, ivignolles@alumnos.unc.edu.ar, arielpola@gmail.com

Abstract—In this article, a hardware architecture for 2D convolution implementation on FPGA when there is not enough RAM memory for instantiate and store a complete image, is presented. 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.

1. Introduction

The origin of digital image processing, due to the high level of processing they require, is directly related to the development and evolution of computers. Most filters for images 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. Applications recently, use convolutional neural networks (CNN) [1] where a fast and efficient convolution implementation is important.

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 accelerate the calculation. In addition, FPGAs are reconfigurable, allowing the flexibility that is often desired in neural networks. This coupling of processing, parallelism 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 the FPGA area. As shown in [3], block random access memory (BRAM) is one of the most used elements and consume more power. Therefore, the architectures seek to reduce the use of BRAMs by dividing them into two groups that consider the total 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 convolvers are used [6],

[7]. The difficulty presented by these schemes is the non-modularization of processing, making it difficult to increase the work rate by increasing parallelism.

In this document, we propose a high-performance, parallelizable hardware architecture and dynamic memory reuse with a linear increase in BRAM resources. In addition, theoretical estimates of complexity are compared with synthesis results in FPGA. The remainder of this paper is as follows. Section 2 introduces the image processing algorithm. The architecture design of image filtering is presented in section 3. In section 4, hardware implementation and experimental results are shown. Finally, a conclusion will be done in section 5.

2. Image Processing Algorithms

Filtering is widely used in many applications, such as aeronautics, navigation, CNN, etc. They are applied as pre-processing to eliminate useless details and noise. One of the preferred devices for the development of these techniques is the FPGA, since it presents an excellent relationship between speed of development and flexibility in the implementation. It allows the parallelization of the process and the reconfiguration of the design in a very short time. We will focus on the efficient use of FPGA resources by applying the 2D convolution technique and reducing the dynamic range of the pixels in the image.

2.1. Convolution Operation

Bidimensional convolution is defined mathematically as

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

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 $(h \times h)$ and $S(x, y)$ is convolution result of size $(m - 2 \times n - 2)$ pixels because we are considered valid convolution [8].

2.2. Change of Dynamic Range

We will focus on grayscale images because of their simplicity of implementation, but the design can be extrapolated

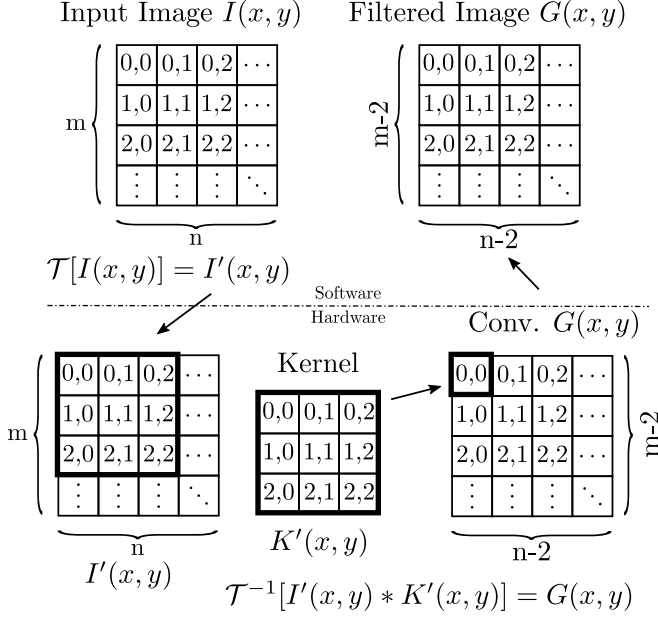


Fig. 1. Transformation steps during processing.

to any other type of 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 dynamic range centered on zero. Therefore, we apply dynamic range expansion [9] and maximum norm [10] to rearrange $\mathcal{T}[I(x, y)] = I'(x, y)$ between $[0, 1]$ and $\mathcal{T}[K(x, y)] = K'(x, y)$ between $[-1, 1]$, respectively. Therefore, replacing in (1)

$$G(x, y) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} K'(i, j) I'(x - i, y - j). \quad (2)$$

Finally, processed image reconstruction is only needed since $G(x, y)$ values are already rearranged in FPGA. The complete process is detailed 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 detailed transformation in Section 2 using a Python script. In addition, the script divides the image into batches and is sent by the UART port to the FPGA. In the FPGA a microprocessor receives the information and communicates it to the convolution module using a 32 bits general purpose input-output (GPIO) port. This port writes the BRAMs that store the data that will be filtered. Due to the limitations of the size of the FPGA, no other interface was used, such

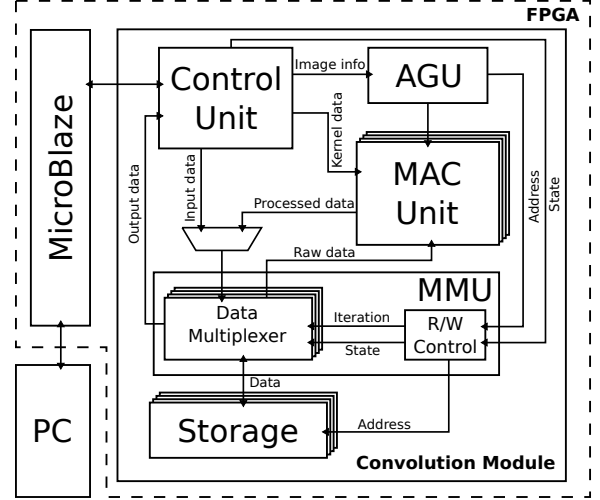


Fig. 2. Simplified block diagram of the convolution module.

as Ethernet or direct memory access (DMA) 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 the user establishes through the GPIO. When the operation ends, a notification is sent to the microprocessor, which gives the order to recover the processed batch of the module and sends it to the central processing unit (CPU). Finally, the post-processing stage joins 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 **multiple accumulator (MAC) unit** is where the sum of the products of the kernel's coefficients with the pixels happens. The **address generation unit (AGU)** manages the memory addresses along with the **memory management unit (MMU)**, that manages how the values in memory must be 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. It is possible to see that during the work cycle, the convolution module goes through several states, as shown in Fig.3. States transition are controlled by coded instructions embedded in the 32 bits input frame, and the module will 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, to do so it must receive a reset signal.

After receiving correspondent coded instruction, its will switch to the **KernelLoad** state, in which the kernel coefficients are loaded into the module. Then it goes to the **SizeLoad** state where the image's height is loaded, the

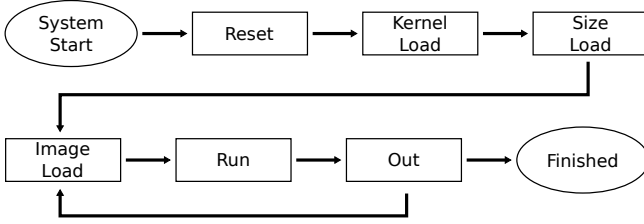


Fig. 3. Module's state diagram.

argument for doing this is explained in section 3.2.2. Once the module has gone through this setup states it will not come back to them until the entire image is processed.

To begin with the processing loop, first the module goes to the **ImageLoad** state and stores the batch in memory. Then it goes to the **Run** state where the processing occurs, and while processing is being made the module can not be interrupted, i.e. it will ignore all instructions till the batch is processed completely. As soon as processing step is ended, a notification will be emitted through Control Unit's *EoP* pin and the module will wait for **Out** state switch instruction, state where the already processed batch is sent back to the microprocessor.

All this states logic is managed by the control unit, which depending on the state will communicate the corresponding control signal to the other components of the module.

3.2.2. Data storage. The kernel coefficients are stored in registers inside each MAC unit. In order to store a batch, the proposed approach is to arrange the block RAM memory in columns, where each of them will store a image's pixel column. Therefore a batch size is given by the height of the image and the number of memory columns, also the maximum height of the image will be restricted by the number of entries that each memory columns has.

During a 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 reduces drastically the amount of memory needed 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. Given a $k \times k$ kernel, each MAC unit takes k adjacent memory columns as input. To produce the first processed pixel, a MAC unit first needs to load k pixels from each columns in such a way as to have $k \times k$ pixels loaded into it, then proceeds to multiply the pixels with the kernel's coefficients and sum everything. Finally, it truncates the result to reduce the bit length and the result is saved into the first position of the memory.

Once a pixel is processed, it is saved into 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, like a FIFO structure. This procedure

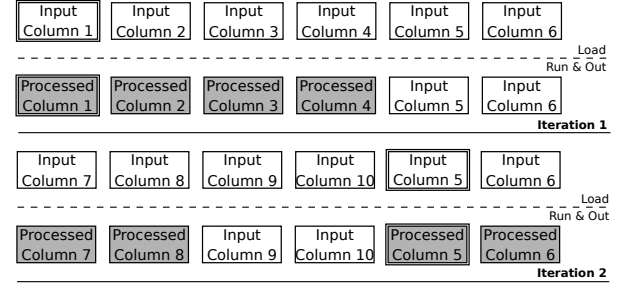


Fig. 4. Memory writing process using circular shift for $N = 4$ and $k = 3$.

is equivalent to shift the kernel vertically on the image. All this steps are synchronized in such a way that one processed pixel is generated in every clock cycle.

The AGU manages the memory's read and write addresses, and it takes into account the latency between the clock cycle where the first pixel in the memory is read and the clock cycle where the first processed pixel is written in memory. This latency is equal to k plus some more clock cycles needed to latch values during the convolution, and the difference between the read address and the write address must be equal to it.

Until 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, that means that $2k$ columns are needed for two MAC units. However, given the nature of the convolution operation, to produce a contiguous column it is necessary to shift the input columns by one. Hence, there is an overlap between the two inputs and this produces that even k input columns are still needed per MAC unit, only $k+1$ different input columns are needed for all the units. For this reason, it was decided that multiple MAC units in parallel will produce contiguous processed columns, sharing the common input columns. With this approach, for N MAC units the number of memory columns needed gets reduced from $N \cdot k$ to $N + k - 1$, concluding that adding a new MAC unit only adds one new memory columns.

The same overlap explained above produces also that repeated data will be needed between one batch and the next one. In order to eliminate the need to send data that is already in memory, some data is reused between iterations. Given N convolution units and a $k \times k$ kernel, a $N + k - 1$ batch width is needed, but the processed batch has a width of N columns, i.e. one for each MAC unit. Therefore the last $k-1$ memory columns are not overwritten and maintain the input data. This $k-1$ columns will be reused as the first columns of the next batch, thus the batch width is reduced to N , with the exception of the first batch that will still have a $N + k - 1$ width.

A consequence of reusing the memory columns written by the previous batch is that in every iteration the position of memory columns associated with each MAC unit describes a circular shift by N places. From the above it follows that a

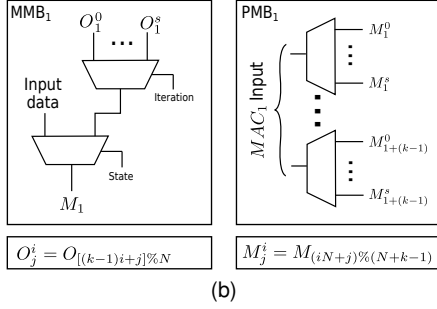
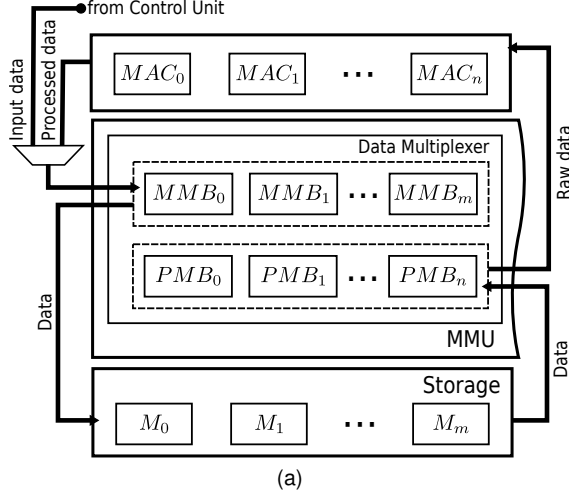


Fig. 5. (a) shows the data flow between the MACs and the storage. (b) shows the internal structure of MMB and PMB.

periodicity in the relationship 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 relationship, i.e. when $It(k-1)$ is a multiple of $N+k-1$. That is, there must be an integer m such that

$$\frac{It}{m} = \frac{N}{k-1} + 1 \quad (3)$$

The 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. Fig. 4 shows how the memories are written using this logic.

The MMU keeps track of the memory positions where it must save the input batch, the information which must be fed to every MAC unit, the memory position where 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 positions named above. The number of states is given by eq. 3.

To route the incoming and outgoing data, the MMU has a set of multiplexers whose select lines are managed by the finite state machine (Data multiplexer and FSM respectively in Fig. 2). The data multiplexer is conformed

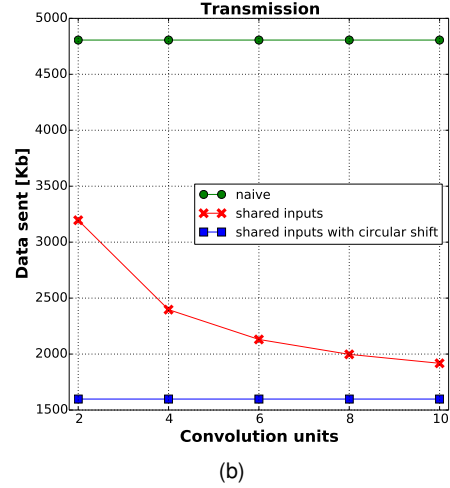
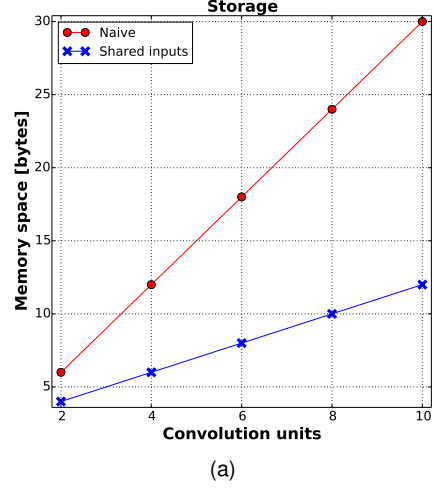


Fig. 6. Comparison between different approaches. (a) Amount of memory required. (b) Amount of data transmitted for an image of 1600×1024 px and a 3×3 kernel.

by a set of smaller blocks which are classified into two classes depending on 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, while the PMB routes the data from the memories to a MAC. Fig. 5 shows the information flow between memories \rightarrow MMU \rightarrow MACs \rightarrow MMU \rightarrow memories. Both the MMBs and PMBs have a number of inputs dependent on the number of states from the MMU's FSM and its represented by the letter s in the figure.

A comparison between the naive (which would be assign 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 in function of the parallelism degree while Fig. 6b shows the amount of transferred bytes (considering that every byte transmitted is stored in only one memory register) for processing a 1600×1024 image and a 3×3 kernel.

TABLE 1. RESOURCE UTILIZATION TABLE WITH AND WITHOUT DSP.

P	With DSP [N](%)			Without DSP [N](%)		
	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)

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 an Xilinx Artix-35T FPGA (xc7a35ticsg324-1L) using Xilinx Vivado Design Suite 2017.4 tools. A 100 MHz reference clock frequency was synthesized. The client where the pre and post-processing occurs was written in Python 2.7.

In the first instance the desired kernel's 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 resource consumption compared with other transmission medias. 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 functioning 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. As it can be seen, processed image has good quality with all kernel filters tested.

The architecture was synthesized for different degrees of parallelism, Table 1 shows the resources consumed by the module and the microprocessor on the FPGA with and without DSP respectively. As awaited, a linear consumption increase is shown according to parallelism degree. Due to limitations of the FPGA used a parallelism degree up to 8 was achieved using DSP and up to 24 without DSP but with a more intensive use of LUT.

A comparison between the estimated BRAM utilization and the results obtained from synthesis is shown in Fig.



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

8. Normalization was made with respect to BRAM utilization percentage dividing by maximum utilization percentage value, and considering that microblaze instance already occupies 18 % of BRAM resources. As wanted, utilization increment linear behaviour was achieved.

Considering convolution operation processing, one pixel per clock cycle is obtained, per instantiated MAC unit, i.e. if 16 MAC unit were instantiated, with a reference clock at 100 MHz, 1.6 Gigapixels/second (Gp/s) throughput is achieved. In addition, regarding load and read stage, where one pixel every two clock cycles is load, 23.2 Megapixels/second (Mp/s) throughput is obtained.

5. Conclusion

Using FPGA we are able to process the filtering at the same time of reading the current image batch. In this paper, we have presented the implementation of two-dimensional

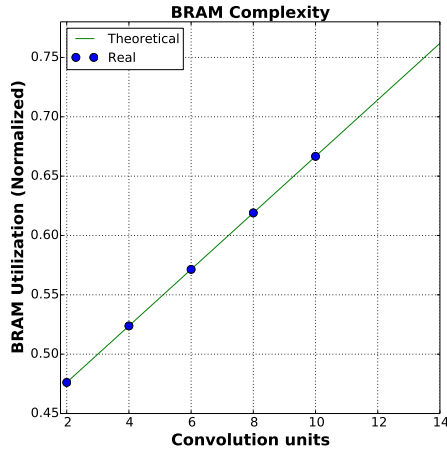


Fig. 8. Normalized curve for BRAM resources utilization.

convolution on a Xilinx Arty 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 relationship between instantiated BRAM blocks and MAC units was found in the architecture presented, which allows our system work with different parallelism degrees.

In addition, we optimized the use of memory resources implementing an algorithm used not only for reading and writing operations but also interaction between implemented modules. 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 mentioned above are easily solved by using an FPGA with more slices and resources.

Acknowledgments

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] A. D. Leila kabbai, Anissa Sghaier 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] R. R. Diego Ramírez 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. E. W. Rafael C. González, *Digital Image Processing*. Prentice Hall, 2007.
- [10] W. Rudin, *Principles of Mathematical Analysis*, 3rd ed. McGraw-Hill, 1976.