Journal of Engg. Research Vol. 8 No. (1) March 2020 pp. 179-197

Implementation of an Edge Detection Algorithm using FPGA Reconfigurable Hardware

Sa'ed Abed

Computer Engineering Department, College of Engineering and Petroleum, P.O.Box 5969, Safat 13060, Kuwait University, Kuwait
Corresponding Author: s.abed@ku.edu.kw

ABSTRACT

Digital image processing involves the manipulation of images via computer algorithms to enhance the images or classify targets and features. Some of these algorithms, such as edge detection algorithms, involve convolution operations, which require much computation. Generally, software running on a processor performs these manipulations. To achieve higher computation performance in terms of execution time, these algorithms are implemented on reconfigurable hardware such as field-programmable gate arrays (FPGAs). Parallel and pipelined architectures can be implemented on an FPGA to speed computations up. In this work, we provide a detailed description of implementing an edge detection algorithm on the SGI–RC100 platform. The implementation of the algorithm consists of two parts of code: the host program is implemented using ANSI-C and the Mitrion–C program. Mitrion–C offers efficient ways for writing code for parallel and pipelined architectures to preform edge detection. Next, the algorithm is tested on an Intel Intanium 2-based architecture, and its execution time is compared with the RC100 platform-based algorithm to check the speed gain achieved by the FPGA-based algorithm. The experimental results indicate that the speed of the reconfigurable hardware FPGA-based algorithm was over 50 times faster than that of the software-based approach.

Keywords: Convolution; Sobel filter; edge detection operator; FPGA; Gaussian filter; Mitrion–C; speed.

INTRODUCTION

Edge detection is a fundamental process that is used heavily in computer vision systems and has been considered a vital problem in image processing for more than 50 years. Gaining better results at different processing levels in computer vision depends greatly on good low-level feature extraction. Edge detection has widespread applications in the signal and image processing field areas such as image extraction, enhancement, and segmentation. Edges have massive intensity contrasts and can be defined as a sharp change in intensity. The process of detecting intensity transition locations is called edge detection. The amount of data that needs to be processed is reduced significantly by considering an image as the representation of edges in which important information concerning the image is still preserved. Moreover, edges can be used to describe an image and involved easily in image-processing algorithms and techniques (Basu, 2002). An image consists of pixels, and each pixel has an intensity. A pixel at the edge has a higher intensity than its neighboring pixels. Edge detection can be achieved by checking for discontinuities between a pixel's intensity and that of its neighboring pixel. There are many techniques used to implement edge detection algorithms (Rashmi *et al.,* 2013). In general, these techniques can be classified into two general categories: the gradient and Laplacian approaches (Shrivakshan *et al.,* 2012). The gradient technique is based on detecting the edge's variation in the first derivative of the image intensity between the neighboring values (maximum and minimum) and response to the peak value; hence, some operations are needed to suppress pixel weight (thinning). On the other hand, the Laplacian technique is based on detecting the image points in the second derivative of the image intensity by finding the zero crossings (Mintz, 1992; Rao *et al.,* 2004; Huertas *et al.,* 1986).

An edge detection algorithm involves two main steps: (i) Image smoothing is used to suppress the noise in an image. A noisy pixel has a higher intensity discontinuity with respect to its surrounding pixels. The edge detection process may detect a noisy pixel as an edge pixel if an image-smoothing operation is not performed prior to edge detection. Different masks are available to smooth the image. The Median, Gaussian, and averaging (Box) filters are all examples of masks that are widely used to smooth images. The filter operation is performed by moving the mask over an image and convoluting the mask and image. (ii) An edge detection operator is used to highlight edges that have sharp changes (discontinuities) in intensities and does so by applying a high-pass filter. It preserves the important parts of the image; therefore, less time is needed to process the result. The edge detection operator is used to perform this operation at each point of the image in order to estimate the magnitude of the gradient. Edge detector kernel-based (masking) algorithms, such as Sobel, Prewitt, Robinson, and other edge detectors, use a 3x3 pair of convolution masks for this estimation. One pair is used to estimate the gradient in the x-direction (columns), and the other pair is used to estimate the gradient in the y-direction (rows). Finding the gradient in the x-y direction is not an easy task.

In both steps mentioned before, a spatial operation in image processing, the so-called "convolution," uses the neighborhood of image pixels to apply a certain type of filtering. It is an important mathematical operation in the fields of signal and image processing. Convolution is performed between the mask and the part of the image covered by the mask. Mathematically, convolution is a process used on two continuous functions *f* and *g* to produce a third filtered function of *f* via Equations 1 and 2 below.

$$h(x,y) = f(x,y) * g(x,y) \tag{1}$$

$$h(x,y) = \iint_{-\infty}^{\infty} f(s,t) \cdot g(x-s, y-t)\, ds dt \tag{2}$$

Since we are dealing with a spatial filtering domain, producing discrete versions of Equations 1 and 2 is quite simply a matter of substituting summation notation for integral notation. Generally, the convolution of a pixel at location (*x*, *y*) in the image *G* using a mask *F* of size *K*×*K* is defined in Equation 3.

$$H[x,y] = \sum_{-K/2}^{K/2} \sum_{-K/2}^{K/2} F(S,T) \cdot G(x-S, y-T), \tag{3}$$

where *H* is the new image after filtering, and *S* and *T* are the kernel dimensions.

The image and filter are functions that are represented in 2D arrays in spatial domain filtering. The basic idea of convolution is to multiply two arrays of numbers (pixel values) and generate a new array of numbers (pixels). Equation 4 represents the weighted average of the pixel values at a point. The summation in the denominator represents the weighted values of the kernel and is called the "Normalization constant." This constant is used to create natural units of one gray-level per pixel. Literally, the normalization constant is used to preserve the color depth of the image. The new calculated pixel value in the output image depends on the mask applied. If the sum of the kernel`s coefficients is non-zero, then a normalization constant is needed to produce a maximum possible value within range of the image color depth.

$$p = \frac{\sum p_i w_i}{\sum w_i} \tag{4}$$

Spatial filters are commonly used to evaluate various outputs of different masks. Among these filters is the 2D convolution algorithm used in image processing. Its main purpose is to prop a fixed-size window of different weights and values assigned to each pixel, which is called a convolution mask. The convolution operation is then performed on each pixel of an image, as shown in Figure 1. The operation X in Figure 1 represents the dot multiplication per element (image convolution is basically a scalar product of the mask weights and all pixels of the image within a kernel). The image is convolved with a mask of radius 1 (one-ring neighborhood is deliberated throughout the convolution process). In some cases, the mask is centered at a location *(i,j)* of the input image where pixels of the original image at location *(i+a,j+b)* $\{-1 \le a, b \le 1\}$ are outside the image and can be treated as zeroes.

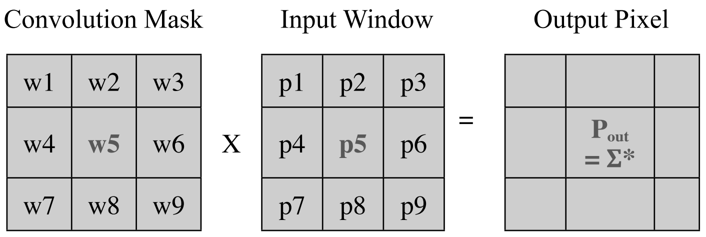

Fig. 1. Convolution operation.

For example, the 3x3 edge detector Sobel mask requires a total number of operations (C) of C*, which represents 9 multiplication operations + 1 addition operation for each pixel of the image in the edge detection step. The same number of operations is required in an image smoothing operation. Generally, a naïve 2D convolution algorithm takes $O(N^2)$. As the dimensions increase, k for example, the implementation becomes very slow and takes $O(N^{2k})$ (Yip *et al., 1999*). This algorithm can be implemented in a high-level programming language such as C/ C++, Java, and Python. The software runs on a General Purpose Processor (GPP) and detects the edges of an image. An GPP cannot perform such operations rapidly due to being limited by parallelism. Increased speed can be achieved by implementing the algorithm on reconfigurable hardware, such as a field-programmable gate array (FPGA), which offers different types of parallelism provided by spatial and pipeline architectures.

In this work, the implementation of the edge detection algorithm was performed using Mitrion-C (Mitrionics, 2001) and a Mitrion compiler targeted for an SGI Altix 350 (SGI Inc., 2006). The SGI Altix used here contained two RC100 FPGA Boards, and each board contained two Xilinx Virtex 4 LX200 FPGAs. The edge detection algorithm was also implemented via the ANSI-C language running on an Intel processor. The speed of the FPGA-based algorithm was measured and compared with the software-based implementation's speed. Our results showed that the reconfigurable hardware FPGA- based algorithm was 50 times faster than the software-based approach.

The rest of the paper is organized as follows. Section 2 presents the work closest to our own. Next, Section 3 describes the edge detection algorithm. The description of the implementation flow of the algorithm in Mitrion-C and the parallelization strategy are discussed in Section 4. Section 5 presents the comparison of the results obtained via the software-based implementation and the FPGA-based implementation. Finally, Section 6 concludes the paper and presents future trends.

RELATED WORKS

In this section, we review the edge detection work carried out on software-based implementations and hardware-based implementations (based mainly on FPGAs), while comparing between the two approaches and concentrating on the hardware-based approach.

Software-based approaches

In this subsection, MATLAB software and high-level languages, such as Java, the C family, Python, and other programming languages, are considered to be software-based approaches to implementing an edge detection algorithm. One approach for implementing and testing edge detection algorithms is to use MATLAB software for different platforms.

In Elaraby *et al.* (2013), the authors proposed a new edge detection algorithm based on both Tsallis and Shannon entropy. The algorithm showed a reduction in noise and decrease in execution time compared to the Canny, Laplacian of Gaussian (LoG), Sobel, Prewitt, and Roberts's algorithms. The performance results were based on a blood test image corrupted with (0-30)% salt and pepper noises. The simulation was carried out using MATLAB software without pre-processing ten times for all operators to measure the computation time. The proposed algorithm showed enhanced resilience with high-quality edge detection of noisy medical images. Furthermore, the implementation was simple.

Abed *et al.* (2017) proposed a multi-phase operator that consisted of two derived phases. The first phase of the proposed algorithm was dedicated to pre-processing, and the RGB color map image was converted to a grayscale image type using the NTSC weighting approach for further image processing operations. The second phase involved the edge detection process itself. An adaptive edge detection process was proposed in order to benefit from both the gradient and Laplacian approaches.

In Madhulika *et al.* (2013), the authors used three edge detection operators: Sobel, LoG, and Canny. They used a smoothing filter (high-pass filter) and then the infilter function in the MATLAB software. The Canny edge detector outperformed the other algorithms with its better feature extraction.

Topological gradient-based edge detection is presented in Samuel *et al.* (2015). The proposed method maximized the variance of the image by using the elliptic restoration equation for edge detection. A numerical comparison of the proposed detector with the Canny edge detector was included. The proposed detector was also compared with two other elliptic detectors and showed more robustness with respect to higher noise levels.

Edge detection based on the generalized structured learning approach was presented in Dollar *et al.* (2014). The proposed method took advantage of the intrinsic structure present in edge patches and was computationally efficient. A random forest framework was then used to capture the structures. The proposed approach used learning decision trees for structured labels to regulate the excruciating function on individual branches of the tree. Each forest foresaw a patch of edge pixel labels that were combined across the image to compute the final edge map.

Ant colony optimization-based image edge detection was proposed by Etemad *et al.* (2015). The method was proficient in carrying out feature extraction for edge detection and segmentation in images with noise. It was tested with different samples, and the results were compared to representative non-swarm-based methods. The advantages of the proposed method included lesser distortion when noise was added to the test images. Both qualitative and quantitative evaluations proved that the approach resulted in less distortion.

A new meta-heuristic approach for edge detection was presented by Xiaochen *et al.* (2015). The proposed method was based on ant colony optimization, used a heuristic function, and implemented a user-defined threshold in the pheromone update process. It had four main steps: the initialization, construction, pheromone update processes, and the end criterion. Evaluation results showed that the proposed method outperformed two other ant colony optimization-based edge detection techniques.

Edge detection with fuzzy cellular automata optimized via the Particle Swarm Optimization method (PSO) was investigated by Uguz *et al.* (2015). The paper proposed an efficient and simple thresholding technique for edge detection. The method converted neighborhood gradient magnitudes into membership degrees for fuzzy-cellular automata. The evaluation results showed that the method provided finer connectivity, leading to fewer false edges when compared with other techniques and removing misleading edges.

Hardware-based approaches

In this subsection, a hardware-based implementation (mainly via a FPGA) is introduced using different types of operators and methodologies. An FPGA is considered as an alternative to software-based methods for implementing scientific algorithms. FPGAs have a set of characteristics that make them unique, and their most important feature is their ability to change the operation of a device. Many approaches to edge detection have used FPGAs to enhance speed, time, power, or area features.

The authors in Rashid *et al.* (2014) proposed a Canny edge detection algorithm based on VHDL. The algorithm was designed to detect digital image edges automatically. It used Gaussian filtering to decrease the noise and combined MATLAB, Simulink, and XSG tools. The input image was a standard test image, and the output image consisted of a set of images in a certain phase until a Canny edge detected the image. The experiments carried out on medical images showed that the algorithm reduced the area and complexity of images in order to diagnose diseases.

Yasri *et al.* (2008) proposed a Sobel edge detection implementation based on an FPGA board. The algorithm was modeled as an FSM implemented in Verilog to run a matrix area gradient operation to evaluate the variance. The evaluation was done by measuring the differences in pixels and then displaying the results on a monitor. The experimental results showed that the proposed algorithm could operate at 27 MHz using a 720 x 720 pixel image and that the image could be detected within 2ms.

Another approach based on pipelining the Sobel edge detection algorithm via an FPGA was presented in Vanishree *et al.* (2013). The algorithm optimized the gradient-based edge detection method, which was able to decrease the delay by 50% compared to conventional techniques (Guo *et al.,* 2010) for an input image of 128x128 pixels loaded into the block memory. This approach can be used in high-speed applications such as a vehicle collision avoidance system.

Lavanya *et al,* (2014) proposed another optimized version of Sobel edge detection based on an FPGA. The high-level implementation of the algorithm was done on MATLAB, and the other parts were downloaded on a Spatran-3E board. When the algorithm was tested, it showed an enhancement of 40% of the area and 51% of the propagation delay, which required two cycles to determine the gradient in all directions.

Another architecture for Sobel edge detection based on an FPGA was proposed by Nausheen *et al.* (2018). They implemented the Sobel edge detection algorithm on hardware to show the benefits of parallelism over software in terms of computation. The proposed architecture performed at a faster frequency than existing designs while using less space complexity than other methods.

Bouganssa *et al.* (2017) presented a high throughput edge detection algorithm based on the discrete wavelet transform. They used filters with different directions to present the maximum existing contours of the image and downloaded the design to a Xilinx Sparten6 FPGA. The results showed less computation time and a low area for several images while maintaining the plasticity of the structure to support an adaptive system.

Singh *et al.* (2013) proposed a real-time colored edge detection algorithm to reduce the FPGA area usage using a Sobel operator. The algorithm was implemented using VHDL and downloaded on a Virtex-5 FPGA platform. The algorithm needed one Processing Element (PE) to calculate the gradients for the R, G, and B colors. The results showed that the algorithm achieved a video frame rate that was double the standard real-time one (25 fps x 2) for surveillance applications.

Real-time edge detection and range finding using FPGAs as presented in Khan *et al.* (2015). It applied a Sobel edge detector, making hardware implementation easier. Four blocks were used in the approach: 1) Image normalization: this block normalized the input image stream for a more comprehensive gradient image. 2) Sobel edge detector: this block was used to detect the edges. 3) Range sensor: this block was used to send a trigger out to the ultra-sonic ranger. 4) Number displayer: this block was used to display values on the FPGA board. The paper concluded that using image normalization first before edge detection provides a more detailed gradient.

Many approaches have been utilized in edge detection using FPGA to enhance the speed or area features. They are extremely beneficial for real-time processing in terms of time, speed, area, and power. Currently, manufacturers are building bridges between the hardware and the software to reduce the float between them as a hybrid approach. All integrated environments or platforms, such as MATLAB, CUDA, and Quartus (FPGA boards), are trying to consolidate the hardware and software in one place to reduce implementation efforts and focus on implementing the logic itself. However, some vendors prevent high-performance computations from taking place if their platform is integrated with other tools instead of their bundled/certified tools.

EDGE DETECTION ALGORITHM

In this section, we provide an overview of the edge detection algorithm (Basu*,* 2002). It consists of four main steps. In the first step (pre-processing), a 2D Gaussian convolution operator is applied to smooth the image by removing noise. Gaussian filters are used widely in image processing and are very useful for edge and line detection. A 2D Gaussian filter is the only rotationally symmetric filter that is separable in Cartesian domain. It also satisfies the uncertainty relation defined in Equation 5 below, which provides the best trade-off between the incompatible goals of localization in the spatial (x) and frequency (ω) domains (Basu, 2002; Daugman, 1985).

$$\Delta x \Delta \omega \geq \frac{1}{2} \tag{5}$$

Gaussian-based detectors are detectors where an isotropic (i.e., circularly symmetric) 2D-Gaussian distribution variety is introduced in Equation 6. They are used to reduce the noise in the image before applying the edge detection process.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-(\frac{x^2+y^2}{2\sigma^2})} \tag{6}$$

where σ is the standard deviation of the distribution denoted by "Sigma" factor of the Gaussian distribution. The Gaussian function provided in Equation 6 is a normalized version of the function since the factor $1/2\pi\sigma^2$ is included. An important note here should be pointed to the reader that the size of the mask (resolution) and the value of σ will affect smoothing scale in the image. A higher value of σ leads to higher level of blurring the image.

The 2D Gaussian smoothing operator uses this distribution as a point spread function and is realized by using the convolution. In this work, we used the mask in the next step for the smoothing operation.

The second step (detection) is detecting the edge of the smoothed image. A 2D spatial gradient measurement is performed on the image using a Sobel operator in order to locate the approximate absolute gradient magnitude at each pixel in an input grayscale image. Eight neighboring pixels are required to calculate each pixel's gradient magnitude. In a Sobel edge detector, a pair of 3x3 kernels or convolution masks is used. One pair is used to estimate the column gradient in the x-direction, and the other pair is used to estimate the row gradient in the y-direction (Mintz, 1992; Rao *et al.,* 2004). In general, the convolution mask is much smaller compared to the actual image. Consequently, the mask covers the image to handle squares of pixels simultaneously. Gradients of the image can be derived with respect to the X-Y axis using Equation 7 below. Based on these gradients, Sobel`s gradient operators G_x and G_y can be derived from Figure 1, as shown in Equations 8 and 9, respectively.

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix} = \begin{pmatrix} G_x \\ G_y \end{pmatrix} \tag{7}$$

$$G_x = (p_7 + 2 \cdot p_8 + p_9) - (p_1 + 2 \cdot p_2 + p_3) \tag{8}$$

$$G_x = (p_3 + 2 \cdot p_6 + p_9) - (p_1 + 2 \cdot p_4 + p_7) \tag{9}$$

where p_i represents the pixel values of the image.

The window-based Sobel mask used for edge detection is shown in Figure 2.

G_x=

-1	0	+1
-2	0	+2
-1	0	+1

G_y=

-1	-2	-1
0	0	0
+1	+2	+1

Fig.2. G_x is the gradient in the x-direction, and G_y is the gradient in the y-direction.

The gradient magnitude is then determined using the formula shown in Equation 10:

$$M(x,y) \equiv mag(\nabla f) = \sqrt{G_x^2 + G_y^2} \tag{10}$$

Moreover, this equation is computationally costly because of square and square root operations for every pixel. It is more suitable computationally to approximate the square and square root operations by absolute values as shown in Equation 11.

$$M(x,y) \approx |G_x| + |G_y|$$
$$M(x,y) \approx \max(|G_x|, |G_y|) \tag{11}$$

where the $|G|$ value is assigned to the mask's central pixel.

The angle expressed in Equation 12 below gives the direction of the gradient vector.

$$\alpha(x,y) = \tan^{-1}\left(\frac{G_y}{G_x}\right) \tag{12}$$

Note that the magnitude is actually independent of the direction of the edge (isotropic operators).

Figure 3 shows the bmp input image of Lena (Images, 2019).

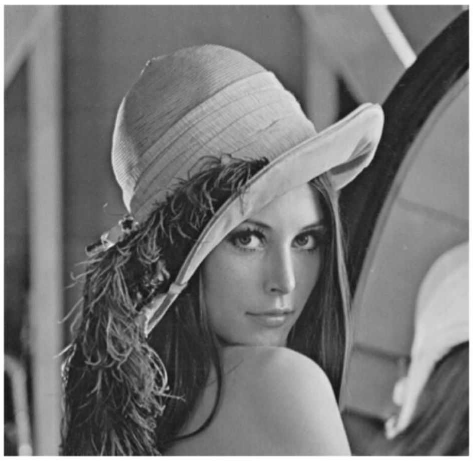

Fig. 3. The Lena bmp input image.

The direction and magnitude of the gradient are helpful in applying a non-maximum suppression in the next stage. The output from applying the Sobel operators convolved with a 3x3 Gaussian window on the Lena image is shown in Figure. 4.

Due to the magnitude computation for the image using the Sobel operator, the output image contains various ridges around the local maxima. As a result, a thinning process is used to reduce the thickness of the pixels in the image.

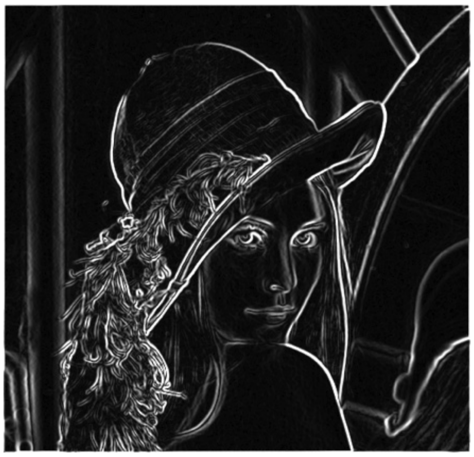

Fig. 4. The final result of applying the Sobel operators G_x and G_y after smoothing the Lena image (Fig. 3) with a 3x3 Gaussian convolution mask.

The third step is finding the maximum edge strength in a specific direction to start the process of thinning those ridges through a Non-Maximum Suppression (NMS) process. Figure 5 shows the edge direction and edge strength at a point where the edge normal (gradient vector) is perpendicular to the direction of the edge.

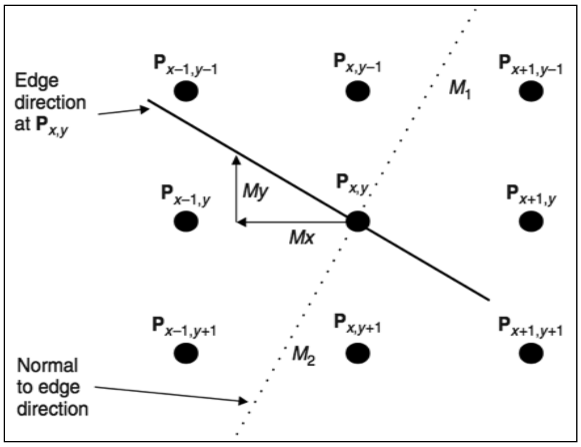

Fig. 5. The gradient vector and the edge direction of the edge point (Nixon *et al.,* 2008).

An NMS process is a thinning process and formulated as a Local Maximum Search that suppresses all the gradient values to zero except for the local maxima values by using the edge direction information and the magnitude along that direction. The algorithm used for Non-Maximum Suppression (Pham, 2010) can be briefly explained in Algorithm 1.

Algorithm 1: A Non-Maximum suppression pseudo-code for a 3x3 region

```
Input a 2-D image $A(x,y)$
1.  Compute $M(x,y)$ vector.
2.  Compute $a(x,y)$ vector.
3.  Define $D_k$ as a set of angles [-45°,0°,45°,90°].
4.  for all $i, j \in A(x,y)$
5.      Find $D_k$ closest to $a(i,j)$.
6.      if $M(i,j) < M(x,y)$ along positive $a(i,j)$ OR $M(i,j) < M(x,y)$ along negative $a(i,j)$ then
7.      $M(i,j) \leftarrow 0$.
8.      else $G(i,j) \leftarrow G(i,j)$.
9.      end if
10.     end for
output a 2-D NMS image $G(x,y)$
```

Algorithm 1 starts by calculating the magnitude vector and directional angle vector at each point of the input image using Equations 10 and 12, respectively. Based on the edge direction corresponding with the input image, the angle is approximated by one of predefined angles D_k. The magnitude at the location that matches the edge direction location in the angle vector is compared with the two neighbors along that direction. If the value of the magnitude *M(x,y)* is less than that of one of the two neighbors (along the positive and negative directions), then *M(x,y)* is suppressed; otherwise, it is left as it is (Gonzalez *et al.,* 2008). The output of the NMS routine is shown in Figure 6.

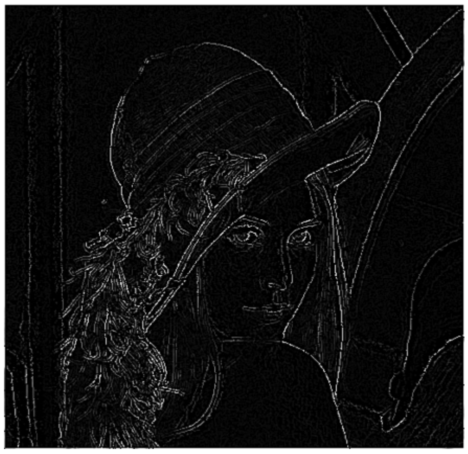

Fig. 6. Result of applying Non-Maximum Suppression on Figure 4. The pixels of the edge ridges are reduced to a one-pixel intensity and all values that are not local maxima are reduced to zero.

The fourth step (thresholding) is done to further sort out the edges. Many edge detection algorithms use hysteresis thresholding (double thresholding) after the NMS process to preserve the connected edges through the low and high thresholds. However, in our algorithm, the results of the third step in edge detection are multiplied by k, where k=5 in our case. Multiplying produces an amplified result with higher intensities. Since the results of NMS lead to a lower-intensity image, the process of thresholding is harder to maintain. Figure 7 shows the result after applying thresholding.

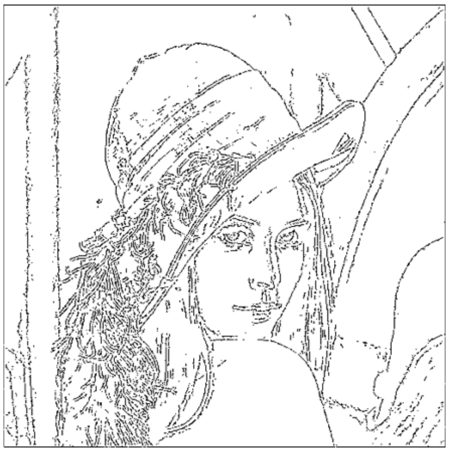

Fig. 7. The final result of applying thresholding at T = 242 in the edge detection algorithm.

After these four main steps, the final step is to scale the edge image to accommodate the original 8-bit grayscale image. Using software, the image is converted to a more precise version; thus a higher value is assigned during the processing stages of the image. The scaled image f_s can be obtained using the formulas provided in Equations 13 and 14.

$$f_m = f - \min(f) \tag{13}$$

$$f_s = K \cdot \frac{f_m}{\max(f_m)} \tag{14}$$

where f is the input image to be scaled, f_m is the minimum image intensity obtained from subtracting the minimum intensity value of the input image from the original image, and K is the upper limit of the values from $[0, K]$. For example, when working with a 4-bit image, K=15, and the scaled image f_s intensities will span the full 4-bit scale from 0-15.

IMPLEMENTATION OF THE ALGORITHM IN MITRION-C

This section provides details on the source of the parallelism in the edge detection algorithm, the implementation of parallelism in Mitrion-C, and bottlenecks.

Source of parallelism in the algorithm

Having gone through the algorithm in the previous section, it is easy to see that the convolution operation (i.e., the multiplication of each pixel in the window by its convolution mask and the addition of all products) can be performed in parallel and that fine-grained parallelism can be achieved, making it possible to harness the power of FPGA and speed the process up. To perform this operation, each pixel requires its eight neighboring pixels. Hence, logically, the process can be implemented in such a manner that it receives a signal pixel and generates a window including its eight neighboring pixels. Each pixel then has an independent window. During one memory read operation, if 16 pixels are read, then 16 windows can be generated independently, and convolution operations over 16 pixels can be applied in parallel. Running the convolutions in parallel speeds the process up.

The implementation of the edge detection algorithm consists of two parts of code. The first part is called the host code, and it is implemented using ANSI-C. The main task of this part is to check the file I/O, check for error, maintain the FPGA resources, and handle the handshaking to and from the FPGAs. The second part is called the Mitrion-C code, which is compiled and synthesized to generate a bit stream code, then loaded onto the FPGA in order to carry out edge detection. The complete details of the two codes are described below.

Mitrion-C code: The strategy of parallelism and bottlenecks

The image is stored to the external memory of the FPGA. The width of the external memory is 128 bits (SGI Inc., 2018). Therefore, 16 pixels can be stored in a single word of memory. Only one word can be read/written from/to memory at a time. As mentioned in Section 2, every pixel must be accompanied by its eight neighboring pixels in order to perform edge detection and smoothing by convolution. There are two ways to access the eight neighboring pixels. In the first method, access to the neighboring pixels is gained by reading data from three memory locations in sequence. In this method, each memory location needs to be accessed three times (first when the pixel is the current or central pixel of the window, second when the pixel is in the upper line of the current or central pixel in the window, and third when the pixel is in the lower line of the current or central pixel of the window). This method requires that the memory be accessed many times. As memory access always bottlenecks the performance, this method may require higher computation time.

In the second method, a pipelined window generator is implemented. The window generator produces a 2D array of 3x3 windows of pixels in the input image during every Mitrion Virtual Processor (MVP) clock cycle. These windows are not stored in memory but, instead, stored in logic as the image is read (SGI Inc., 2018). Windows can be stored in the internal memory of an FPGA, but memory access (bottlenecking the performance in terms of computation time) may reduce the performance. Therefore, storing the windows as logic will simplify the design and decrease the convolution function's memory usage because the function will work on a single isolated window, where the Gaussian and Sobel kernel functions will be called inside a nested parallel loop corresponding to the 2D representation of the input image. The remaining Mitrion-C code routines will read the data from the input memory bank and send the results to the output memory bank.

After generating the 3x3 window for 16 pixels (one word of memory), all the 16 windows are given to the <do_gausian> function. The Gaussian smoothing operation is then performed by this function.

The main body of the program, which is called <do_gausian>, includes many nested series of parallel foreach loops where every foreach loop of the vector collection type permits all iterations of a loop to run in parallel. The Mitrion-C code used to run the Gaussian function can be found in Appendix A.

EXPERIMENTAL RESULTS FOR THE FPGA APPROACH

In this section, our approach is tested and evaluated using common benchmarks for different images datasets (Lena, peppers, cameraman, and fishing boat) as shown in Figure 8 (Images, 2019). The SGI Altix contains two RC100 FPGA boards. Each of the RC100 boards contains two Xilinx Virtex 4 LX200 FPGAs. Each FPGA has two external SRAM banks of 16 MB to serve as the data source and sink. The word length of each SRAM is 128 bits. The FPGAs can be clocked up to 200 MHz when using an VHDL design flow, but the Mitrion compiler generates a virtual processor that is clocked at a fixed clock frequency of 100 MHz.

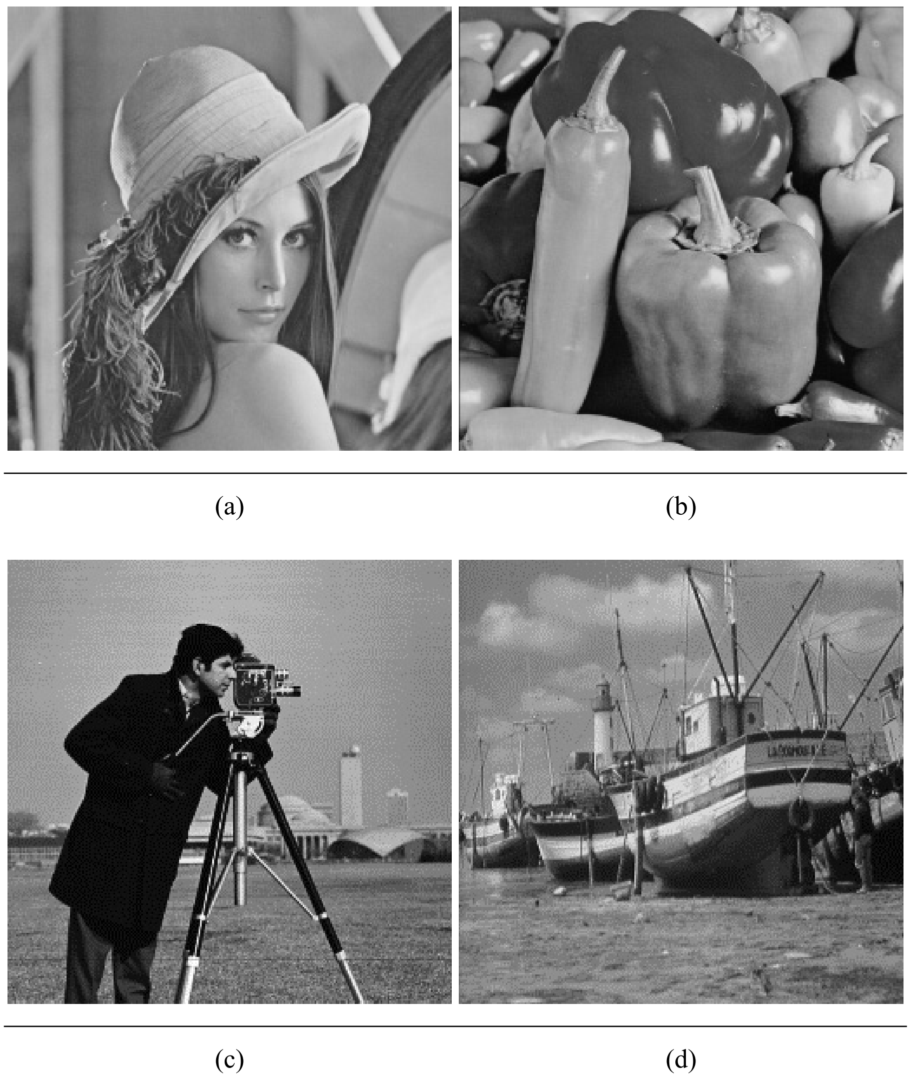

Fig. 8. Images benchmarks used for evaluation (a) Lena, (b) Peppers, (c) Cameraman, and (d) Fishing boat (Images, 2019).

The results for the FPGA-based edge detection algorithm and the software-based edge detection algorithm are shown in Table 1. An example of the benchmarks used for the Lena image is shown in Figures 9 and 10, respectively.

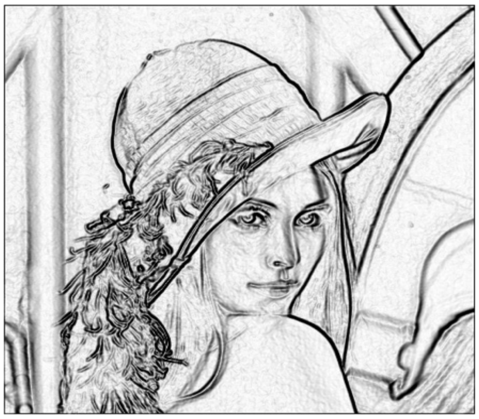

Fig. 9. Edge detected image using the FPGA-based algorithm.

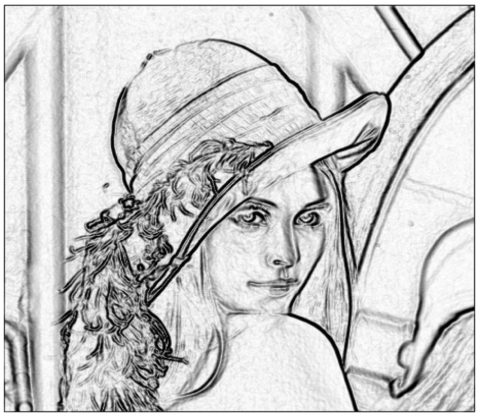

Fig. 10. Edge detected image using the software-based algorithm.

Table 1 compares the average execution time for the benchmarks used to differentiate the two implementations of the edge detection algorithm. As shown in the table, the FPGA-based algorithm can execute, on average, 50 times faster than the software-based edge detection algorithm. As explained earlier, the architecture implemented on the RC100 platform contains a Xilinx Virtex 4 LX200 FPGA. It has 16 Gaussian kernels and 16 Sobel kernels with which to perform parallel convolution operations on 16 input pixels. Each kernel can perform 9 multiplications and one addition of the products in parallel. The software-based algorithm is implemented using ANSI-C. The software is executed on an Intel Itanium 2-based architecture running at 1500 MHz. The software-based implementation performs convolutions sequentially using the loop-over-array method; hence, it is not very fast. The parallelism and pipelined features of the RC100 platform-based architecture provide a higher speed than the software-based implementation.

Table 1. Comparison between the software-based algorithm and hardware-based Algorithm in terms of execution time.

Image / Platform	Lena		Peppers		Cameraman		Fishing boat	
	SGI -RC 100 Based platform @100 MHz	Intel Itanium 2 @ 1500 MHz	SGI -RC 100 Based platform @100 MHz	Intel Itanium 2 @ 1500 MHz	SGI -RC 100 Based platform @100 MHz	Intel Itanium 2 @ 1500 MHz	SGI -RC 100 Based platform @100 MHz	Intel Itanium 2 @ 1500 MHz
Gradient Calculations Execution time (msec)	7	367.91	6.85	340.43	6.87	344.7	7.11	373.32
Gaussian Blur Execution time (msec)	5	240	4.87	230.5	4.89	232	5.07	242.5
NMS Execution time (msec)	1.9	100	1.79	97.2	1.81	97.5	1.95	101.5
Total Execution time (msec)	13.9	707.91	13.51	668.13	13.57	674.2	14.13	717.32
Language	Mitrion-C	ANSI-C	Mitrion-C	ANSI-C	Mitrion-C	ANSI-C	Mitrion-C	ANSI-C

Table 2 compares the hardware-based approach with the software-based approach of images in terms of different performance metrics such execution time, complexity, quality, power efficiency, and programmability. While software-based approaches are easy to implement and may provide a good image quality, hardware-based implementation is fast in terms of execution time due to parallelism and pipelining, suitable for real time systems as well as working under different times and frequencies constraints. The quality of the images in hardware approaches can be similar to the software approaches if there are no limitations on the resources (mainly the memory).

Table 2. Comparison between the software-based approach and hardware-based approach.

Metrics	Hardware-based Approach	Software-based Approach
Programmatically Easiness / and available APIs	• In case of FPGA, an HDL (Verilog – VHDL) is needed to implement every instruction, which make it hard task. • No way to build an API since each FPGA platform has different constraints and specifications. • Complicated and the implementer needs to consider limited resources presented by the board.	• A vast library and APIs are available to end-user such as OpenCV, image processing toolbox in MATLAB, NVIDIA Vision Works and many others. • Easy to be implemented, sometimes with only few lines in MATLAB.
Computational Performance, adaptability and Power Efficiency	• Very fast execution and computational time. • High performance per watt. • In terms of saving power, adaptable to application constraints. • Exploiting High level of parallelism rate of the application. • At lower frequencies, faster than multi-core CPUs (Afonso *et al.,* 2012, Asano *et al.,* 2009, Matic *et al.,* 2017).	• Slow execution time compared to hardware. • High performance rate. • Not adaptable to power constraints. • Restricted level parallelism due to compiling and translating stages.
Image Quality	• Same as software if no limitation on resources such memory otherwise the quality will be less.	• Same or better than hardware.
Working under timing constraints and different frequencies.	• For partially system and running on a bound of frequencies, FPGA outperform CPUs (Asano *et al.,* 2009, Kiran *et al.,* 2017).	• Can run almost in one frequency while changing the frequency suddenly may introduce crushes.
Complicated vision kernels and pipeline energy/frame.	• Outperform CPU/GPU for complicated kernels as complexity grows 1-22.3x (Qasaimeh *et al.,* 2019).	• Outperform FPGA for simple kernels on ratio 1-3.3x.

CONCLUSION AND FUTURE WORK

In this paper, we presented an image-processing algorithm based on convolution and edge detection that needs high computation time. In order to achieve higher computation performance in terms of execution time and speed, the algorithm is implemented on a reconfigurable hardware-based FPGA with parallel and pipelined architectures. The edge detection algorithm is implemented using ANSI-C to handle the host program and the Mitrion–C language and carried out on an SGI–RC100 platform. Mitrion–C provides an effective code for performing an edge detection suitable for parallel and pipelined architectures. The algorithm is benchmarked with an Intel Itanium 2-based architecture. In addition, the algorithm is implemented using software-based architecture. A comparison between the two approaches is carried out in terms of execution time. The results indicate that the FPGA-based algorithm is 50 times faster than the software-based edge detection algorithm. Some issues that can be handled in the future include using a Graphical Processing Unit (GPU) to enrich the comparison and dealing with the FPGA's clock since an FPGA can be clocked up to 200 MHz when using a VHDL design flow (the Mitrion compiler generates a virtual processor that is clocked at a fixed clock frequency of 100 MHz). Other future trends can be explored and compared with the work in this research by using hardware-based approaches to show the potential gains made by the parallel and pipelined approach.

REFERENCES

Abed, S., Ali, M.H. & Al-Shayeji, M. 2017. An adaptive edge detection operator for noisy images based on a total variation approach restoration, International Journal of Computer Systems Science & Engineering, **32**(1): 21–33.

Afonso, G., Atitallah, R.B. & Dekeyser, J. 2012. Software Implementation vs. Hardware Implementation: The Avionic Test System Case-Study. ASPLOS 2012, London, United Kingdom: 1-3.

Asano, S., Maruyama, T. & Yamaguchi, Y. 2009. Performance Comparison of FPGA, GPU AND CPU in Image Processing. 19th IEEE International Conference on Field Programmable Logic and Applications, FPL, Prague, Czech Republic: 126-131.

Basu, M. 2002. Gaussian-based edge-detection methods-A survey. IEEE Transactions on Systems, Man, and Cybernetics **32**(3): 252-260.

Bouganssa, I., Sbihi, M. & Zaim, M. 2017. Implementation in an FPGA circuit of Edge detection algorithm based on the Discrete Wavelet Transforms, Journal of Physics Conference Series, **870**(1):012016.

Canny, J.F. 1986. A computational approach to edge detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, **8**(6):769–798.

Daugman J.G. 1985. Uncertainty relation for resolution in space, spatial frequency, and orientation optimized by two-dimensional visual cortical filters. J.Opt. Soc. Am.A, **2**(7): 1160-1169.

Dollar, P. & Zitnick, C. 2014. Fast edge detection using structured forests. IEEE Transactions on Pattern Analysis and Machine Intelligence, **37**(8): 1558-1570, doi: 10.1109/TPAMI.2014.2377715.

Elaraby, A.E., El-Owny, H.B., Hassaballah, M., AbdelRardy, A.S. & Heshmat, M. 2013. A novel algorithm for edge detection of noisy medical images. International Journal of Signal Processing, Image Processing and Pattern Recognition, **6**(6): 365-374.

Etemad, S.A. & White, T. 2015. An ant-inspired algorithm for detection of image edge features. Applied soft computing, **11**(8): 4883-4893.

Guo, Z., Xu, W. & Chai, Z. 2010. Image edge detection based on FPGA. Ninth International Symposium in Distributed Computing and Applications to Business Engineering and Science (DCABES), Hong Kong, China: 169-171.

Huertas, A. & Medioni, G. 1986. Detection of intensity changes with subpixel accuracy using Laplacian-Gaussian masks. IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-8(5): 651-664.

Images. 2019. Accessible via http://www.dcs.qmul.ac.uk/~phao/IP/Images/ (last access: 16. 7. 2019).

Khan, T.M., Bailey, D.G., Khan, M.A.U. & Kong, Y. 2015. Real-time edge detection and range finding using FPGAs. Optik-International Journal for Light and Electron Optics, **126**(17): 1545-1550.

Kiran, M., War, K., Kuan, L., Meng, L. & Kin, L. 2008. Implementing image processing algorithms using 'Hardware in the loop' approach for Xilinx FPGA. International Conf. on Electronic Design, Penang, Malaysia: 1-6.

Lavanya, K.B., Reddy, K.V.R. & Yellampalli, S.S. 2014. Comparative analysis of different optimization technique for Sobel edge detection on FPGA. In International Conference on Advances in Electronics, Computers and Communications (ICAECC), Bangalore, India: 1-5.

Madhulika, D.Y., Madhurima, P.G., Kaur, G., Singh, J., Gandhi, M. & Singh, A. 2013. Implementing edge detection for medical diagnosis of a Bone in Matlab. In the 5th International Conference in Computational Intelligence and Communication Networks, Mathura, India: 270-274.

Matic, T., Aleski, I. & Hoceski, Z. 2017. CPU, GPU and FPGA Implementations of MALD: Ceramic Tile Surface Defects Detection Algorithm. Automatika, **55**(1): 9-21.

Mitrionics. 2001. www.mitrionics.se

Mintz, D., 1992. Robust consensus based edge detection. In Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '92), Las Vegas, NV, USA: 651-653.

Nausheen, N., Seal, A. Khanna, P. & Halder, S. 2018. A FPGA based implementation of Sobel edge detection, Microprocessors and Microsystems, **56**: 84-91.

Nixon, M. & Aguado, A. 2008. Feature extraction & image processing. Elsevier, 2nd edition.

Qasaimeh, M., Denolf, K., Vissers, J.L. & Zambreno, J., Jones. 2019. Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels. IEEE 15th ICESS 2019: Las Vegas, NV, USA: 1-8.

Rao, D.V. & Venkatesan M. 2004. An efficient reconfigurable architecture and implementation of edge detection algorithm using Handle-C. Proceedings of International Conference on Information Technology: Coding and Computing (ITCC 2004), Vol. **2**: 843–847.

Rashid, S.S., Dixit, S.R. & Deshmukh, A.Y. 2014. VHDL based canny edge detection algorithm. International Journal of Current Engineering and Technology, **4**(2): 749-752.

Rashmi, M.K. & Saxena, R. 2013. Algorithm and technique on various edge detection: A survey. Signal & Image Processing: An International Journal (SIPIJ) **4**(3): 65-75.

Samuel, A. & Fehrenbach, J. 2015. Edge detection using topological gradients: a scale-space approach. Journal of Mathematical Imaging and Vision, **52**(2); 249-266.

Shrivakshan, G.T. & Chandrasekar, Dr. C. 2012. A comparison of various edge detection techniques used in image processing. International Journal of Computer Science Issues (IJCSI), Issue 5, **9**(1): 269-276.

Singh, S., Shekhar, C. & Vohra, A. 2013. Area optimized FPGA implementation of color edge detection. In International Conference on Advanced Electronic Systems (ICAES), Pilani, India: 189-191.

SGI Inc. 2006. Reconfigurable application-specific computing user's guide. 007-4718-007. www.sgi.com.

SGI Inc. 2019. SGI RASC 100 Manual by SGI, (last access: 22. 01. 2019).

Uguz, S., Sahin, U. & Sahin, F. 2015. Edge detection with fuzzy cellular automata transition function optimized by PSO. Computers & Electrical Engineering, **43**(C): 180-192.

Gonzalez, R.C. & Woods, R.E. 2008. Digital image processing. Pearson Education, 3rd Edition.

Vanishree & K.V R. Reddy. 2013. Implementation of pipelined sobel edge detection algorithm on FPGA for high speed applications. International Conference on Emerging Trends in Communication, Control, Signal Processing & Computing Applications (C2SPCA), Bangalore, India: 1-5.

Xiaochen, L. & Fang, S. 2015. A convenient and robust edge detection method based on ant colony optimization. Optics Communications, **353**: 147-157.

Yasri, I., Hamid, N.H. & Yap, V.V. 2008. Performance analysis of FPGA based sobel edge detection operator. Proc. Int. Conf. Electron. Design, Penang, Malaysia: 1-4.

Yip, H.M., Ahmad, I. & Pong, T.C. 1999. An efficient parallel algorithm for computing the Gaussian convolution of multi-dimensional image data. The Journal of Supercomputing, **14**(3): 233-255.

Submitted: 18/02/2019
Revised: 05/10/2019
Accepted: 07/10/2019

Appendix A

Figure A1 shows the Mitrion-C code used to run the Gaussian function.

```
const MEM_SRAM_NWORDS = 0x80000;
const MEM_SRAM_NBITS = 128;
type MEM_SRAM=typedef  mem
bits:MEM_SRAM_NBITS[MEM_SRAM_NWORDS];

type PIXEL = typedef uint:8;
const PIXMAX     = 255;
const IMSIZEY    = 512;
const IMSIZEX    = 512;
const PAR_UNROLL    = 16;

(image_lv, mem_a_02) = foreach
(i in <0 .. NWORDS_PER_IMG-1>)

{PIXEL [PIX_PER_WORD] image_v;
(image_v, mem_a_01) = memread (mem_a_00, i);
} (image_v, mem_a_01);

image_llv = reshape
(image_lv, <IMSIZEY><IMG_BLOCKSIZE> [PAR_UNROLL]);

PIXEL <IMSIZEY><IMG_BLOCKSIZE> [PAR_UNROLL][3][3] stencil_llvvv =window_gn( image_llv );
new_smoothimage_llv = foreach (stencil_lvvv in stencil_llvvv )
{   new_smoothimage_lv = foreach ( stencil_vvv in stencil_lvvv )
        {
  new_smoothimage_v = foreach (stencil_vv in stencil_vvv)
            {
            PIXEL new_smoothimage = do_gausian (stencil_vv);
            } new_smoothimage;
        } new_smoothimage_v;
} new_smoothimage_lv;
```

Fig. A1. Mitrion-C code used to run the Gaussian function (Madhulika *et al*., 2013).

The stencil_llvvv collection type is PIXEL<IMSIZEY><IMG_BLOCKSIZE>[PAR_UNROLL][3][3]. Three nested loops are applied over stencil_llvvv. The outermost loop is over rows (i.e., list collection <IMSIZEY>) of the input image, and it passes over a single row to the inner row next to it in pipelined fashion. Each row has 32 blocks. The next inner loop is over the blocks (i.e., list collection <IMAG_BLOCKSIZE>) within a row, and it passes over a single block to the next inner loop in pipelined fashion. Every block has 16 3x3 windows. The innermost loop is over PAR_UNROLL (i.e., [PAR_UNROLL][3][3] – a vector collection of 16 3x3 windows) within each block. Note that PAR_UNROLL is defined as a constant, as shown in the header of the code. In addition, the PAR_UNROLL (i.e., 16 copies) of the do_gausian kernel will be located on the FPGA working simultaneously because the innermost loop is working on a vector collection. The do_gaussian function performs the convolution operation (explained in Section 1) between the mask and input window, as shown in Figure 2. Each function requires nine multiplication, one division, and one addition operation. The pipelining is provided by the outer two loops; thus 16 pixels are processed per clock tick.

After smoothing the image, the edge detection operation needs to be performed on the smooth image. It requires a 3x3 window for each smoothed pixel in order to perform a convolution operation involving the gradient-x and gradient-y on the window, as shown in Figure 3. The window is generated by calling the window_gen function again. The do_sobel function is used to implement the edge detection convolution. A 3x3 window of values surrounding a given pixel is fed into the function, and a new pixel is calculated via the procedure explained in Section 2. The procedure used to call the do_sobel function is similar that used to call the do_gaussian function to process PAR_UNROLL (16 copies) pixels per clock tick.

To measure the exaction time, the host code (host_temp.c) calculates the execution times of the Mitrion and software versions and compares their speeds. The host code executes the edge detection algorithm on a host processor from the Intel Itanium 2 family. Its main task is to read and write a bitmap image (*.bmp) and direct the image data to the Mitrion-C code which is running on an FPGA.

الكشف عن خوارزمية الحافة عن طريق إعادة تشكيل الأجهزة باستخدام FPGA

سائد عبد
قسم هندسة الحاسوب، كلية الهندسة والبترول، جامعة الكويت، الكويت

الخلاصة

تُعرف معالجة الصور الرقمية بأنها معالجة الكمبيوتر للصور، والتي تتضمن خوارزميات مثل تحسين الصورة أو استخلاص بعض الخواص والمزايا. تتضمن بعض هذه الخوارزميات عمليات مثل الالتواء والكشف عن الحواف، الأمر الذي يتطلب إجراء حسابات عالية. بشكل عام، يقوم البرنامج الذي يتم تشغيله على المعالج بإجراء هذه الحسابات. لتحقيق أداء عالي من حيث وقت التنفيذ، يتم تنفيذ هذه الخوارزميات على أجهزة قابلة لإعادة التكوين مثل FPGA. يمكن للمرء تنفيذ هندسة متوازية ومعمارية على FPGA لزيادة السرعة. في هذا العمل، نقدم وصفاً مفصلاً لخوارزمية تنفيذ الكشف عن الحافة على منصة SGI – RC100. يتم تنفيذ الخوارزمية باستخدام ANSI-C لمعالجة البرنامج المضيف ولغة Mitrion – C. يوفر Mitrion – C طريقة فعالة لكتابة التعليمات البرمجية للعمارة المتوازية والموضوعة في خط أنابيب لإجراء كشف الحافة. بعد ذلك، يتم اختبار الخوارزمية على بنية Intel Intanium 2 ومقارنة وقت التنفيذ مع الخوارزمية المستندة إلى منصة RC 100 للتحقق من زيادة السرعة من خلال الخوارزمية المستندة إلى FPGA. وأظهرت النتائج التجريبية أن سرعة خوارزمية FPGA القائمة على الأجهزة القابلة لإعادة التشكيل تفوقت على المنهج القائم على البرمجيات بأكثر من 50 مرة.