Architectural Exploration of an FPGA-based Hardware Accelerator for the Gaussian Filter using Approximate Computing

Guilherme A. M. Sborz, Felipe Viel, and Cesar A. Zeferino
Laboratory of Embedded and Distributed Systems – LEDS
University of Vale do Itajaí – UNIVALI
Itajaí – SC, Brazil
sborzguilherme@edu.univali.br, {viel, zeferino}@univali.br

***Abstract*— The growing use of computer vision applications has increased the demand for efficient image processing implementations. These applications have constraints that, in some cases, can only be met by dedicated hardware implementations. This work presents architectures that apply approximate computing techniques to improve efficiency and scalability for implementing digital image filters on FPGA. These architectures were implemented as hardware accelerators for an embedded processor in an FPGA-based System-on-Chip. The results show that the use of approximate computing techniques can reduce costs without affecting results for the target application, which is an essential feature for further acceleration using parallel processing on hardware.**

***Index Terms*—Image Processing, Computer Vision, Approximate Computing, Hardware Accelerator, FPGA.**

I. Introduction

Recent advances in computer vision techniques and their use in applications such as object detection [1], motion tracking [2], and semantic segmentation [3] have increased interest in solutions that employ digital image processing (DIP) techniques. These applications require technologies and techniques for rapid prototyping, low power consumption, and low latency. Hardware accelerators (also known as custom processors and co-processors [4]) can be used as an alternative to software implementations to achieve these goals. Another technology typically used for rapid prototyping is Field-Programmable Gate Array (FPGAs), which is an integrated circuit (IC) technology that offers a good trade-off among the design metrics offering the lowest time-to-market in comparison with other IC technologies [5].

The development of hardware accelerators in FPGA can be done by exploring different architectures. Aspects such as the arithmetic used [6], approximate computing techniques (ACTs) [7], and hardware/software communication architecture [8] are some of the features and techniques that can be explored during the implementation of DIP algorithms on hardware. In this context, assessing possible implementation models and comparing their impact on system metrics is essential to assist hardware designers in making decisions about their designs.

DIP applications are good candidates for acceleration by custom processors. As discussed in [9], low-level operations that enable the exploration of parallelism are ideal for hardware implementation. Furthermore, given the vast volume of data to be processed, a software implementation is usually slow and unable to fulfill time requirements. In this sense, several works describe the implementation of DIP algorithms on FPGA. For instance, in [10], the authors present a hardware accelerator for the Sobel edge detection algorithm. They assessed the number of logical resources occupied and the quality of the output images. In another work [11], the authors propose a novel Canny edge detection algorithm and implement it on FPGA. They apply approximate methods for computing gradient magnitude and orientation, aiming at reducing costs. The authors assess the similarity between the proposed algorithm and a conventional Canny implementation, evaluate the occupancy of logical resources, and compare the FPGA performance with those of GPU-based deployments. In [12], the authors present an FPGA implementation of a multidirectional Sobel operator for edge detection. They demonstrate that an eight directional approach is more effective in detecting edges than the traditional Sobel. Results are limited to resource usage and visual comparison of the two approaches to a single image. Even though these works implement DIP algorithms on FPGA, they did not investigate the impact of ACT techniques employing a diversity of evaluation metrics.

Given the context above, this work explores the design space for implementing hardware accelerators for digital image filtering on FPGA by employing two ACTs [7]: precisions scaling and memoization. The former consists of decreasing the number of bits used to represent data, thus reducing the requirements for processing and storing data. The latter works by storing the result of functions in local memories, called look-up tables (LUTs). This technique is useful for applications in which the number of function entries is limited

This work was supported by CAPES – the Brazilian Federal Agency for Support and Evaluation of Graduate Education – Finance Code 001 and CNPq – the Brazilian National Council for Scientific and Technological Development – Processes 315287/2018-7 and 436982/2018-8.

as it determines the size of the memory that will be used. The reader must take care to do not confuse the name of this technique with the word.

In this work, we also evaluate the impact of different architectures on performance and costs indicators of the hardware accelerator. As a case study, we chose the Gaussian filter because it is a classic low-level DIP algorithm and has characteristics similar to several other image filters (e.g., Sobel and Moving Mean). Comparisons among different implementation models are made through the quality of result (QoR), the occupancy of logic resources, performance metrics, and energy consumption. Thus, the contribution of this work is the characterization of possible implementation models for low-level DIP operations using ACTs, as well as demonstrating the impact of each architectural solution in the primary design metrics.

The remainder of this paper is organized as follows. Section II provides background on the Gaussian filter. Section III describes the architectures proposed for the Gaussian filter. Section IV presents and discusses the materials and methods employed in the tests and the results obtained from the experiments. Finally, Section V gives the final remarks.

II. The Gaussian Filter

The Gaussian filter is used for image smoothing [13], and its primary use is as an initial stage of edge detection algorithms, such as Canny, Sobel, and Laplace. These algorithms are sensitive to noise, and an algorithm to reduce these distortions, such as the Gaussian filter, is essential for its operation.

Smoothing filters inhibit the passage of the high-frequency components of an image, functioning as low-pass filters. These components represent, among other characteristics, the contours of the objects in the scene—the more abrupt the change of direction of the contour, the higher its frequency [14]. Therefore, smoothing filters tend to smooth these transitions, serving, for instance, for the reconstruction of incomplete contours caused by distortions resulting from low resolution.

As discussed in [13], the application of the Gaussian filter in digital images is made from the discretization of the continuous 2-D Gaussian function, described by (1). This discretization enables the formation of a mask (or kernel) of size $n \times n$, with radially symmetric coefficients. The obtained approximation is defined from the following parameters: (i) σ, the standard deviation of the Gaussian function, and (ii) n, the height and the width of the mask. The latter is responsible for defining the precision of the filter since it limits the number of values used to represent the Gaussian function.

$$f(x,y) = \frac{1}{2\pi\sigma^2} e^{(-\frac{x^2+y^2}{2\sigma^2})} \quad (1)$$

Another important characteristic of the Gaussian filter is its possibility of separation. As described in [15], *separable filters* enable masks of size $n \times n$ to become two masks of sizes $n \times 1$ and $1 \times n$. This property divides the convolution in two operations, which allows the exploration of temporal parallelism since the second operation can start before the end of the first.

III. Architecture

A. Convolution hardware

The main block of the Gaussian filter hardware accelerator is shown in Fig. 1. The control block comprises counters and comparators responsible for generating the control signals. The delay line buffer is responsible for storing and sorting the pixels of the window to be rendered. Finally, the processing block executes the arithmetic operations between the pixels of the current window and the filter coefficients (i.e., it executes the convolution operation).

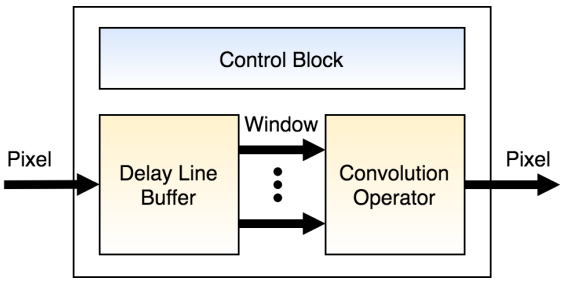


Fig. 1: Convolution datapath.

The delay line buffer is a memory that explores the characteristics of storage and transmission of images and the local properties of the sliding window operation to reduce the number of accesses to memory. Typically, a digital image is saved in a memory named frame buffer with its pixels stored from left to right and from top to bottom. The access to the pixels happens sequentially, with one pixel transferred at each clock cycle. A convolution operation occurs by moving a window (i.e., a small kernel such as a 3×3 square) over the image in the same direction in which the image is accessed. Taking advantage of these principles, a delay line buffer stores the last values read in shift registers. Thus, the pixels that will still be used in some convolution window are stored locally by this buffer. Fig. 2 illustrates the internal structure of a delay line buffer for a 3×3 window. It comprises nine 1-pixel registers to store the nine pixels of the current window and two row buffers that store the remaining pixels of the first two rows of the current sliding window.

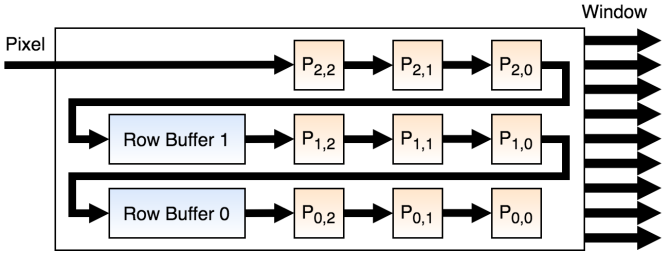


Fig. 2: Delay line buffer.

The different architectures for implementing the Gaussian filter consist of variations of the processing block of Fig. 1 – the convolution operator. The following subsections describe

the four variations implemented in this work, which are based on architectures described in the literature.

1) 2-D Gaussian: The first architecture, illustrated in Fig. 3, uses a multiplier for each filter coefficient and an adder tree to compute the sum of products and generate the output pixel. The adder tree enables several sums to be performed in parallel and the use of pipeline stages to break the critical path and increase performance. As Fig. 3 shows, the nine pixels of the 3×3 window are multiplied by a 3×3 mask composed of the filter coefficients, i.e., the weights $w_{i,j}$, where $i = 0..2$ and $j = 0..2$.

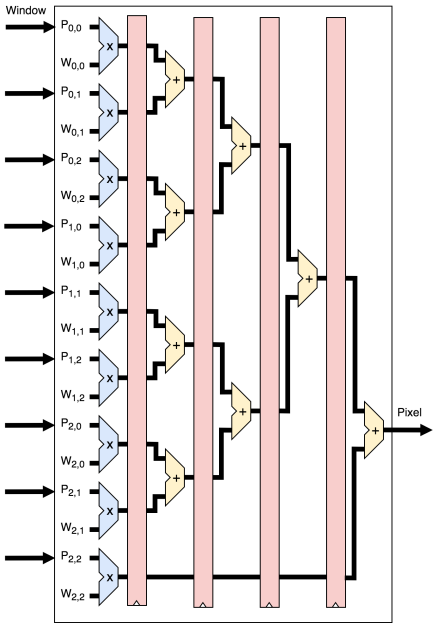


Fig. 3: The 3×3 2-D Gaussian filter.

2) Modified Gaussian: This architecture considers a particular feature of the Gaussian filter, which repeats some coefficients used to multiply the values of the current window. This feature makes it feasible to reorder the operations so that the pixels multiplied by the same values are added before multiplication. As a result, this approach enables reducing the total number of multipliers required, thus saving logic resources. Fig. 4 illustrates how this architecture is implemented to process a 3×3 window. Considering that the four pixels at the corners of the window are multiplied by the same coefficient (w_c), and the four pixels in the neighborhood of the central pixel are also multiplied by the same coefficient (w_n), this architecture requires only three multipliers to execute the convolution on a 3×3 window. However, using this technique can generate overflow, which can be avoided by using a larger data word. Therefore, the costs regarding the adders are higher than in the 2-D Gaussian filter.

3) Separate Gaussian: Another feature of the Gaussian filter, which is found in other low-level processing filters, is that it can be applied separately. For instance, the same result of applying a 3×3 window can be achieved by horizontally convolving a 1×3 window with the input image, followed

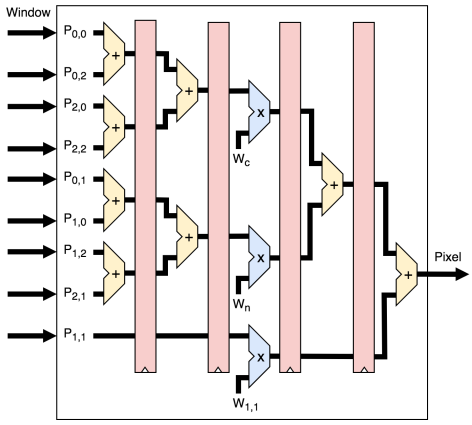


Fig. 4: The 3×3 Modified Gaussian filter.

by a vertical convolution over the image resulting from the horizontal convolution. The architecture developed for this implementation model is illustrated in Fig. 5. It comprises two 1-D Gaussian filters. It executes an 1×3 horizontal convolution followed by a 3×1 vertical convolution. The main difference among the two filters relies on the delay line buffer since the vertical filter uses a 1×3 buffer, whereas the vertical filter uses a 3×3 buffer. As in the previous architecture, it is necessary to use a data word larger than that of the 2-D Gaussian filter to avoid overflow.

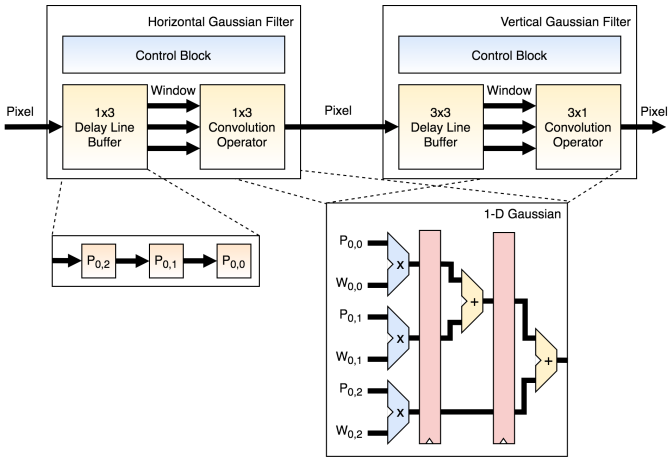


Fig. 5: The 3×3 Separate Gaussian filter.

4) LUT-based Gaussian: The fourth architecture uses the *memoization* technique, and the multipliers of the Gaussian filters are replaced by look-up tables. These tables store all possible results of the multiplication between a pixel and a given coefficient. As the input image is composed of 8-bit pixels, a 256-entry LUT is implemented to replace each multiplier of the convolution operator. It is worth noting that these LUTs store constant values and can be implemented using ROMs. This approach results in greater cost savings than using RAMs, for example. Fig. 6 exemplifies how the 2-D Gaussian filter is implemented using LUTs to replace the multipliers.

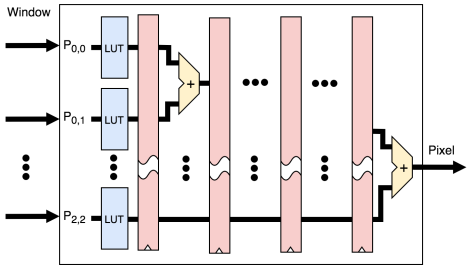


Fig. 6: The 3×3 LUT-based Gaussian filter.

B. Communication

The transfer of data between the FPGA and the ARM processor of the System-on-Chip (SoC) uses direct memory access (DMA) employing the Intel DMA Controller® FPGA IP core. The hardware architecture of the implemented communication model is shown in Fig. 7. As we can see, the Hard Processor System (HPS) uses bridges to communicate with the input and output DMA controllers for writing the image into the input frame buffer and reading the filtered image from the output frame buffer, respectively.

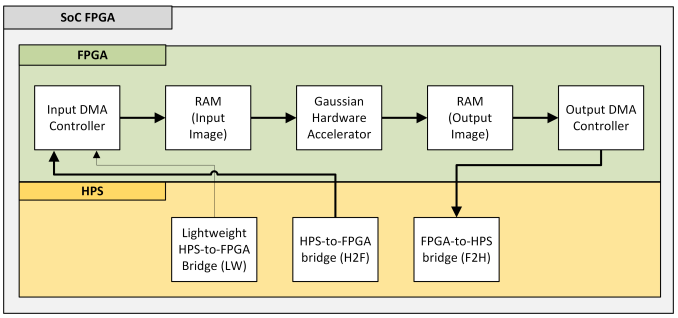
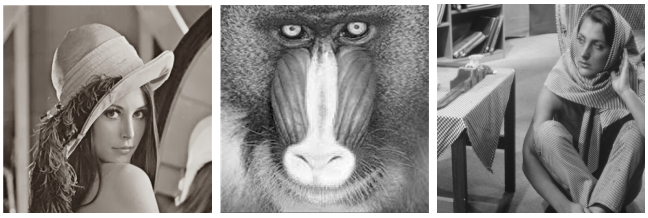


Fig. 7: Hardware/Software Communication.

IV. Implementation and Results

A. Materials and Methods

First, we implemented the Gaussian filter in Python using floating-point and fixed-point arithmetic. The goal was to identify the impact of the use of the precision scaling technique (described in Section I) on the resulting image quality. This evaluation was made for data words composed of an 8-bit integer part and different sizes for the fractional part (the precision). We then performed the tests on the three 512×512 grayscale images shown in Fig. 8, which are classic images for testing and evaluation of DIP algorithms. In both images, we used virtual borders and applied the Gaussian filter using a standard deviation of 1, as well as the kernels most used in this kind of application (i.e., 3×3, 5×5, and 7×7). The metrics used to assess the quality loss were Normalized Mean Square Error (NRMSE) and Peak Signal-to-Noise Ratio (PSNR). RMSE identifies the similarity of two images, and the closer RMSE is to zero, the higher is the similarity. PSNR expresses the ratio between the maximum power of a signal and the power of the noise that affects the signal representation. Images with PSNR above 20 db are considered acceptable [16].



(a) (b) (c)

Fig. 8: Image data set: (a) Lena; (b) Baboon; (c) Barbara.

Next, we described a parameterizable synthesizable VHDL model of each architecture designed for the Gaussian filter. After description, we evaluated the need for larger data words to avoid overflow in the Modified and Separated Gaussian filters. Experiments demonstrated that the integer part of the Modified Gaussian filter should equal 10 bits for the 3×3 window size, and 11 bits for the 5×5 and 7×7 sizes. Experiments also showed that the Separated Gaussian filter requires a 9-bit integer part for all the three window sizes. With this approach, all the hardware implementations produce results identical to those generated by the software fixed-point implementation.

The synthesizable models were validated through simulations using ModelSim simulator and employing physical prototyping on the Cyclone V SoC 5CSEMAF31C6N device from Intel®FPGA. This device integrates 3,972 Kbits of embedded memory, 32,075 ALMs (Adaptive Logic Modules), and 87 multipliers (DSP blocks). Each ALM is composed of an 8-input fracturable look-up table – LUT with four dedicated flip-flops (or registers). The reader should not confuse the LUT of the FPGA with the LUT of the memoization technique. While the former is a 2^m-bit RAM block (where m is the number of inputs), the latter is a table of constants that can be implemented in the FPGA using a ROM or logical equations by defining an equation for each bit of the output data word.

After verification, we assessed the costs (occupancy of logic resources and power dissipation) and performance (maximum operating frequency and processing latency) using the toolset provided with the Intel® Quartus® Prime software suite (version 18.1). From these indicators, we computed the throughput, energy consumption, and power efficiency.

Following, we evaluated the impact of hardware/software communication on the response latency for sending and receiving images and the acceleration of the hardware to a software-based implementation running on the hard-core processor embedded on the FPGA device. These tests were done using a smaller version of the input image (100×100) due to the memory constraints of the FPGA used. The communication tests were performed on the Terasic DE1-SoC development kit, which contains the above FPGA with a Dual-Core ARM® Cortex™-A9 Hard Processor System (HPS) embedded on the FPGA. In this experiment, we did not use the NEON unit available in the Cortex™-A9. This unit provides a single-

instruction multiple-data (SIMD) instruction set to accelerate media and signal processing applications. This assessment will be done in future work.

B. Experimental Results

Table I summarizes the results obtained from the experiments performed to assess the quality loss. It presents the average of the indicators measured for the three images of the data set. As we can see, using only four fractional bits is sufficient to generate a PSNR greater than 20 (lower precisions resulted in a PSNR below 20db). We can also observe that the 3×3 kernel produces better quality indicators than the larger kernels because the accumulated error rises when the number of multipliers increases, indicating more similarity with the image represented using floating-point. It is worth noting that we do not consider the impact of the precision or the window size in the noise removal of a noisy input image. Usually, the wider the window, the better the filtering result. Evaluating the quality of the filtering process with different window sizes is outside the scope of this work.

TABLE I: Fixed-point versus floating-point representation

Format	3×3		5×5		7×7	
	NRMSE	PSNR	NRMSE	PSNR	NRMSE	PSNR
Q8.4	0.06	29.86	0.06	29.77	0,06	29.75
Q8.6	0.02	41.91	0.03	35.88	0.03	36.59
Q8.8	0.00	81.55	0.01	44.40	0.01	44.48

Table II presents the costs and performance results obtained for the 3×3 window using the 8-bit data precision, which was the one with the best quality of results. Concerning the silicon costs, we can see that the Modified and Separate architectures use wider data words to avoid overflow, resulting in a higher occupancy of ALMs than in the 2-D Gaussian filter. We also note that the LUT-based filter is the architecture that occupies more ALMs because it replaces the DSP blocks by ROMs (the memoization's LUTs), which are implemented through logic equations in the ALMs. The number of DSP blocks in the first three architectures varies with the number of multipliers used, and the FPGA utilized can share the same block by two operations. This feature is the reason why the number of DSP blocks is a bit smaller than the number of multipliers of each architecture (as it was shown in Fig. 3, 4, and 5).

TABLE II: Results for a 3×3 window and 8-bit precision

Metric/Architecture	2-D	Modified	Separate	LUT-based
Data format	Q8.8	Q10.8	Q9.8	Q8.8
Number of ALMs	217	258	237	273
Number of DSP blocks	8	3	5	0
F_{max} (MHz)	242	266	271	249
Latency (ms)	1.092	0.993	0.975	1.062
Throughput (frames/s)	916	1007	1026	942
Power (mW)	471.52	475.52	516.32	475.45
Energy (μJ)	514.75	472.25	503.22	504.84
Efficiency (kframes/s/W)	1.94	2.12	1.99	1.98

Regarding performance, all the architectures take a similar number of clock cycles to process the image (about 264 Kcycles). However, the Modified and Separate architectures achieved higher operating frequency and throughput. This result is probably due to the balance between the number of ALMs and DSP blocks occupied, which can facilitate placement and routing by the compiler, thus producing a shorter critical path.

Concerning the power costs, we observe that the Separate architecture dissipates more power than the other filters because it has two delay line buffers and control blocks, which increase the switching activity. We also observe that the Modified Gaussian filter consumes less energy and has a higher power efficiency than the other architectures because it offers the best trade-off between performance and power dissipation.

Table III presents the costs and performance results obtained for the 3×3 window using the 4-bit data precision, which was the one that obtained the lower acceptable PSNR. As expected, the reduction of data precision enables obtaining lower silicon costs because fewer ALMs are necessary, higher performance (clock frequency and throughput) as the critical paths are shortened, lower power costs (power and energy) because the switching activity and the processing latency are reduced, and higher power efficiency (an average of 7%) in comparison with the 8-bit precision. This approach is a viable solution for silicon- or power-constrained designs.

TABLE III: Results for a 3×3 window and 4-bit precision

Metric/Architecture	2-D	Modified	Separate	LUT-based
Data format	Q8.4	Q10.4	Q9.4	Q8.4
Number of ALMs	187	212	213	201
Number of DSP blocks	8	3	5	0
Fmax (MHz)	248	278	268	265
Latency (ms)	1.064	0.949	0.988	0.996
Throughput (frames/s)	940	1054	1013	1004
Power (mW)	452.62	456.51	487.05	469.56
Energy (μJ)	481.56	433.12	481.01	467.82
Efficiency (kframes/s/W)	2.08	2.31	2.08	2.14

Table IV presents the costs and performance obtained for the 7×7 window using the 8-bit data precision– the configuration expected to have the higher costs because it is designed to process the larger window and the highest precision among those evaluated in this work. By comparing these results with the data presented in Table II, we can observe the impact of increasing the window size to the indicators of the hardware accelerator. First, the 2-D Gaussian filter has poor scalability as it consumes much more DSP blocks than the other architectures. Also, the larger the window size, the higher is the number of ALMs occupied. Regarding performance, we observe a degradation of the maximum operating frequency and throughput due to the longer wires resulted from placing and routing a larger circuit. The power dissipation is smaller because the lower operating frequency reduces the switching ratio. On the other hand, the energy is higher, and the power efficiency is lower than those of the filters designed to work with the 3×3 window due to the lower throughput.

TABLE IV: Results for a 7×7 window and 8-bit precision

Metric/Architecture	2-D	Modified	Separate	LUT-based
Data format	Q8.8	Q11.8	Q9.8	Q8.8
Number of ALMs	676	558	476	547
Number of DSP blocks	46	6	11	0
Fmax (MHz)	225	240	254	258
Latency (ms)	1.190	1.116	1.058	1.038
Throughput (frames/s)	840	896	946	963
Power (mW)	469.95	471.46	492.74	473.78
Energy (μJ)	559.25	526.28	521.09	491.87
Efficiency (kframes/s/W)	1.79	1.90	1.92	2.03

Fig. 9 summarizes the average indicators obtained for the four architectures for three different window sizes. For each indicator, the values are normalized regarding the maximum value measured. This chart enables better assessing the impact of the window size on the design metrics.

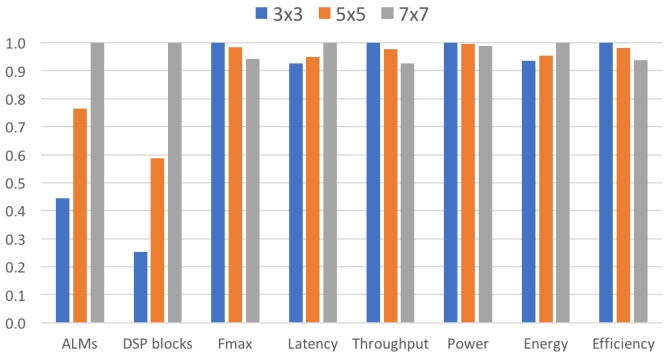


Fig. 9: Average indicators.

We also assessed the impact of the hardware/software interface in the total execution time of the system and the speedup. As mentioned before, we needed to downscale the image resolution to 100×100 pixels due to the memory constraints of the device available in the development kit used. Table V presents the comparison between the software implementation of the Gaussian filter running on the ARM processor and the hardware implementations for the smallest data word size of each architecture. The total latency includes the time to transfer the image between the ARM processor and the hardware accelerator (i.e., 130 μs) and the time to process it on the hardware accelerator. From the results, we can see that communication is responsible for more than 70% of the time spent with the co-processing. However, as the software execution is costly, the co-processing provides an acceleration higher than 200 times over the embedded software execution.

V. Conclusion

This work presented a study on the impact of using architectures and two ACTs on the costs and performance of an image filter implemented in FPGA. The results showed that the use of approximate computing techniques could reduce costs without affecting results for the target application, which is an essential feature for further acceleration using parallel processing on hardware. Also, the exploration of the filter features and parallelism enables to reduce costs and increase performance.

As future work, we intend to compare the performance and development effort of the filters implemented in FPGA with software implementations based on the NEON of the ARM Cortex-A9 processors integrated with the FPGA.

TABLE V: System acceleration

Platform/Architecture	Data Format	Latency (ms)	Throughput (frames/s)	Hardware Speedup
ARM/Software	32-bit IEEE 754	36.076	27.72	1
FPGA/2-D	8.4	0.1789	5591	202
FPGA/Modified	11.4	0.1726	5795	209
FPGA/Separated	9.4	0.1764	5668	204
FPGA/LUT-based	8.4	0.1730	5779	208

Note: ARM running at 800 MHz. Hardware accelerators working at their maximum clock frequency. Kernel size = 7×7.

References

[1] J. Redmon *et al.*, "You only look once: Unified, real-time object detection," in *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition*, 2016, pp. 779–788.

[2] A. Filippeschi *et al.*, "Survey of motion tracking methods based on inertial sensors: A focus on upper limb human motion," *Sensors*, vol. 17, no. 6, p. 1257, 2017.

[3] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition*, 2015, pp. 3431–3440.

[4] M. Dossis, "High level synthesis for embedded systems," in *Embedded Systems: Theory and Design Methodology*, K. Tanaka, Ed. BoD, 2012, ch. 16, pp. 341–266.

[5] F. Vahid, *Digital design with RTL design, VHDL, and Verilog*. John Wiley & Sons, 2010.

[6] F. Cabello *et al.*, "Implementation of a fixed-point 2D Gaussian filter for image processing based on FPGA," in *2015 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*. IEEE, 2015, pp. 28–33.

[7] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput Surv*, vol. 48, no. 4, p. 62, 2016.

[8] T. Li *et al.*, "Efficient parallel implementation of morphological operation on GPU and FPGA," in *Proc. of the IEEE Int. Conf. on Security, Pattern Analysis, and Cybernetics (SPAC)*. IEEE, 2014, pp. 430–435.

[9] D. G. Bailey, *Design for embedded image processing on FPGAs*. John Wiley & Sons, 2011.

[10] R. Menaka, S. Janarthanan, and K. Deeba, "FPGA implementation of low power and high speed image edge detection algorithm," *Microprocess Microsyst*, pp. 103 053–1–103 053–7, 2020.

[11] D. Sangeetha and P. Deepa, "Fpga implementation of cost-effective robust Canny edge detection algorithm," *J Real Time Image Process*, vol. 16, no. 4, pp. 957–970, 2019.

[12] Z. Xiangxi *et al.*, "FPGA implementation of edge detection for sobel operator in eight directions," in *IEEE Asia Pacific Conf. on Circuits and Systems (APCCAS)*. IEEE, 2018, pp. 520–523.

[13] C. Solomon and T. Breckon, *Fundamentals of Digital Image Processing: A practical approach with examples in Matlab*. John Wiley & Sons, 2011.

[14] R. Szeliski, *Computer Vision – Algorithms and Applications*, ser. Texts in Computer Science. Springer, 2011.

[15] A. Joginipelly *et al.*, "Efficient FPGA implementation of steerable Gaussian smoothers," in *Proc. of the 2012 44th Southeastern Symp. on System Theory (SSST)*. IEEE, 2012, pp. 78–82.

[16] D. Salomon, *Data compression: the complete reference*. Springer, 2004.