A HIGH THROUGHPUT FPGA BASED ARCHITECTURE FOR REAL TIME EDGE AND CORNER DETECTION

[1]A.Ramya, [2]L.Kalaiselvi
[1]Resaerch Scholar, Department of ECE, Surya Engineering College, Erode, India
[2]Assistant Professor, Department of ECE, Surya Engineering College, Erode, India.
[1]rams1810@gmail.com, [2]kalaiselvi18@gmail.com.

Abstract: This paper proposes a new flexible parameterizable architecture for image and video processing with reduced latency and memory requirements, supporting a variable input resolution. The proposed architecture is optimized for feature detection, more specifically, the canny edge detector and the Harris corner detector. The architecture contains neighborhood extractors and threshold operators that can be parameterized at runtime. Also, algorithm simplifications are employed to reduce mathematical complexity, memory requirements, and latency without losing reliability. Furthermore, we present the proposed architecture implementation on an FPGA-based platform and its analogous optimized implementation on a GPU-based architecture for comparison. A performance analysis of the FPGA and the GPU implementations, and an extra CPU reference implementation, shows the competitive throughput of the proposed architecture even at a much lower clock frequency than those of the GPU and the CPU. Also, the results show a clear advantage of the proposed architecture in terms of power consumption and maintain a reliable performance with noisy images, low latency and memory requirements.

Index Terms: Reconfigurable hardware, graphics processors, real-time systems, computer vision, edge and feature detection.

1. INTRODUCTION

Feature detection algorithms, such as edge and corner detection, are essential components of many computer vision applications [1], e.g., image segmentation, object recognition, and feature tracking. The Canny edge detector and the Harris corner detector are the most widely- used feature detection algorithms due to their reliable performance with noisy images. The computation- ally intensive nature of these algorithms imposes high clock frequencies and significant power consumption on general microprocessor architectures, especially when it is necessary to meet real-time constraints. Due to their inherent parallelism, algorithms for image and video processing are better performed on parallel architectures, such as Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs). Starting with the graphic processing units, these units present a massively parallel architecture that consists of many processors. They have dramatically evolved during the last decade and hence achieve more computational power than Central Processing Units (CPUs) [2]. This evolution makes them highly attractive hardware platforms for general purpose computation. For a better exploitation of this high power, the GPU's memory bandwidth has also evolved significantly. Moreover, the advent of GPGPU (General Purpose GPU) languages makes it possible to exploit GPU for more types of application and not only for image rendering and video games. In this context, NVIDIA launched the API CUDA (Compute Unified Device Architecture) [3], a new programming approach which exploits the unified design of the most current graphics processing units from NVIDIA. Under CUDA, GPUs consist of many processor cores which can address GPU memories directly. This fact permits a more flexible programming model than previous ones [4]. As a result CUDA has rapidly gained acceptance in domains where GPUs are used to execute different intensive parallel applications. FPGAs, on the other hand, are fine-grained recon- figural architectures that can virtually perform any processing operation at a hardware level, satisfying real-time requirements of image and video processing and off-loading these computing intensive tasks from general microprocessors. However, the development time needed to create a working hardware implementation of an algorithm is longer and less flexible than a software analogous implementation. To comeup with competitive implementations of both the canny edge detector and the Harris corner detector algorithms that satisfy real-time requirements, we propose a new multi- resolution FPGA-based architecture that supports runtime parameterizations of its internal processing blocks We also propose an optimized GPU implementation of those algorithms in order to provide a comparison between these two

approaches, analyzing their advantages and drawbacks. With proper design constraints and application-to-architecture mapping, we show how FPGAs can be a suitable alternative to GPU- based image and video processing units, both in terms of flexibility and real-time performance. This is especially valid when portability, low-latency and power consumption are needed. This paper is an updated extension of the work [5] in which only the FPGA implementation was addressed. Also, the present work Provides additional results obtained with up-to-date FPGA- and GPU- based platforms. The rest of this paper is organized as follows: Section 2 presents the state-of-the-art and related works on feature detection algorithms, specifically edge and corner detection, and their implementations on reconfigurable platforms. Section 3 gives a brief overview of edge and corner detection and describes the algorithms explored in this work. The GPU and FPGA architectures and their implementations are presented in Sections 4 and 5, respectively. Section 6 presents the analysis of results obtained with these architectures. Finally, Section 7 gives the conclusions of this work as well as the future work perspectives

2. RELATED WORK STATE OF THE ART

This section is divided into two parts. The first part is dedicated to the feature detection algorithms, more specifically edge and corner detection, some keywords, and the evaluation of several state of the art techniques, at the origin of the algorithms studied in this work. The second part presents some important GPU implementations of feature detection algorithms followed by a description of FPGA-based architectures with the same objectives, analyzing their positive and negative aspects and comparing their achievements to our proposed architecture.

2.1 Feature Detection Algorithms

Several techniques have been proposed for both edge and corner detection. Regarding edge detection algo- rithms, the works in [6] and [7] present a comparison of several classical edge detection techniques. Results show that the canny edge detector, proposed in [8], has a better performance than the other detectors in different scenarios. Although the Canny approach is a well-known technique with a good response to noisy images and largely employed in recent applications as in [9] and [10], new techniques have been explored showing better performance. One example is the global probability of boundary (gPb) proposed in [11]. Regarding corner detection algorithms, the work in [12] presents a comparison of classical techniques, and according to the results presented, the Harris corner detector, proposed by [13], has a better performance than other detectors. As in the case of Canny, the Harris technique is a well- known robust solution for tracking interesting points in a video stream. A recent work in [14] presents the state-of-the- art of interest point detectors describing new techniques with better performance than Harris, e.g., the Fast Hessian [15]. In this work, we chose to explore the widely-used Canny and Harris detection algorithms due to their reliable performance with noisy images.

2.2 Processing Architectures and Implementations

Most of feature detection algorithms include sections that consist of similar computation with pixels. This fact means that these algorithms are appropriate for acceleration on GPU by exploiting its processing units in parallel.

In this context, [16] implemented several classic image processing algorithms on GPU with CUDA [3]. The OpenVIDIA project [17] has implemented different com-puter vision algorithms running on graphic hardware such as single or multiple graphic processing units. In the medical imaging domain, there are some GPU works for new volumetric rendering algorithms [18] and Mag- netic Resonance (MR) image reconstruction [19]. There are also different works dealing with the exploitation of hybrid platforms of multicore processors and GPUs. OpenCL [20] proposed a framework for writing pro-grams which execute across hybrid platforms consisting of both CPUs and GPUs. The work of [21] presented a flexible programming model for multicore processors. In the same context, StarPU [22] provided a unified runtime system for heterogeneous multicore architectures permit-ting the development of effective scheduling strategies Regarding FPGAs, many recent works have presented feature detector implementations in order to meet real- time requirements as in [23], [24], [25], and [26]. All these implementations have a fixed or slightly configurable architecture, as in [26] which presents an improved canny edge detector with a self-adaptable threshold mechanism. Most of the previously cited works are basically a cascaded set of neighborhood operators that must be redesigned

and resynthesized for every different algorithm or frame resolution. This characteristic reduces the system flexibility since they do not permit parame terization at runtime. Our proposed architecture reduces these limitations by using configurable neighborhood extractors, as explained in Section 5.2. In terms of processing performance, these Implementations are, in the best of cases equivalent to our architecture performance on a single pipeline configuration. In [27] a distributed implementation of the canny algorithm is presented with a performance around 3.8 times faster than our implementation. To achieve this, the input image is split into 16 blocks and each block is handled by a particular processing core. This solution requires more resources and a simultaneous reading from 16 distinct regions of an image, which indicates that the input image must be pre-buffered. In order to minimize latency, our proposed architecture was designed to process a flow of pixels without a pre-buffering stage. Minimal latency is important in applications that require a quick response from the system when any change in the input occurs, e.g., vehicle obstacle detection and military targeting systems. However, if the application permits a pre-buffering stage, the main pipeline can be replicated to simultaneously work in different pixel flows, increasing the computing performance by a factor equal to the number of pipelines. In this case, we could achieve a computing performance similar to [27] implementation with less than 4 pipelines. In [28], two implementations of the same medical video processing application are presented, one on GPU and another on FPGA. It also presents an interesting discussion about GPU vs. FPGA implementation highlight -ing the fact that FPGA solutions can be more compact and consume less power, if compared to GPUs, at the cost of a high development time.

Another comparison of FPGA and GPU, presented in [29], gives similar conclusions to [28], adding that FPGAs are not recommended for applications using datasets with a large fixed-point representation is not suitable. On the other hand, according to [29], GPUs are not suitable for applications that require very short latency responses. These conclusions agree with our proposal of low-latency feature detection architecture.

3. EDGE AND CORNER DETECTION

Many computers vision applications use edge and corner detectors as primary operators before high level processing, such as object recognition and tracking. For instance, the information associated to the edges of an object in an image which is, in many cases, sufficient to identify the object. In this section, we will give an overview on these two components of vision systems, showing their foundation and describing the most common techniques of implementation.

3.1 Edge Detection

Edges are defined as an image position where a significant change in intensity values occurs [30]. Basically, if the brightness of a pixel has a significant difference from pixels in its neighborhood, it may contain an edge. In order to detect those changes, the local gradient approximation of image function I (u, v) is usually applied, which is the basis of many traditional operators of edge-detection. The gradient vector ∇I is composed of the first order partial derivatives (1) of function I alongside its coordinate axes (u, v).

$$\nabla I(u,v) = \frac{\delta I(u,v)}{\delta u}$$

$$\frac{\delta I(u,v)}{\delta v}$$

The magnitude of ∇I is obtained from (2)

$$\nabla I(u,v) = \frac{\delta I(u,v)}{\delta v}^2 + \frac{\delta I(u,v)}{\delta v}^2$$

Some operators are commonly used for approximating this gradient. Two examples of these are the Sobel and Prewitt operators. They use linear filters to obtain gradi- ents in each direction x and y. Equations (3) and (4) show the filter matrices H_x and H_y of these two operators

$$H_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad H_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \qquad (3)$$

$$H_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & -1 \end{bmatrix} \quad H_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \qquad (4)$$

Applying these filters to an input image for either a Sobel or a Prewitt operator results in two gradients Gx and Gy, as shown in equations (5) and (6). Notice that the function presents the convolution operator

$$Gx(u, v) = Hx \otimes I(u, v) \quad (5)$$
$$Gy(u, v) = Hy \otimes I(u, v) \quad (6)$$

With these gradients, it is possible to obtain a gradient magnitude that represents the local edge strength |G| (7)
and the local edge orientation angle Φ (8).
$$((G(u,v)) = (G_x(u, v)), (Gy(u, v)) \quad (7)$$
$$\Phi(u, v) = \tan^{-1} G_y(u, v) \quad (8)$$

$$G_x(u, v)$$

More elaborate algorithms can be used in order to enhance edge location. One of the most popular of these algorithms is the Canny edge detector due to its mini- mum number of false edge points, good localization of edges, and single mark on each edge [31].

Basically, the canny algorithm is composed of three steps: smoothing, edge enhancement, and localization. For the smoothing step, the canny algorithm uses a Gaussian low pass filter to suppress the noise of the input image. Next, in the edge enhancement stage, it is necessary to calculate a gradient vector at each pixel of the smoothed image. For example it is possible to calculate the gradient vector, magnitude and angle, by either processing the Sobel or Prewitt operator, or simply computing the local first norder derivatives along its coordinate axes by the approximations (9) and (10).

$$Ix(u,v) = \delta I(u,v) \sim {}^{f}(u+1) - {}^{f}(u-1) \quad (9)$$
$$\delta x \qquad 2$$
$$Iy(u,v) = \delta I(u,v) \sim {}^{f}(v+1) - {}^{f}(v-1) \quad (10)$$
$$\delta y \qquad 2$$

It is also possible to combine the smoothing and edge enhancement step in one single Step by convolving the derivative of a Gaussian kernel the localization step is divided into two stages: non-maximum suppression and hysteresis thresholding. The objective of the non-maximum suppression is to eliminate non- ridge pixels giving a one pixel wide aspect at the edges. A ridge pixel is defined as a pixel with a gradient magnitude greater than that of the adjacent pixels in the gradient direction. In the hysteresis thresholding stage, two thresholds are used, Tlow and T_{high} . All pixels with a magnitude higher than T_{high} are considered true edges. Pixels with magnitude between T_{low} and Thigh are considered as edge candidates. Pixels that do not satisfy these two criteriaare suppressed. Edge candidates become true edges if they are connected to true edges directly or through other candidates. The values of Tlow and Thigh depend on image characteristics, e.g., brightness and contrast, and have the same range of pixel intensity, e.g., 0 to 255 in 8-bit gray scale. Methodologies to determine the threshold values are out of this paper's scope and will be treated as values specified by the user beforehand. For more information regarding these methodologies, we refer the readers to [1], [30], [31], and [32].

3.2 Corner Detection

A corner is defined as an area that exhibits a strong gradient value in multiple directions at the same time [31]. The Harris operator uses this premise to find corners in an image. The first step is to obtain the first partial derivative of the image function I (u, v) in directions, horizontal and vertical, based on the approximations (9) and (10). With the values of Ix and I_y , it is possible to calculate the elements of the matrix M, described in (11), using (12), (13), and (14).

$$M = A \; C$$
$$C \; B \quad (11)$$
$$A = I^2 x \otimes \omega \quad (12)$$
$$B = I^2 y \otimes \omega \quad (13)$$
$$C = (Ix \; Iy) \otimes \omega \quad (14)$$

where ω is a smoothing circular operator, e.g., a Gaussian filter. The final step is to obtain the Harris operator response R as in (15).

$$R = Det[M] - k \cdot Tr^2[M] \quad (15)$$

where R is positive in corner regions, negative in edge regions, and is very small in flat regions, and k is a coefficient that, in practice, is a fixed value in the range of 0.04 to 0.06. This step can also be obtained by analyzing the eigenvalues of the matrix M.

4. GPU ARCHITECTURE AND IMPLEMENTATION

Feature detection algorithms are excellent candidates for acceleration on GPU since they consist mainly of common computation over many pixels. This fact is due to the exploitation of the high number of GPU computing units in parallel. Thus, we can say that graphic cards present an efficient tool for boosting the performance of image processing techniques. This section is presented in two parts: The first describes our proposed development scheme for image processing on GPU, based upon CUDA

for parallel constructs and OpenGL for visualization. The second part is devoted to the presentation of the GPU implementation of edge and corner detection methods based on the canny and Harris techniques respectively.

4.1 Proposed scheme for image processing on GPU

We propose in this section a development scheme for image processing on GPU, making it possible to load, treat and display images on graphic cards. This scheme is based upon CUDA for parallel constructs and Open GL for visualization, which reduces data transfer between the device and host memories. It is based on three steps as illustrated in Fig. 1:

1) Input data loading: This step loads the input data (images) from host (CPU) to device (GPU) memory which makes it possible to apply GPU treatments on the copied image

2) CUDA parallel processing: This step has two main stages: Threads allocation: Once the input data are loaded on the GPU memory, the number of threads in the GPU grid has to be selected, so that each thread can perform its processing on one or a group of pixels. This enables threads to treat pixels in parallel Note that the number of threads depends on the number of pixels in the input image. CUDA processing: The CUDA functions (kernels) are executed using the number of threads selected in the previous step.

3) Output results: After processing, results can be presented using two different scenarios:

• Open GL visualization: The graphic library OpenGL is used for displaying the output images quickly, since it exploits buffers that already exist on GPU. Indeed, this avoids more data transfer between host and device memories. This scenario is useful when parallel processing is applied on a single image only since we cannot display many images using one video output (one GPU disposes of one video output). Transfer of results: OpenGL visualization becomes impossible in the case of multiple images processing using one video output only. In this case, the transfer of output images from the GPU to the CPU memory is required. This transfer time presents an additional cost for the application for an optimized utilization of graphic processors; we propose to exploit the GPU's texture and shared memories. Hence, we loaded the input image on the GPU's texture memory for a fast access to pixels. We have also loaded each neighboring pixel onto the GPU's shared memory for a faster processing of the image's pixels using their neighbors' values

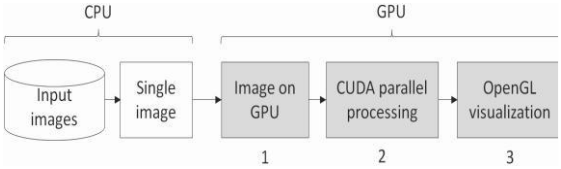


Figure 1: Image processing on GPU based upon CUDA and OpenGL.

Based on the scheme described above, we propose the GPU implementation of edges and corners detection methods, enabling both efficient results in terms of the quality of detected edges and corners, and improved performance thanks to the exploitation of GPU's computing units in parallel.

4.2 Edge detection on GPUs

This section describes the GPU implementation of the edge detection step based on a recursive algorithm using the Canny-Deriche design [33]. Noise truncate immunity and the reduced number of required operations make this method very efficient. This technique is based on four principle steps

1) Recursive gradient computation (G_x,G_y).
2) Gradient magnitude and direction computation.
3) Non-maximum suppression.
4) Hysteresis thresholding.

Notice that the recursive gradient computation step applies a Gaussian smoothing before filtering the image recursively using two Sobel filters in order to compute the gradients Gx and G_y. Within the steps of gradient magnitude and direction computation, the non- maximum suppression and hysteresis occur thus representing the same steps used for the canny filter described in the previous section.

The proposed GPU implementation of this recursive method is based on the parallelization of all computation steps on GPU. Below, we describe the implementation and the steps as presented in Fig. 2.

Recursive Gaussian smoothing: The GPU implementation of the recursive Gaussian smoothing Step is developed using the CUDA Software Development Kit (SDK) individual sample package [34]. This parallel implementation is applied on the Deriche recursive method [33]. The advantage of this method is that the execution time is independent of the filter width. The use of this technique for

smoothing permits a better noise truncate immunity.

Sobel filtering: The recursive GPU implementation of this step is also provided from the CUDA SDK individual sample package [34]. This parallel implementation exploits both shared and texture memories which leads to a boosting of the performance.

Gradient magnitude and direction computing: Once the horizontal and vertical gradients (G_x and G_y) have been computed, we calculate the gradient magnitude (intensity) using equation (7) and the gradient direction

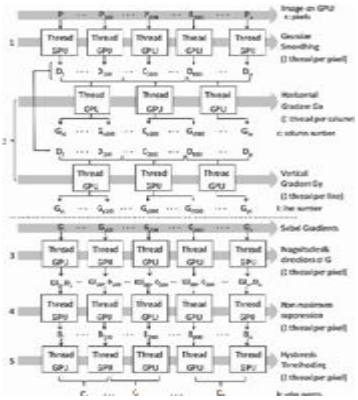


Figure 2: GPU implementation of Canny-Deriche

Edge detector using equation (8). The CUDA implementation of this step is applied in parallel on image pixels, using a GPU grid computing containing a number of threads equal to the number of pixels in the image. For example, 480,000 threads would be required for an 800×600 image resolution. Thus, each thread calculates the gradient magnitude and direction of one pixel of the image.

Non-maximum suppression: After computing the gradient magnitude and direction, we apply a CUDA function (kernel) which enumerates the local maximum, which are pixels with high gradient intensity. We pro- pose to load the values of neighbor pixels (left, right, top, and bottom) in shared memory since these values are required for the search for the local maximum. The number of selected threads for parallelizing this step was also equal to the number of pixels in the image.

Hysteresis thresholding: Hysteresis presents the final step of edge production. The GPU implementation of this step can be presented in two phases. The first one consists of selecting threads with a number equal tothe number of image pixels. Each thread checks if its corresponding pixel has a gradient value greater than T_{high}. This pixel will be marked as an edge point. Then, for the second phase, each block of threads will treat one marked edge point and its eight neighbors (connected pixels). These pixels are loaded on the shared memory in order to have a fast access to their values. Each connected pixel will be marked as an edge point if its gradient intensity is greater than the low threshold T_{low} .

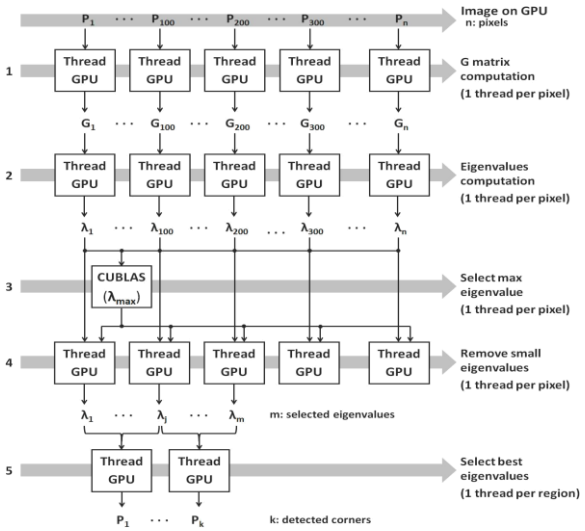


Figure 3: GPU implementation of the Harris corner detector.

4.3 Corner detection on GPUs

We developed the GPU implementation of Bouguet's corners extraction method [35], based on the Harris detector [13]. This\ method is proven to be efficient thanks to its invariance to rotation, scale, brightness, and noise. Our GPU implementation of this method is based on parallelizing its five steps on GPU as shown in Fig. 3.

Spatial derivatives computation: The first step consist on computing the matrix G of spatial derivatives for each pixel using equation (16).). This matrix of 4 elements (2×2) is calculated with the spatial derivatives I_x and I_y which are computed using the equations(9) and (10) respectively.

$$G = \begin{matrix} I^2x & Ix\,Iy \\ Ix\,Iy & I^2y \end{matrix} \qquad (16)$$

The GPU implementation applies a parallel treatment of pixels using a GPU grid which contains a number of threads equals to the number of pixels.

The values of neighbors' pixels (left, right, top, and bottom) of each image point are loaded in the GPU shared memory since these values (neighbors) are required for computing the spatial derivatives. Each thread computes the spatial derivatives of one pixel. Then, each thread can calculate the elements of the matrix G.

Eigenvalues computation: Based on the matrix G, we calculate the two eigenvalues of each pixel. Then, we keep the highest eigenvalue for each pixel. The GPU implementation of this step is performed by computing these eigenvalues in parallel over image pixels. In this case, we have also used a GPU grid which contains a number of threads equals to the number of pixels.

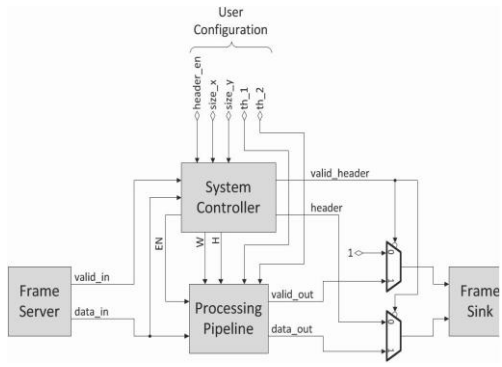


Figure 4: Functional blocks of the proposed architecture

Maximum eigenvalue selection: Once the eigenvalues are calculated, we extract the maximum value. This value is computed on GPU using the library CUBLAS [36].

Removing of small eigenvalues: The research of eigenvalues is performed such that each GPU thread compares the eigenvalue of its corresponding pixel with the maximum eigenvalue. If this value is lower than 5% of the maximum value, the pixel will be excluded.

Selection of best values: The last step enables, for each image area, the extraction of the pixel with the highest eigenvalue. For GPU implementation, we create a GPU thread for each group of 10×10 pixels. Each thread allows the detection of the maximum eigenvalue in a region using the CUBLAS library. The pixels with these extracted values represent the detected corners. For more details about this implementation, we refer the readers to [37] and [38].

5. FPGA ARCHITECTURE AND IMPLEMENTATION

The proposed architecture processes a streamed image or sequence of images with variable resolutions. The frame resolution can be detected directly from the header of images files or it can be manually configured by the user. In both cases, this information adapts the whole architecture on-the-fly. Fig. 4 shows the functional blocks of the main architecture for both the Canny and Harris detectors. The proposed Architecture can work as an accelerator for image processing where the Frame Source and Frame Sink are the interface between the host computing system and the architecture, e.g., PCIe or Gigabit Ethernet. In a different operating mode, it can work as a stand-alone image processor placed directly on a pixel stream, e.g., embedded in a camera system.

This section is divided into four parts where the first two parts are dedicated to the main components of the proposed architecture, the System Controller and the Neighborhood Extractor (NE). The last two parts describe the computational blocks used to implement the Canny and Harris detectors.

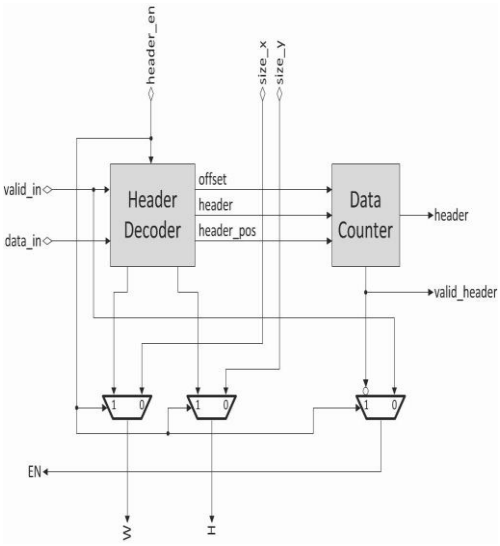


Figure 5: Architecture of the System Controller block

5.1 System Controller

The System Controller, shown in Fig. 5, is composed of two main blocks, the Header Register and the Data Counter. These blocks operate in two different modes ac- cording to the user input signal header en. This signal indicates if the data input is a single

image (header en = 1) or an image sequence (header en = 0). If the data input is a single image the Header Register can extract image characteristics directly from the file's header.

The Width (W) and Height (H) characteristics are sent to the Processing Pipeline in order to configure the line registers, which are image width dependent. The Data Counter examines the current position of the stream in order to generate two signals are transferred to the output (Frame Sink) without traversing the Processing Pipeline. In the case of an image sequence, the Header Register and the Data Counter blocks are disabled and the user configuration is transferred di- rectly to the Processing Pipeline. The System Controller only supports non-compressed image and video formats, more specifically, bitmap (BMP) images on single image processing mode and regular progressive raster scanned video stream on image sequence mode.

When the proposed architecture is operating on single image mode, it must process one image completely before starting to process a new one. This approach allows it to process a sequence of images with different sizes since every new image can readjust the architecture parameters without interfering with the previous image processed. However, if all the input images have a known and fixed size, the image sequence mode can be used to reduce idle resources and latency. In this mode, the architecture processes all input images in a sequence, keeping the processing pipeline full all the time.

5.2 Neighborhood Extractor

The NE block provides a sliding window with a fixed dimension (w × h) to the subsequent processing block

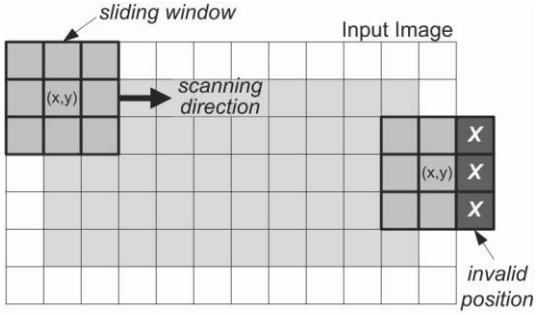


Figure 6: A 3×3 sliding window where the valisscaning positions are the gray pixels in the input image.

It was designed to support images with variable resolution and automatically handle the image borders, keeping a reduced memory requirement and minimizing the latency. In order to simplify the description of the NE operation, the smallest version in the proposed system, a 3×3 NE window, will be used as a reference. As an illustration, Fig. 6 presents 3×3 NE window characteristics where the window scans the whole image following the image coordinates that go from (0, 0) at the origin to(W-1,H-1)

A characteristic problem that concerns neighborhood computations is the border problem illustrated in Fig. 6. It occurs because a neighborhood can only be processed if it fits wholly within the image, resulting in a smaller image. To solve this problem, we have added a padding mechanism that extends the image boundaries by repli- cating, or clamping, the pixels at the image limits.

The basic structure ofthe NE is a set of cascaded line buffers connected to register arrays from where it is possible to read the current and two or more previously stored pixels. Fig. 7 shows the proposed 3×3 NE block architecture functional blocks and Fig. 8 presents the register array architecture. A secondary structure is responsible for processing the image borders. This structure is based on the Coordinate Counter (Fig. 7) Which Provides? The input coordinates to the mechanism selecting the output according to the window position. A variation of the NE is used in the Canny detector's hysteresis stage, where the input of last line buffer (line buffer is connected to the output of the connector block (Fig. 10), allowing it to reuse its own output as part of its Neighborhood input. This recursive behavior improves the hysteresis' performance in a one-pass image scan.

The Line Buffer is shown in Fig. 9. Based on the image width and window position, the Line Buffer Controller generates the write and read addresses, WR ADD and RD ADD respectively, for a dual-port on-chip RAM block with a size of 4096 Bytes. This configuration has the advantage of supporting different image resolutions without requiring are synthesis process.

Indeed, the System Controller (Fig. 5) can reprogram the Line Buffer Controller on the fly when a new image with a different resolution arrives or it can be done manually by the user. The maximum resolution supported by the NE is$W_m \times H_m$ pixels, where W is limited by the on-chip RAMblock size, 4096 pixels in this case, and H depends on the

size of a System Controller internal register, which is fixed to 12 bits, addressing up to 4096 pixels (2^{12}).

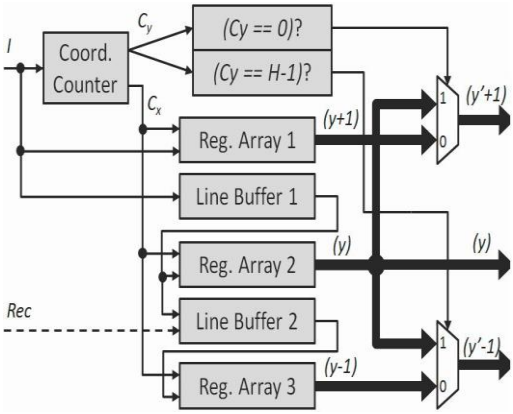


Figure 7: The 3×3 NE architecture

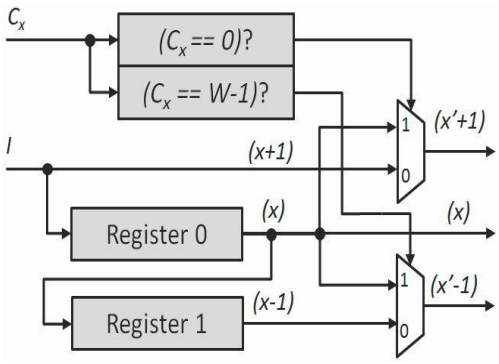


Figure 8: The register array architecture

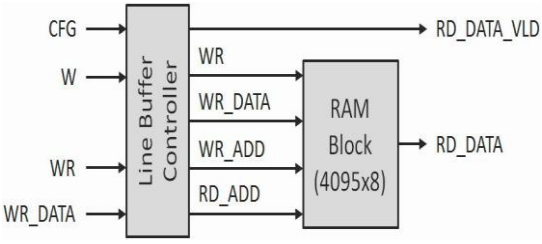


Figure 9: The line buffer architecture

In terms of latency, the NE block minimizes the required number of buffered pixels. Considering that the window size is w × h, the latency in pixels of the NE block can be calculated by (17).

$$NElatency = \frac{w-1}{} \quad \frac{h-1}{} + W$$

For different window sizes, the NE only differs in the number of line buffers, the size of the register array and the complexity of the image border handler that must include the extra elements in the window.

5.3 Canny Detector

The Canny detector processing pipeline follows the original Canny algorithm with some modifications to simplify mathematical operations, optimizing performance and utilization of resources.
Fig. 10 shows the functional diagram of the canny edge detector processing pipeline. Below, we describe all the steps along the pipeline

Color to grayscale: The first step in the FPGA implementation of the Canny edge detector is a conversion from 24-bit RGB color standard into gray scale where each pixel n is n represented by 8-bit samples carrying the pixel's intensity. This step is performed by the C2BW block which computes the average intensity of the three colors (red, green, and blue) in each input pixel.

Gaussian smoothing: The smoothing stage is based on a Gaussian low-pass filter. The Gaussian filter requires a 5×5 pixels window provided by an NE block and is computed based on a fully parallelized linear filter operator defined in (18). This operator firstly multiplies all the elements of the input window by the corresponding kernel coefficients (Fig. 11). Then, these intermediate results are summed up in an adder tree. Finally, the total is divided by normalization factor.

$$g(u, v) = f(u + i, v + j) \cdot h(i, j) \qquad (18)$$

Where g(u, v) is the resulting image, f (u, v) is the input image, and h(i, j) is the kernel.

Sobel filtering: The edge enhancement is made by processing the Sobel operators defined in (3).
In this step, a single 3×3 NE block is necessary and the computation is similar to the Gaussian filter, based on linear filtering. The two Sobel kernels work in parallel processing the gradients G_x and G_y .

Magnitude & Direction: The magnitude and direction equations, defined in (7) and (8), are quite expensive to implement on hardware. To avoid these complex computation tasks, we implemented the approximation solutions proposed in [7]. These

solutions are defined in (19) and (20).

$$|G| \approx |G_y| + |G_x| \quad (19)$$

$$\Phi \approx \begin{cases} 90^o & |Gy| > |G_x| \\ 0^o & |Gy| > |G_x| \end{cases} \quad (20)$$

Non-maximum suppression: The Non-Maximum Suppression (NMS) step eliminates pixels with gradient magnitude smaller than adjacent pixels in the gradient direction. Fig. 12 shows the NMS hardware architecture.

Hysteresis thresholding: The final step is the hysteresis thresholding where two different thresholds T_{high} and T_{low} are applied to the input image T_{high} saturates every pixel with a gradient value greater than its threshold value. T_{low} bypasses every pixel with a gradient value greater than its threshold value. The output of these two blocks are added up, resulting in a stream where the saturated pixels are considered part of the edges and the other pixels different than zero are considered edge candidates. Then, a sequence of operators test all pixels within the image to determine if edge candidates are connected to edge pixels for reducing the fragmentation of contours in the edge map. The connector blocks test if at least one of the eight neighborhood pixels of an edge candidate is a true edge. If the test is positive, the edge candidate is marked as a true edge. To improve the efficiency of this test, the connector blocks

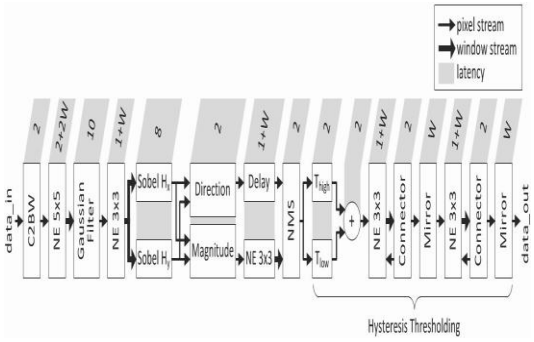


Figure 10: Functional diagram of the canny edge detector processing pipeline where the latency is indicated on the top of each stage and W is the number of pixels per image line.

$$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Figure 11: Kernel coefficients of the Gaussian filter with the origin coordinates (i, j) at the central pixel.

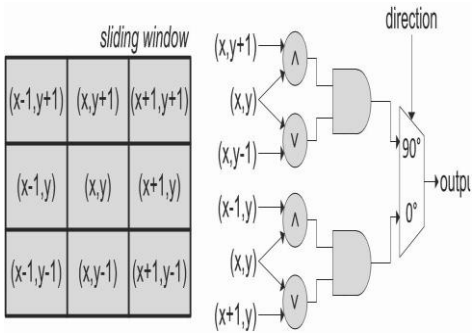


Figure 12: Non-maximum suppression hardware architecture.

Utilize recursive NEs, as described in Section 5.2, and mirror blocks to invert the image scanning direction. The mirror blocks allow edge candidates to be tested in both directions, right-to-left and left-to-right. Similarly to the NE block, the mirror block was de- signed to support images with variable resolutions. The architecture of the mirror block, shown in Fig. 13, is similar to the line buffer architecture (Fig.9). The main difference is that the mirror block has two RAM blocks. While one RAM block is storing the current input line, the other outputs the previous line in a last-in-first- out (LIFO) fashion. When the line is finished, the RAM blocks change their roles and the process starts again. The Mirror Controller generates all the controlling signals, including the read and writes addresses for both RAM blocks. In terms of latency, the Mirror block minimizes the number of pixels buffered before it starts sending its results. The latency in pixels of the Mirror block is equivalent to one image line size (W).

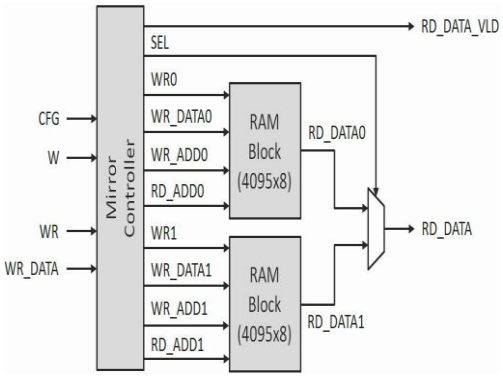


Figure 13: Architecture of the mirror block

5.4 Harris Detector

The Harris corner detector processing pipeline, shown in Fig. 14, is based on the original Harris algorithm presented in Section 3.2. Below, we present the FPGA implementation of this algorithm, divided into five steps.

Color to gray scale: The first step of this implementation is identical to the one presented in the previous section for the Canny implementation.

Spatial derivative computation: This step computes the first derivatives $I_x(u, v)$ and $I_y(u, v)$ of the input image $f(u, v)$ by applying the approximations presented in (9) and (10).

Building the matrix M: In this step, the values A, B, and C, defined in (12), (13), and (14), are computed to build the matrix M, defined in (11). Three sub-pipelines are applied in parallel to perform these computations.
Each sub-pipeline is formed by a multiplier, a 5×5 NE block, and a Gaussian filters.

Harris response: The Harris response operator computes the values of R, defined in (15). To keep the pixel stream within an 8-bit resolution without losing weak corner values, R is truncated at 255. This approach can create large regions around the corner spot with saturated values, making difficult the following NMS process.
To solve this, a threshold block eliminates low R values that do not represent corners followed by an extra Gaussian filter to blur these saturated regions, producing a maximum spot at the center of these regions.

Non-maximum suppression: The final step is to select

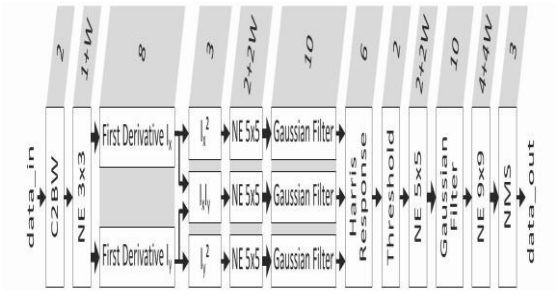


Figure 14: Functional diagram of the Harris corner pixels is indicated on the top of each stage and detector processing pipeline where the latency in W is the number of pixels per image line.

6. RESULTS AND ANALYSIS

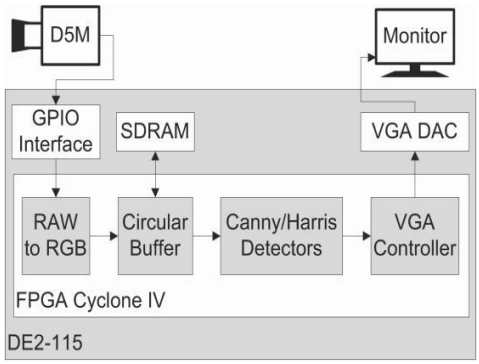


Figure 15: Simplified functional diagram of the proposed architecture evaluation system.

The best values representing corners. To do NMS block analyses a region (window) and maximum value as a detected corner In order to evaluate the proposed architecture we have implemented it in the Altera development board DE2-115 containing a Cyclone IV EP4CE115 FPGA device along with a digital camera daughter board D5M. The complete system works as A stand-alone 480 kpixel digital camera where the proposed architecture is embedded, working on image sequence mode.
Diagram of the complete system where a Circular buffer is placed between the this, a 9×9 and filter (RAW to RGB converter) marks the architecture (Canny/Harris Proposed detectors) in order to detach the frame rate from the input frame rate.

In terms of latency, we can define two types of latency, the Initial Latency (IL) and the Processing Latency (PL). The IL is defined here as the amount of time between when the first pixel arrives at the input of the system and when it is received at the system's output, i.e., it corresponds to the time expended to fill the pipeline. The PL is defined here as the sum of the IL and the time to process all the pixels of an image. Since our proposed architecture works at the same pixel rate as the input pixel stream and considering that the input pixel stream has a constant rate, we can express IL and IP in terms of pixels, as shown in Fig. 10 and Fig. 14. Based on these

TABLE 1: Canny edge detector timing performance

Image resolution (W × H)	CPU (ms)	GPU (ms)	FPGA [a] (ms)
512×512	30	2.11	1.10
1024×1024	101	6.08	4.37
1476×1680	267	13.90	10.31
3936×3936	1497	59.94	64.16

\

A FPGA working at 242 MHz (Fmax) definitions, we can write equations (21) and (22).

TABLE 2: Harris corner detector timing performance

Image resolution (W × H)	CPU (ms)	GPU (ms)	FPGA (ms)
512×512	20	2.32	1.15
1024×1024	60	4.49	4.56
1476×1680	171	13.16	10.75
3936×3936	1402	64.41	66.93

$$P\,L_{Canny} = W \times H + (37 + 8W) \quad (21)$$

$$P\,L_{Harris} = W \times H + (53 + 9W) \quad (22)$$

Where W × H is the dimensions of the image and the expression within parentheses is the IL. The processing time can be obtained by dividing PL by the pixel rate. Timing performance comparisons between three different platforms are shown in Table 1 for the canny edge detector and Table 2 for the Harris corner detection. In order to provide results from a more up-to-date tech- nology than the 60 nm Cyclone IV, we have synthesized both Canny and Harris detectors targeting the 28 nm Arria V 5AGXFB3 FPGA device, which is part of the latest midrange FPGA family from Altera. The other platforms utilized are: a CPU Intel Core2 Duo E6600, 2.4 GHz; and a GPU GeForce GTX 580, 1.54 GHz. In this analysis, the frequency of the FPGA implementation is the maximum frequency (Fmax) obtained during synthesis using the tool Quartus II v12.1. Tables 1 and 2 show that the FPGA has an evident advantage over the CPU implementation. The FPGA speedup factor for the CPU implementation varies from

TABLE 3: Comparison of power and energy consumption for the canny edge detector implementations

Image resolution (WxH)	CPU (W)	CPU (J)	GPU (W)	GPU (J)	FPGA (W)	FPGA (mJ)
Standoff	136	-	229	-	0.9	-
512X512	141	4.2	231	0.5	1.5	1.6
1024x1024	147	14.8	244	1.5	1.5	6.4
1476x1680	149	39.8	248	3.4	1.5	15.0
3936x3936	153	229.0	251	15.0	1.5	93.6

23.1 to 27.2 with the canny algorithm, and from 13.2 to 20.9 with the Harris algorithm. Regarding the GPU implementation, the FPGA presented an advantage on 512×512, 1024×1024, and 1476×1680. For larger images however, the GPU has an increasingly better performance in function of the image resolution, while the FPGA has a constant performance. This advantage of the GPU in terms of is due to its high number of CUDA cores (512 in a GeForce GTX 580). Indeed, the use of high definition images enables more CUDA threads to be launched so that each one can treat one or a group of pixels, which offers a massively parallel processing. Moreover, the treatment of large images enables the use of GPU to be increased at the expenses of data transfers between CPU and GPU memories. An efficient exploitation of GPU requires the application of a highly intensive processing (in parallel) of large datasets (im- ages). The treatment of low resolution images on GPU is hampered by the cost of data transfers between CPU and GPU memories. These costs can be neglected when processing high definition images since the treatment will be accelerated by launching many CUDA threads in parallel.

In addition to its competitive performance, the FPGA implementation can still offer portability and much lower power consumption when compared to GPUs and CPUs, as we can see in Tables 3 and 4. Although both FPGA implementations perform the same computations as the concurrent architectures, the FPGA solutions consume from 94 to 151 times less power than the CPU implementation and from 154 to 254 times less power than the GPU implementation. In terms of energy efficiency, the figures are even better compared with competitive architectures. The FPGA implementations are from 1316 to 2652 times more efficient than the CPU solution and from 161 to 315 times more efficient than the GPU solution. It is also important to highlight here that the GPU, despite being more power consuming than the CPU, it is more energy efficient than the CPU due to its higher performance. Regarding resource utilization, the FPGA Canny de- tector version occupies only 3 % of the Arria V 5AGXFB3 resources and the Harris detector occupies 7 %, as de- scribed in Table 5 The proposed Canny and Harris detectors were also

TABLE 4: Comparison of power and energy consumption for the Harris corner detector implementations

Image resolution (WxH)	CPU (W)	(J)	GPU (W)	(J)	FPGA (W)	(mJ)
Standoff	136	-	229	-	1.1	-
512X512	141	2.8	231	0.5	1.5	1.7
1024x1024	147	8.8	240	1.1	1.5	6.7
1476x1680	147	25.1	242	3.2	1.5	15.8
3936x3936	152	213.1	249	16.0	1.5	98.2

TABLE 5: FPGA resources utilization in the Canny edge detector and Harris corner detector implementations. The numbers within parentheses correspond to the percentage of use in the Arria V 5AGXFB3.

Memory Blocks	Algorithm	ALM[a]	Register (kb)	DSP
	Canny ED	3406 (2)	533 (3)	28 (3)
	Harris CD	8624 (6)	17137	863 (5) 76 (7)

Adaptive Logic Modules. Evaluated in terms of efficiency and noise tolerance. In these tests, different levels of Gaussian noise were added to the original image. Fig 1shows the proposed canny edge detector results in an image degraded by Gaussian noise. In this figure, the images of the edges were inverted for better visualization.

Thgraph in Fig 17 compares our results and the results of aCanny detector provided as a plugin of the ImageJ tool [39], called Feature J [40]. These results show that the proposed canny detector has a similar response to the analogous implementation in software, demonstrating the efficiency of the architecture despite the algorithmic simplifications. Results also show that our system can eliably detect edges in noise degraded images down to 20 dB of SNR, where many false edges start appearing.

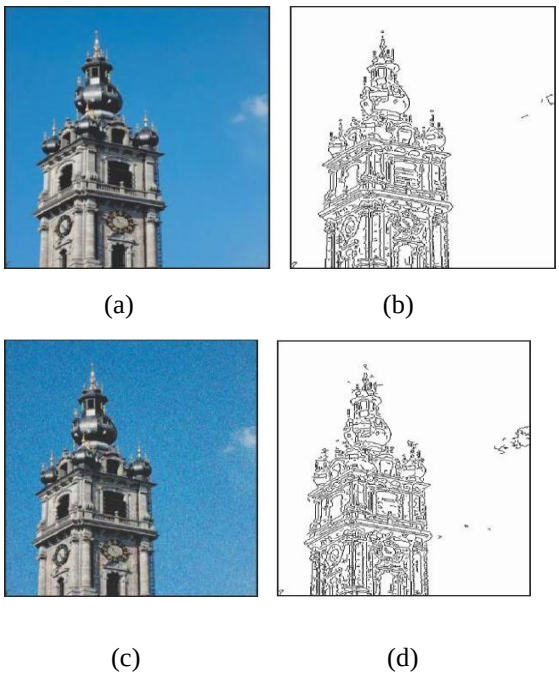
The same idea of comparing SNR degraded image resolutions was used to test the Harris corner detector. Instead of computing the SNR of the output image with corner detection, the number of corners detected was analyzed and compared to the number of corners detected in the original image. Fig. 18 shows the

Harris corner detector results on an image degraded by additive Gaussian noise. The graph in Fig. 19 shows the relation between the number of corners detected and the image degradation level. We can see that Corner detector has a reasonable number of corners detected in images de- graded down to 30 dB SNR. After this point, the number of false positive corners increases significantly.

7. CONCLUSIONS AND FUTURE WORK

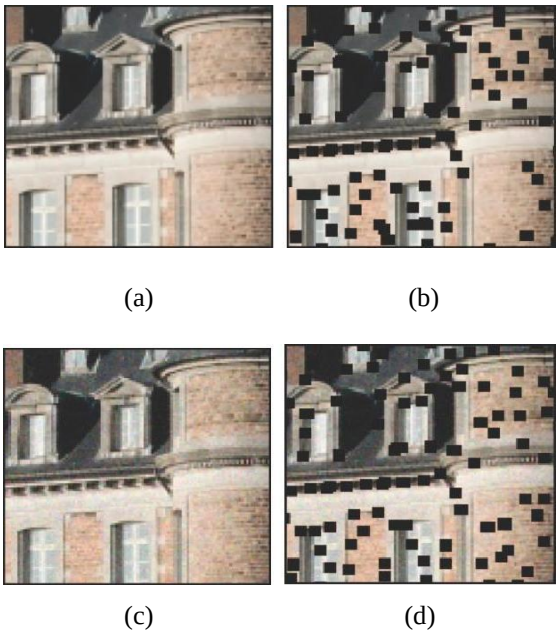
In this paper, we have presented a new flexible architecture for Canny and Harris feature detectors. This new Architecture has a reduced latency and memory requirement supporting images with variable resolutions. The key component in this architecture is the NE that can be parameterized on-the-fly based on the image characteristics. Some simplifications in the algorithms that reduce mathematical complexity, latency, and Memory requirements are also presented in this paper.

The proposed architecture was evaluated on an FPGA- based platform and the results have shown the efficiency of the NE block and the algorithm simplifications that



(a) (b) (c) (d)

Figure 16: Canny edge detector results in an image de- graded by Gaussian noise. (a) Original image; (b) edges detected in (a); (c) noise-degraded image (SNR = 20 dB); (d) edges detected in (c)



(a) (b) (c) (d)

Figure 18: Harris corner detector results on an image de- graded by Gaussian noise.(a) original image(detail); (b) corners detected in (a); (c) noise-degraded image (SNR = 30 dB); (d) corners detected in (c).

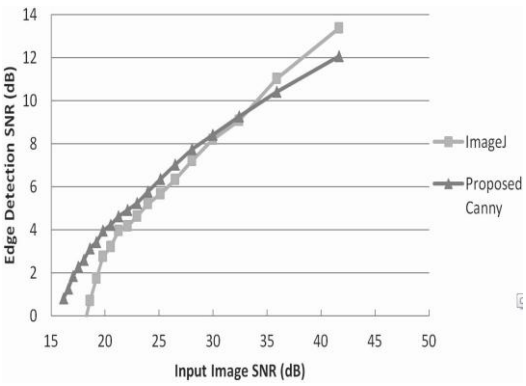


Figure 17: Edge detection degradation levels vs. Input image degradation levels.

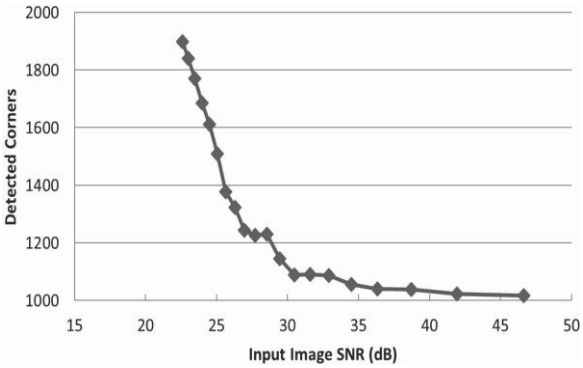


Figure 19: Number of corners detected vs. Image degradation levels.

Did not significantly change the algorithm's reliability. The results have also shown that the proposed architecture presented a very competitive performance com- pared with the analogous implementation in a GPU. The FPGA implementation can deliver a maximum through put of 242 Mpixel/s and 232 Mpixel/s in the Canny and Harris detectors implementation, respectively. This performance is sufficient to support high definition (HD) formats, including Full HD streams in a 1080p60 format (resolution of 1920×1080 pixels at a rate of 60 progressive frames per second). Furthermore, it has a clear advantage in applications where low power consumption, low latency, and portability are required.

Future work will be devoted to increasing the flexibility level of the architecture including a reconfigurable interconnection between the building blocks in such a way that several different processing pipelines can be created at runtime. In this way, a single architecture can be used for a wide range of image and video processing algorithms. An extension of this work will be the design of a mapping method to try to reduce the application development time. Another extension will be the addition of a histogram analysis module to automatically adjust threshold levels and/or input image equalization Histogram analysis demands at least one pre-scan on the input image which could significantly increase the latency of the system. However, considering that in video processing the input context will not drastically change between two consecutive frames, it is possible to use the histogram analysis of one frame to adjust the architecture for the next frame, without increasing the latency.

ACKNOWLEDGMENTS

This work is supported by the French Community of Belgium under the Research Action ARC-OLIMP (Optimization for Live Interactive Multimedia Processing (2003-13)

REFERENCES

[1] R. Maini and H. Aggarwal, Study and comparison of various image edge detection techniques,‖ International Journal of Image Processing (IJIP), vol. 3, no. 1, pp. 1–12, 2009.

[2] C. Harris and M. Stephens, ―A combined corner and edge detec- tion,‖ in Proceedings of The Fourth Alvey Vision Conference, 1988, pp. 147–151.

[3] S. Gauglitz, T. Hllerer, and M. Turk, ―Evaluation of interest point detectors and feature descriptors for visual tracking,‖ International Journal of Computer Vision, vol. 94, no. 3, pp. 335– 360, Mar. 2011.

[3] J. Fung, S. Mann, and C. Aimone, OpenVIDIA: Parallel GPU computer vision,‖ In Proc of ACM Multimedia, pp. 849–852, 2005.

[5] Khronos-Group, ―The open standard for parallel programming of heterogeneous systems, ‖ 2009. [Online]. Available: http: //www.khronos.org/opencl

[6] W. He and K. Yuan, An improved Canny edge detector and its realization on FPGA,‖ in Intelligent Control and Automation, 2008. WCICA 2008. 7th World Congress on, june 2008, pp. 6561 –6564.

[7] Q. Xu, C. Chakrabarti, and L. J. Karam, ―A distributed Canny edge detector and its implementation on FPGA,‖ in Digital Signal Processing Workshop and IEEE Signal Processing Education Workshop (DSP/SPE), 2011 IEEE. IEEE, Jan. 2011, pp. 500–505.

[8] K. Pauwels, M. Tomasi, J. Diaz Alonso, E. Ros, and M. Van Hulle, A comparison of FPGA and GPU for real- time phase-based optical flow, stereo, and local image features,‖ Computers, IEEE Transactions on, vol. 61, no. 7, pp. 999 – 1012, july 2012.