Efficient VHDL Implementation of an Upscaling Function for Real Time Video Applications

Pablo Rubio-Ibáñez, José Javier Martínez-Álvarez, Ginés Doménech-Asensi
Dpto. de Electronica, Tecnología de Computadoras y Proyectos
Universidad Politécnica de Cartagena
Cartagena, Spain
pablo.rubio@upct.es, jjavier.martinez@upct.es, gines.domenech@upct.es

***Abstract*— This paper describes the design and the implementation of a high-performance area-efficient upscaling function on a FPGA. The proposed function performs the resizing of real time video images, from M x N pixels to 2M x 2N pixels, using a bilinear interpolation method. The hardware implementation has been carried out using an RTL structural description in VHDL, to maximize the use of the specific resources of the FPGA, especially its memory resources (BRAM). For this purpose, an optimized memory access schema, that allows the best performance of the architecture with the least use of hardware resources, is proposed. Details of the design and the implementation on a Zynq UltraScale+ MPSoC are given, as well as speed-performance and area utilization. The proposed solution significantly improves the results achieved using this schema compared with those obtained using the Resize function from the Xilinx OpenCV library, when both are configured to obtain the same spatial transformation.**

Keywords— FPGA; BRAM; Upscaling; Optimized Memory Access; Resize

I. Introduction

Digital image upscaling is the method of generating an image of larger size, with a significantly higher number of pixels, from a low resolution image. Video upscaling is considered a common operation, of high practical significance, which is used to improve the image quality and processing performance. Upscaling is used in many areas of application, such as medical imaging processing, remote sensing, video surveillance, computer vision and so on, to increase the actual resolution of videos to extract relevant sub-pixel information from images.

There is a large number of methods to perform image upscaling. Different types of interpolation algorithms and their hardware architectures (VLSI) are analysed and compared in [1]. Interpolation methods can be mainly divided into two categories: adaptive and non-adaptive algorithms. Non-adaptive methods are simplest; they perform interpolation by averaging neighbouring pixels using the same fixed linear filter for every pixel. This operation inevitably smooths the discontinuities along object edges and details of the images, causing undesirable blurring and aliasing effects such as artifacts or jagged-edge. On the other hand, adaptive algorithms usually make the most of spectral information present in the neighbourhood pixel to perform the interpolation and reduce or eliminate the aliasing effects. These methods provide more accurate images at the expense of increased computational and memory requirements. This makes them less suitable for real-time or low-cost applications.

For the convenience of implementation, as well as fast computation, some applications recommend using the so called cheap interpolation algorithms to upscale images. This is the case of the SIFT algorithm [2], where a computationally low cost bilinear interpolation is used to double the size of the input image to compensate for the prefiltering needed and reduce the noise or aliasing in the incoming image. The interpolation separates the interest points, providing more distinguishability. By implementing this technique, the number of features detected in the SIFT algorithm increases by a factor of 4. Another interesting application that uses cheap interpolation is the RAISR algorithm [3] developed by Google. The main idea behind RAISR is to increase the quality of a computationally low cost upscaler (e.g. bilinear) by applying a set of pre-learned filters on the image. The algorithm uses machine learning and train on pairs of images, a low resolution (LR) one, and a high resolution (HR) one, to find filters that, when applied to the upscaled image, will reconstruct details that are of comparable quality to the original. The training method involves, first, applying a computationally cheap upscaler to double the size of the LR image and then training the filters from the LR and HR image pairs. Since the input images are already upscaled to begin with, the training phase takes much less time to come up with a filter that accurately approximates the actual image. Finally, at run-time, from a given LR image, it is produced its HR approximation by first interpolating it, using the same cheap upscaling method that is used in the learning stage, followed by a filtering step with the pre-learned filter. Computationally inexpensive interpolation can be useful in many applications to improve the quality of images, including ANN [4], deep learning, CNN and so on.

In this paper, we propose a design to implement a computationally cheap upscaling function that increases the size of an incoming image from M x N pixels to 2M x 2N pixels using a bilinear interpolation method. In particular, the main contribution of this work is the design of an efficient architecture for resizing images in real-time, in which the computational requirements have been minimized and the use of the specific memory resources optimized. The rest of the paper is organized as follows. Section II describes the interpolation approach proposed for the upscaler function. In Section III, the main features, and details of the architecture and the memory access schema are presented. In section IV, the implementation results and a comparative study are shown. Finally, Section V concludes the paper.

II. Interpolation approach

The more basic methods to perform upscaling are the nearest-neighbour, bilinear and bicubic [5] interpolation. The simplest case is the nearest-neighbour approximation, where the value of a new pixel is determined by the value of its nearest pixel. Although this method is the fastest and does not require any calculations, the approximation obtained is not optimal because it produces large amount of blocking artifacts and jagged-edges in the image. A better quality of image is achieved by using bilinear and bicubic interpolations. These methods provide a continuous transition among the pixels to be interpolated, which is more desirable. The bilinear method calculates the new pixel by averaging the value of its nearest 2 x 2 neighbourhood, while in the bicubic method, the approximation is obtained from a third-order polynomial, considering the nearest 4 x 4 neighbourhood of each pixel. Generally, although bicubic algorithm achieves better quality images, it is computationally slower and requires more complex calculations, which is why, in this work, the bilinear approach has been used.

There exist different hardware architectures to perform real-time bilinear interpolation, such as [6] and [7]. Most of these solutions propose efficient designs for the implementations of this technique. However, because they work on any resolution at any scaling factor, they require specific hardware resources to calculate the value of each new pixel at any location. In the general case, the value of each new pixel is determined by (1), where α and β are, respectively, the horizontal and the vertical distance between the LR pixel $P_{(i,j)}$ and the target pixel $P_{(i-\alpha, j-\beta)}$. Since α and β can take any value from 0 to 1, several multiplier operators are required to compute the bilinear interpolation.

$$P_{(i-\alpha, j-\beta)} = (\alpha \;\; 1\text{-}\alpha) \cdot \begin{pmatrix} P_{(i\text{-}1,\, j\text{-}1)} & P_{(i\text{-}1,\, j)} \\ P_{(i,\, j\text{-}1)} & P_{(i,\, j)} \end{pmatrix} \cdot \begin{pmatrix} \beta \\ 1\text{-}\beta \end{pmatrix} \quad (1)$$

In this work, in order to achieve an efficient and computationally cheap interpolation function, the general expression (1) has been restricted to the case of calculate only double-sized images, i.e. parameters α and β only can take two possible values $\{0, 0.5\}$. This consideration allows reducing the computational requirements of the hardware and hence improves the trade-off between image quality and computational efficiency. As shown in Fig.1, the proposed approach determines both vertical and horizontal interpolation by averaging only two pixels of the LR image, while four pixels are required for central interpolation. In every case, the interpolation process does not need multipliers because when α and β are 0.5 and so, the multiplications can be replaced by simple right shift operators.

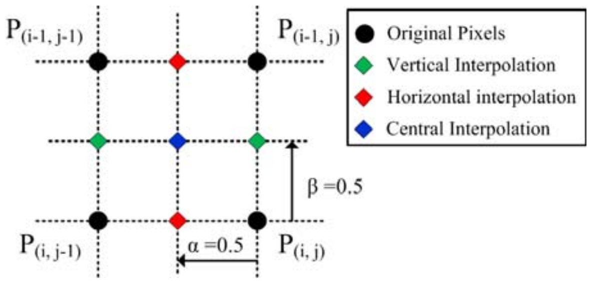

Fig. 1. Interpolation grid used by the proposed upscaling function.

III. Architecture and Memory Access Schema

In this section, the proposed design for the upscaling function and the access schema to the memory will be described.

A. Hardware Architecture

The hardware architecture of the upscaling function includes a two-line buffer, an interpolation unit, two counters, a shift register bank and two state machines which control the operation of the system. All of these components carry out a series of tasks that, combined together, allow the generation of resized images in real-time. The complete architecture of the system is shown in Fig. 2.

The two-line buffer has been designed using a 1K x 36-bit data width BRAM. A maximum of 18 bits per pixel has been assumed so that the BRAM can store the two lines of the image that are required to perform the interpolation in real time. This configuration allows to work with images with a maximum resolution of 1024 pixels per column (2048 pixels for the resized image) and no restrictions on the maximum resolution per row. The BRAM has been configured in read-first mode, so that, when a read/write operation occurs simultaneously at the same position, the memory register is first read and then overwritten with the input data. Pixels are read/written at CLK frequency through Port A, while output pixels are read through Port B at 4 x CLK frequency. As pixels enter to the circuit, row by row, the current input row is stored in the 18-LSB part of the BRAM and the previously stored line is fed back and stored, through Port A, in the 18-MSB part of the BRAM.

Read and write operations are addressed by two counters, the Column Counter A (CCA), that addresses the Port A, and the Column Counter B (CCB) which addresses the Port B. Output pixels are transferred to the interpolation unit through Port B in order to carry out the calculations of bilinear interpolation (1). The operations performed by this unit depend on the pixels involved in the interpolation (vertical, horizontal or central) as well as the on the boundary conditions. The boundary conditions are only applied to compute the last row and the last column of upscaled image. The last row and column of the image will be identical to the previous ones respectively (e.g. for a 1280x960 resolution upscaled image, the row 1280 will be equal to the row 1279 and the column 960 will be equal to the column 959).

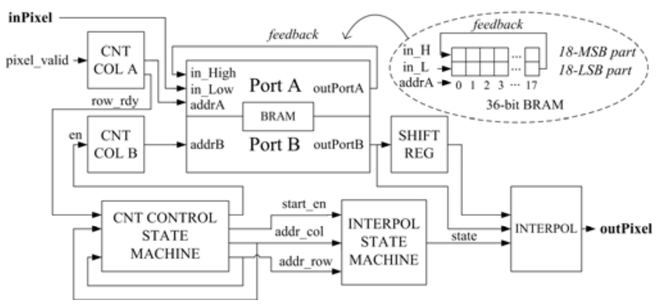

Fig. 2. Architecture of the upscale function..

When the Interpolation Unit computes a central interpolation, four pixels of the LR image will be required. In order to have all these pixels available in the same clock cycle, the Shift Register Bank provides the pixels $P_{(i-1,j-1)}$ and $P_{(i,j-1)}$, which were stored in the previous clock cycle, while the pixels $P_{(i-1,j)}$ and $P_{(i,j)}$ are provided directly through Port B. The Counter Control State Machine starts working when CCA sends a flag confirming that first row of the input image has been stored in the BRAM. The main task of this State Machine is to control the CCB and two built-in counters. These two internal counters are used by the Interpolation State Machine to obtain the type of interpolation required in each case (horizontal, vertical or central) and apply the boundary conditions.

B. *Memory Access Schema*

Fig. 3 shows an example of the interpolation grid for a 20 column input image and an upscaled 40 column output one. In each pixel or cell, the upper value represents the row in the image and the lower one the column. Pixels of the input image are grayed out, and are represented by integer coordinates. Pixels of the output image are those placed in odd rows and columns, and are represented by fractional coordinates. The interpolation algorithm begins once the initial row (row 0) has been written in memory (Fig. 3.a). Then, the interpolated values for row 0 are computed and, at the same time, the pixel values of this row are sent to the output. (Fig. 3.b). During this operation, five input pixels are written in row 1. Note that the output throughput is 4 times faster than the input one. The timing diagram for this operation is detailed in Fig. 4.a. Next, the rest of pixels of row 1 are stored in memory and row 0.5 is sequentially interpolated (Fig. 3.c). The timing diagram for the generation of this odd row is detailed in Fig. 4.b. This sequence (Fig3.b – Fig 3.c) is repeated for all the rows of the image. Thus, the pixel output rate is given by the frequency of CCB.

As shown in Fig.5, when a new incoming pixel is stored in a register of the 18-LSB part of the BRAM, the pixel previously stored in that register is shifted to next register in the 18-MSB part through Port A (feedback). However, the result of this behavior will be that pixel positions of both lines stored in BRAM will not be synchronized (the line stored in 18-MSB part will be shifted to the right). To solve this, bits read from the 18-LSB part through Port B are delayed one clock cycle in order to match the bits of 18-LSB part read through Port B.

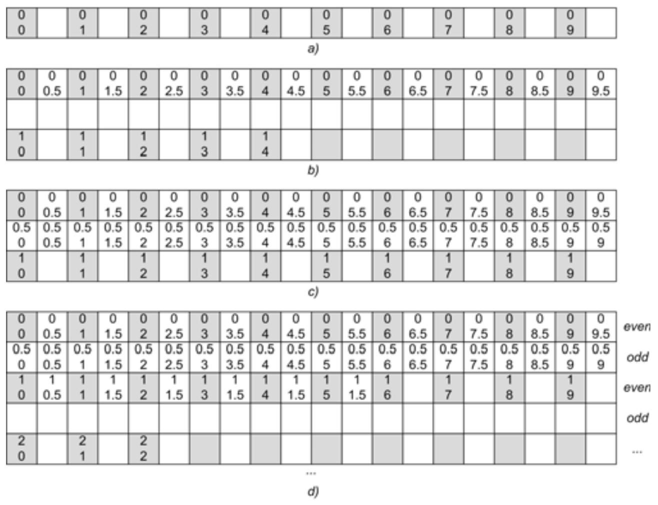

Fig. 3. Interpolation grid for a 20 column input image and a 40 column output one. a) Write first row (row 0) b) Write first half of row 1 and output row 0 c) Write second half of row 1 and output row 0.5 d) Start writting first half of row 2 and output of row 1.5.

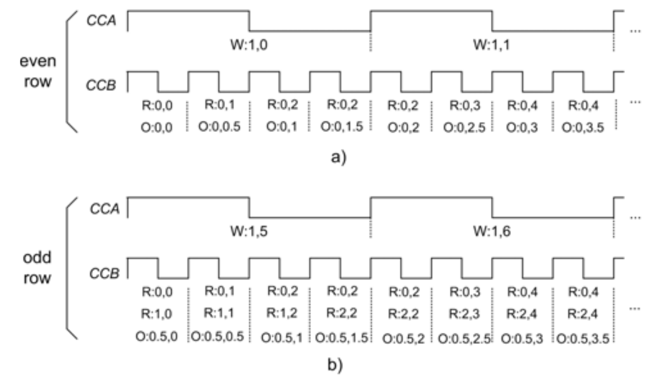

Fig. 4. Timing diagram of read/write and pixel output operations. R: read, W: write, O: output a) Output of an even row b) output of an odd row.

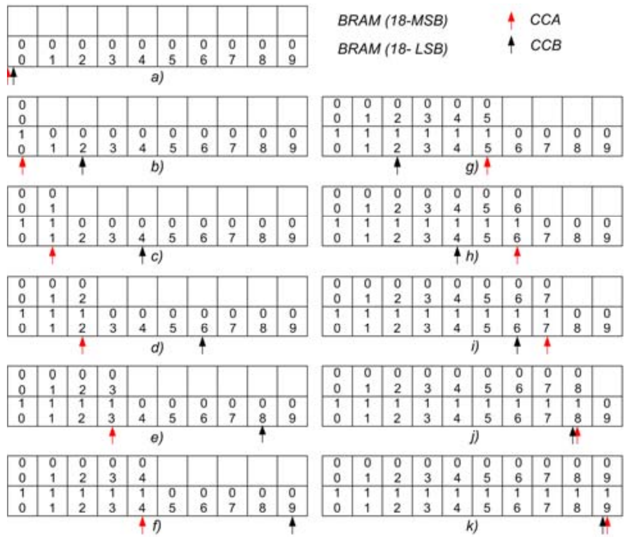

Fig. 5. Memory management. a) First line written in 18-bit LSB BRAM. b) to f) During generation of even row g) to k) During generation of odd row

.

TABLE I. USAGE OF HARDWARE RESOURCES

Input Image Resolution	This Work					xfOpenCV (resize function)				
	LUT6	FF	BRAM (36K)	DSP	Freq (MHz)	LUT6	FF	BRAM (36K)	DSP	Freq (MHz)
640x480	326	147	1	0		5224	3591	4.5	12	
1280x720	408	133	2	0	480	5263	3691	6.5	12	300
1920x1080	490	134	2	0		5267	3629	7.5	12	

IV. RESULTS

The proposed architecture has been implemented in a Xilinx Zynq Ultrascale+ MPSoC platform (XCZU9EG-2FFVB1156E FPGA with a grade speed of -2) at a work frequency of 480 MHz. Table I shows the resources utilization of this implementation for 18-bit pixel precision with different input image resolutions. The use of resources is compared to those ones achieved by the *resize* function included in the xfOpenCV library [7].

Table I shows that, in spite of the fact that the proposed architecture works with a higher frequency than the *resize* function, the resource utilization is considerably lower. However, the resize function is a generic implementation designed to upscale or downscale images using different multiplying factors. On the other hand, the *resize* function has some limitations for the type of data that can be used. In the architecture proposed in this work, there is no limitation of pixel bit width because it has been developed so that it can be defined by parameters without range restrictions. However, optimal results are achieved when pixel width is set to 18 bits. When comparing the latency of both works, it can be noted that the results are quite similar. For example, for a VGA 640x480 input image, the proposed architecture needs 2,566 clock cycles to produce the first pixel of the upscaled image and 1,231,366 clock cycles to complete the operation on the whole image, whereas *resize* function produces the result after 1,247,794 clock cycles. At a frequency of 480 MHz this is a throughput of 384 fps. For input images of 1280x720 pixels and 1920x1080 pixels, the upscaling rates achieved using the proposed architecture are, respectively, 130 fps and 57 fps. Compared to the work described in [6], where a bilinear interpolation VLSI hardware implementation is detailed, the prosed architecture is much simpler, given that it has been optimized for applications where parameters α and β only can take two possible values $\{0, 0.5\}$. This allows to use only 7 adders and no multipliers, while the architecture described in [6] uses 4 multipliers and 36 adders.

Fig. 6 shows an example of an image upscaled using the proposed architecture. The image has been taken from the Oxford affine covariant regions dataset [8]. Figs. 6.b and 6.c show, respectively, a detail of a 26x26 pixels input image patch and the corresponding upscaled 52x52 pixels patch.

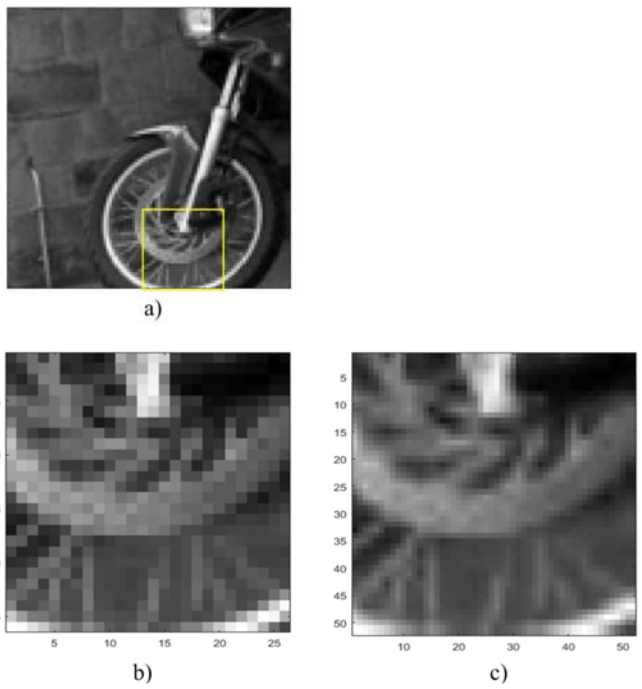

Fig. 6. Detail of upscaling. a) Original image b) 26 x 26 pixels input patch taken from a). c) 52x52 pixels upscaled output patch.

V. Conclusion

In this paper an efficient and computationally cheap upscaling hardware architecture to perform bilinear interpolation has been presented. The proposed architecture imposes a very low usage of the available FPGA resources while still achieving good performances and an excellent trade-off between image quality and calculation requirements. To achieve this performance with a low area, a tailored interpolation approach was used together with a careful memory access schema to assure an efficient memory usage and the minimum computational complexity of the system. The architecture has been implemented in VHDL code using a structural RTL description. The proposed circuit has been completed and tested on a Zynq UltraScale+ MPSoC, demonstrating that real-time bilinear interpolation is achievable for different standard resolutions. The evaluation of the proposed architecture and its implementation shows very good results compared to other existing proposals, for a double-size upscaling operation. Due to its very few computational requirements and its real time capability, the proposed architecture can be considered a competitive solution for those applications that require the use of upscaling in real-time with the minimum cost in hardware.

Acknowledgements

This work has been partially funded by Spanish government projects RTI2018-097088-B-C33 and TEC2015-66878-C3-2-R (MINECO/FEDER, UE).

References

[1] C. John Moses, D. Selvathi and V. M. Anne Sophia, “VLSI Architectures for Image Interpolation: A Survey,” VLSI Design, vol. 2014, 2014, pp. 1–10.

[2] D. G. Lowe, “Distinctive Image Features from Scale-Invariant Keypoints,” Int. J. Computer Vision, vol. 60, no. 2, pp. 91–110, 2004.

[3] Y. Romano, J. Isidoro, and P. Milanfar, “RAISR: Rapid and Accurate Image Super Resolution,” arXiv:1606.01299, June 2016.

[4] M. S. Hasan and S. T. Haque, “Single Image Super-Resolution Using Back-Propagation Neural Networks,” 20th International Conference of Computer and Information Technology (ICCIT), 2017, pp. 1–5.

[5] M. Keys, “Cubic Convolution Interpolation for Digital Image Processing,” IEEE Transactions on Acoustics, Speech and Signal Processing, vol. 29, no. 6, pp. 1153–1160, 1981.

[6] P. A. Dilip, K. Rameshbabu, K. P. Ashok and, S. A. Shivdas, “Bilinear Interpolation Image Scaling Processor for VLSI Architecure,” International Journal of Reconfigurable and Embedded Systems, vol. 3, no. 3, pp. 104–113, 2014.

[7] Xilinx, ”Xilinx OpenCV”, User Guide UG1233 (v2017.1), July, 2018.

[8] [Online]. Available: http://www.robots.ox.ac.uk/~vgg/data/data-aff.html