STATUS OF THESIS

Title of thesis | OPTIMIZATION OF FPGA-BASED PROCESSOR ARCHITECTURE FOR SOBEL EDGE DETECTION OPERATOR

I ZAHRAA ELHASSAN MOHAMED OSMAN ,

hereby allow my thesis to be placed at the information Resource Center (IRC) of Universiti Teknologi PETRONAS (UTP) with the following conditions:

1. The thesis becomes the property of UTP
2. The IRC of UTP may make copies of the thesis for academic purposes only.
3. This thesis is classified as

☐ Confidential

☑ Non-confidential

If the thesis is confidential, please state the reason:

_____

_____

_____

The contents of the thesis will remain confidential for _____years.

Remarks on disclosure:

_____

_____

_____

Endorsed by

_____ _____

Signature of Author | Signature of Supervisor

Permanent address: | Dr. Fawnizu Azmadi Hussin

Dept. of Computer Eng.,

Faculty of Eng. and Tech.,

University of Gezira,

P.O. Box 20 - Wad Medani,

Sudan.

Date: _____ | Date: _____

UNIVERSITI TEKNOLOGI PETRONAS

OPTIMIZATION OF FPGA-BASED PROCESSOR ARCHITECTURE FOR SOBEL EDGE DETECTION OPERATOR

By

ZAHRAA ELHASSAN MOHAMED OSMAN

The undersigned certify that they have read, and recommend to the Postgraduate Studies Programme for acceptance this thesis for the fulfillment of the requirements for the degree stated.

Signature: _____

Main Supervisor: Dr. Fawnizu Azmadi Hussin

Signature: _____

Co-Supervisor: Dr. Noohul Basheer Zain Ali

Signature: _____

Co-Supervisor: Dr. Likun Xia

Signature: _____

Head of Department: Assoc. Prof. Dr. Nor Hisham Bin Hamid

Date: _____

OPTIMIZATION OF FPGA-BASED PROCESSOR ARCHITECTURE FOR SOBEL EDGE DETECTION OPERATOR

By

ZAHRAA ELHASSAN MOHAMED OSMAN

A Thesis

Submitted to the Postgraduate Studies Programme
as a Requirement for the Degree of

MASTER OF SCIENCE

ELECTRICAL AND ELECTRONICS ENGINEERING

UNIVERSITI TEKNOLOGI PETRONAS

BANDAR SRI ISKANDAR

PERAK

MAY 2011

DECLARATION OF THESIS

Title of thesis | OPTIMIZATION OF FPGA-BASED PROCESSOR ARCHITECTURE FOR SOBEL EDGE DETECTION OPERATOR

I ZAHRAA ELHASSAN MOHAMED OSMAN ,

hereby declare that the thesis is based on my original work except for quotations and citations which have been duly acknowledged. I also declare that it has not been previously or concurrently submitted for any other degree at UTP or other institutions.

Witnessed by

_____ _____

Signature of Author
Permanent address:
Dept. of Computer Eng.,
Faculty of Eng. and Tech.,
University of Gezira,
P.O. Box 20 - Wad Medani,
Sudan.
Date: _____

Signature of Supervisor
Dr. Fawnizu Azmadi Hussin

Date: _____

ABSTRACT

This dissertation introduces an optimized processor architecture for Sobel edge detection operator on field programmable gate arrays (FPGAs). The processor is optimized by the use of several optimization techniques that aim to increase the processor throughput and reduce the processor logic utilization and memory usage. FPGAs offer high levels of parallelism which is exploited by the processor to implement the parallel process of edge detection in order to increase the processor throughput and reduce the logic utilization. To achieve this, the proposed processor consists of several Sobel instances that are able to produce multiple output pixels in parallel. This parallelism enables data reuse within the processor block. Moreover, the processor gains performance with a factor equal to the number of instances contained in the processor block. The processor that consists of one row of Sobel instances exploits data reuse within one image line in the calculations of the horizontal gradient. Data reuse within one and multiple image lines is enabled by using a processor with multiple rows of Sobel instances which allow the reuse of both the horizontal and vertical gradients. By the application of the optimization techniques, the proposed Sobel processor is able to meet real-time performance constraints due to its high throughput even with a considerably low clock frequency. In addition, logic utilization of the processor is low compared to other Sobel processors when implemented on ALTERA Cyclone II DE2-70.

ABSTRAK

Tesis ini memperkenalkan senibina pemproses yang optimal untuk alat pengesan sisi Sobel pada Field Programmable Gate Array (FPGA). Prosesor ini dioptimumkan dengan menggunakan beberapa teknik optimasi yang bertujuan untuk meningkatkan proses pengeluaran dan mengurangkan penggunaan logik pemproses serta penggunaan memori. Di samping itu, FPGA menawarkan keselarian peringkat tinggi yang dapat dimanfaatkan oleh pemproses untuk melaksanakan proses pengesanan sisi selari dengan meningkatkan pengeluaran prosesor dan mengurangkan penggunaan logik. Dengan itu, pemproses yang dicadangkan ini haruslah terdiri daripada beberapa blok. Sobel yang mampu menghasilkan piksel hasil pengeluaran secara selari. Penyelarian ini membolehkan data untuk digunakan semula di dalam blok pemproses . Selain daripada itu, prestasi pemproses ini berganda dengan faktor yang sama dengan bilangan blok Sobel yang terdapat di dalam blok pemproses itu.

Pemproses yang terdiri dari satu baris blokSobel mengeksploitasi penggunaan semula data di dalam satu baris piksel imej dalam pengiraan kecerunan mendatar. Penggunaan semula data di dalam satu dan beberapa garis imej dibolehkan dengan penggunaan pemproses yang dilengkapi dengan barisan berganda blok Sobel yang membolehkan hasil pengiraan kecerunan mendatar dan menegak diguna semula. Dengan pelaksanaan teknik optimasi ini, pemproses Sobel yang dicadangkan mampu memenuhi kehendak permintaan prestasi semasa/terkini . Hal ini adalah kerana kadar pengeluaran yang tinggi dengan frekuensi clock yang rendah. Akhir sekali, prosesor ini memerlukan pengunaan logik yang rendah berbanding dengan prosesor Sobel yang lain apabila ia diterapkan pada Altera Cyclone II DE2-70.

ACKNOWLEDGEMENTS

First, I would like to Thank ALLAH the almighty, for with his consent and blessing, the work of this thesis was completed.

Second, I owe my deepest gratitude to my parents and my family members for their immortal love and support.

It is my pleasure to thank my supervisor Dr. Fawnizu Azmadi Hussin who has been made available his help, courage, and support in many ways. My sincere gratefulness to my co-supervisor Dr. Likun Xia for his continuous help, encouragement, and guidance. A great appreciation to my co-supervisor Dr. Noohul Basheer Zain Ali for all his help and guidance.

Thanks and gratitude must be given to the graduation assistantship scheme and the Electrical and Electronics Department of the Universiti Teknologi PETRONAS for this great opportunity. Thanks extended to all the postgraduate office members for their help.

Heartfelt gratitude is extended to my father Mr. Elhassan Osman, my mother Mrs. Afaf Ibrahim, My uncle Alhussain Osman, my sisters Zuhal, Zeinab, Zulfa, Ameera, Melina and my bothers Mohamed and Mazin who without their love and support it would have been impossible for me to complete this work.

My sincere appreciation goes to my dear friend Mr. Ngo Huy Tan for his valuable help and support.

Last but not least, I would like to thank all my friends and colleagues who have made this journey very special and unforgettable.

DEDICATION

To My Beloved Parents and Family

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Introduction

Image processing systems are computational and data intensive. Such systems require high computing capabilities to meet real-time applications requirements. A typical digital video camera capturing high definition (HD) images with frame size of 640x480 pixels at 30 frames per second (fps) requires multiple processing stages at the rate of 27 million pixels per second (pps). This rate increases with the use of high definition television (HDTV) technology with a 1280x720 frame size and 30 fps rate and data rate that reaches 83 million pps. The computational burden increases with the frame size being increased. Although the processing power of general purpose processors is increasing rapidly, there is always a need for more processing power to match new image processing applications. Real-time video surveillance, real-time biometric (face and finger print) recognition and satellite surveillance applications are examples of such systems [1-2].

The use of parallel processing provides a solution for the problem of calculating the extremely large amount of data in image processing applications. Image and video processing systems are then implemented by taking advantage of the high level of parallelism within their operations.

The computational operations in a complete image/video processing system may be divided into two levels of processing operations, low level operations and high level operations. The operations in the lower level are more data intensive while the operations in the higher levels of processing are more control intensive. The low level

operations include image capture, noise reduction, image segmentation (e.g. edge detection) and followed by image analysis (e.g. feature extraction). The large amount of data encourages the use of parallel architectures to implement low level operations, (e.g. the use of dedicated hardware rather than using general purpose processors). The system back-end carries out high-level operations like image interpretation and image recognition to recognize image contents. The high-level operations require less data bandwidth and are characterized as control-intensive operations which promote the utilization of instruction-level of parallelism (e.g. the use of general purpose processors).

From the above, one can conclude that only one general purpose processor is not adequate to perform this variety of operations on such large amount of data in real-time. A more feasible system should contain a dedicated parallel processor at the front-end for the low-level processing of data, attached with a fast general purpose processor at its back-end to perform high level processing operations [3].

1.2 Real-Time Image/Video Processing

Real time image processing is defined as the digital processing of an image which occurs seemingly immediately, without a user-perceivable calculations delay [4]. A real-time system is one whose logical correctness is based both on the correctness of the outputs and their timeliness [5]. A common misconception about real-time systems is that their time constrains can be met by the use of the fastest and most powerful hardware. That is because these hardware solutions are not always feasible to implement the real-time systems, especially for embedded systems that have cost and power constrains to their applications such as camcorders and digital cameras.

1.3 Challenges In Real-Time Image And Video Processing Systems

Since a fast and powerful hardware alone does not have sufficient computation power to meet the time requirements of a real-time system, building a feasible design necessitates merging hardware and software design approaches. The hardware approach implies selecting the best hardware platform for the system. The software

approach entails meeting time constraints of the system by choosing between algorithms depending on the mathematical complexity, or tuning the algorithm to meet the system deadlines [2].

For a low level operation such as edge detection, the calculation of edges within the image pixels depending on the operator used (Robets, Prewitt, or Sobel) implies multiple levels of calculations to be applied on each pixel in the input image (refer to chapter 2). In images with large frame size, the ability of the system to deliver the output images in real-time (more than 30 frames per second) is a challenging process that can be achieved by merging the hardware and the software approaches mentioned above.

1.4 Hardware Platforms of Image and Video Processing Systems

Image processing operations are well known for their large amount of data and calculations; they require high computational capabilities, memory space, and processing time. Although today's general purpose processors are powerful, real-time image and video processing applications still need more computational power than what a general purpose processor can deliver. Such applications include real-time face/fingerprint recognition, real-time video surveillance, intelligent systems, military aerial and satellite surveillance systems. The architectures of the image processing systems should grant massive parallelism to keep pace with large amounts of data that are required to be processed in a certain time frame. Earlier real-time image/video processing systems were built using several parallel mainframes; which limited the use of these systems for portable applications [2]. A faster and more efficient alternative to general purpose processors is to implement image processing systems on dedicated hardware.

Image processing systems can be implemented using different commercial hardware platforms; such as digital signal processors (DSPs), the very expensive application–specific integrated circuits (ASICs), field programmable gate arrays (FPGAs), and recently graphic processing units (GPUs) [1-2]. Each of which has its advantages and disadvantages which makes the choice of the optimum platform

depend on the application requirements, solution availability, and cost.

An optimum choice of a hardware platform should produce a design with the following characteristics: high computational performance, flexible, easy to upgrade, low in development cost, and migration paths to lower costs as the application matures and volume ramps [6]. These characteristics are discussed here:

i. High performance: Low level operations such as color space conversion, filtering and edge detection demands higher performance than higher level operations do. Most of the platforms mentioned earlier in this section cannot keep pace with such performance with a single device. A single DSP processor running at 1 GHz cannot perform H.264 HD decoding or H.264 HD encoding, which is ten times more complex than decoding [6]. FPGAs are the only hardware solution that can satisfy these requirements with a single FPGA [6].

ii. Flexibility: Flexibility enables easy upgrading and decreases the time to market for the design. Using ASICs to implement designs with high volume consumer market will be very risky. The re-programmability of the FPGAs and their scalability make them a good solution for this kind of designs.

iii. Low development cost: the development of a 90-nm standard-cell ASICs costs around US$30 million. While a complete FPGA development kit could cost as low as US$1,095 [6].

iv. Migration path to lower unit costs: It is always important to have a design with low cost migration path. ASICs - due to their high cost -are feasible only for the highest volume consumer applications such as video and imaging applications, e.g., camcorders, monitors and LCDs, digital cameras, and cell phones. For applications with lower volume designs, FPGA is a better choice that eases the development of low cost designs with no risk of obsolescence[6].

Overall, FPGAs are probably the best candidates for image and video processing applications such as medical imaging, video surveillance and video conferencing [6]. Moreover, FPGA provides a high level of parallelism and greater I/O bandwidth to its embedded memory. This makes most of the image processing applications run faster on FPGA than on general purpose processors [1]. Since image edge detection is the basic operation for most of the applications mentioned above, this project focuses on designing an edge detection processor using FPGAs.

1.5 Design Trade-Offs

Designing an image processing system, particularly the one specialized for a low-level operation on a dedicated hardware with limited resources is not an easy procedure. It requires sacrificing some advantages for others, for instance memory space for processing speed and logic utilization for performance. A designer should keep in mind the necessary characteristics (mentioned in section 1.4) to achieve a feasible design and at the same time perform trade-offs between those characteristics to deal with the hardware limitations. This requires optimization of the image processing algorithm during the process of transforming the algorithm from a research development phase to a hardware working system [2].

1.6 Problem Statement

Edge detection is a low-level operation that requires large amount of computations and data which necessitate an architecture with high-level of parallelism to be able to satisfy the system's time constraints. Although FPGAs provide high parallelism and re-programmability, their logical and memory resources are limited. Since design parameters such as throughput, power and area are dependent on each other, several trade-off decisions should be made to the edge detection algorithm in order to find the optimum performance/area trade-off of the hardware design. An optimized system should be able to deliver a high throughput (parallelism) with the minimum amount of logic (area) and consumes less power (low frequency).

1.7 Research Objectives

This research aims to present an optimization of an edge detection processor based on Sobel edge detection operator (refer to Chapter 2) to be implemented on FPGAs. The research objectives of this project can be summarized as follows:

1. Design a dedicated edge detection processor based on Sobel operator.
2. Optimize the processor to increase the throughput and reduce the logic utilization.
3. Implement the processor design on FPGA.
4. Evaluate the processor performance for real-time system constrains and for logic utilization.

1.8 Scope of Research

The research focuses on optimizing the design by the simplification of the Sobel edge detection algorithm and the utilization of several optimization techniques. The optimization aims to minimize the logic and memory resources usage and hence the power consumption. The optimization intends to maximize the processor throughput and frame rate while using a considerably low frequency by increasing the processor parallelism.

1.9 Research Methodology

This research has evolved through three different phases. The work flow description of each phase is stated as follows:

First Phase:

- Literature review on image edge detection operation and the operator used to detect edges in digital images.
- Review on how edge detection systems based on Sobel operator are designed to be implemented on FPGAs and the pros and cons of each design technique.

- Review of general optimization techniques used to optimize digital systems architectures for speed and area.

Second Phase:

- Implementation of a basic Sobel edge detection processor on FPGA.
- Analysis of the design in order to highlight how the design is to be optimized.

Third Phase:

- Implementation of the optimization techniques to optimize the basic Sobel edge detection processor architecture.
- Implementation of the optimized Sobel edge detection processor on FPGA.
- Evaluate the processor logic utilization
- Evaluate the optimized Sobel processor performance.
- Compare with other Sobel processors mentioned in the research literature.

1.10 Research Contributions

This research proposes an optimized Sobel edge detection processor to be used in real-time edge detection application. The expected contributions of the thesis can be summarized as follows:

i. The Sobel algorithm is simplified to reduce the number of operations performed by the processor block which reduces the logic utilization caused by the loop unrolling and reduces the processor size and execution time.
ii. The processor employs a specially designed line buffer that delivers the input data in a way that matches the optimized Sobel processor.
iii. The optimization reduces the processor size significantly compared to other Sobel processor architectures.

1.11 Thesis Organization

This thesis is divided into five chapters where chapter 1 highlights the general characteristics of image processing systems operations. This is followed by a brief introduction on why these systems have better performance on dedicated hardware platforms than general purpose processors. The need for design trade-offs and optimization to deal with resources limitation when using FPGA as a hardware platform for image processing systems is also spotted. The chapter summarizes the thesis contents and underlines the objectives of the research, the methodology used to achieve these objectives and the research contributions.

Chapter 2 reviews image edge detection as an important image segmentation technique followed by some of the most famous edge detection operators. This chapter shows how these image processing algorithms can be simplified and optimized using different optimization techniques to implement real-time systems. The chapter also reviews several examples of edge detection systems implemented on FPGA as part of this research literature.

Chapter 3 introduces the implementation of a basic Sobel processor architecture on FPGA board followed by analysis of the basic processor performance. The basic optimization techniques are then used to optimize the basic processor design and develop the proposed architecture. The processor is integrated in a complete FPGA based edge detection system to be tested for real-time performance.

In chapter 4, the Optimized Sobel processor is implemented on FPGA. The logic utilization and the performance of the processor are evaluated. A comparative analysis is performed between the optimized Sobel processor and other FPGA based Sobel processors to show the effect of the optimization applied on the processor.

Chapter 5 summarizes the contribution of the previous chapters. Possible future extensions of the research area are discussed.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

In digital image processing, the image undergoes different processing levels with different kinds of operations in each level. Some of these operations aim to produce a more descriptive version of the original image, so called image analysis operations. Other operations highlight some features in the image or improve the image in a certain way, and those are called image enhancement operations. Image enhancement and image analysis are classified under low-level processing and named as pre-processing operations. The pre-processed image contents can be interpreted using image recognition which is classified as the high-level processing operations. Image recognition is also called image understanding or computer vision and it belongs to the area of Artificial Intelligence (AI).

This chapter discusses edge detection as a low-level image processing operation that may be used to analyze or enhance digital images and how edge detection algorithms can be optimized using several optimization techniques. The chapter also discusses how Sobel edge detection operator is implemented on FPGA using several design methodologies.

2.2 Pre-Processing of Digital Images

2.2.1 Grayscale Transformation

It is one of the most common and simple preprocessing operations performed on images to transform a colored image to a grayscale representation as shown in Fig 2.1 [7].

a) RGB image

b) Grayscale image

Figure 2.1The effect of grayscalling on RGB colored images

Figure 2.1 a) represents a colored image which consists of three layers or channels; Red, Green and Blue (RGB). The sensitivity of the human visual system (HVS) to each layer varies significantly. The human eye is most sensitive to green light and least sensitive to blue light. The sensitivity level can be weighted by the vector $[0.299 \quad 0.578 \quad 0.114]$ for the red, green and blue channels, respectively. In digital images, a pixel in a colored image has three values, one for each of the layers. While in grayscale image as in the Fig 2.1 b), the pixel has only one value that represents the gray shade or the image luminance at that pixel position. Processing colored images requires the processing of each layer separately. This can take three times of the memory-space and the processing time to process and store a grayscale image which has only one layer. Grayscaling reduces memory utilization, computations and speedup processing time. It linearly transforms 24-bit colored image (8-bits per channels) to an 8-bit gray-level image.

Grayscalling may be achieved in many ways. By using the equation (2.1), a colored image can be transformed into a grayscale one by taking the average of the three channels (RBG). All grayscaled channels (R', G', and B') will have the value of the gray level Y [8]:

$$Y = 0.333*R + 0.333*G + 0.333*B \tag{2.1}$$

$$Y = R' = G' = B'$$

Another way is to take the green channel as the gray level, because the human eye is most sensitive to the green color. A more accurate transformation can be achieved by using the luminance model shown by the equation (2.2), also referred to as the international telecommunication unit (ITU) standard [9]:

$$Y = 0.299*R + 0.587*G + 0.114*B \quad (2.2)$$

$$Y = R' = G' = B'$$

The equation results in better accuracy and is used by the National Television System Committee (NTSC) and Joint Photographic Experts Group (JPEG). The floating point calculations can be avoided by multiplying the equation (2.2) by 32,768 (or 2^{15}) without losing the accuracy. The produced grayscale Y is divided by the same scale factor 32,768 (or 2^{15}) to maintain the same dynamic range. The division may be replaced by shifting the result 15 times, since shifting is faster and more efficient as in equation (2.3) [9].

$$Y = (9798*R + 19{,}235*G + 3736*B) >> 15 \quad (2.3)$$

$$Y = R' = G' = B'$$

After the image is pre-processed by grayscalling, it is ready for analysis. The next section discusses image segmentation approaches that are used to analyze digital images.

2.3 Image Segmentation

Analyzing a digital image requires performing operations that result in another image which is a detailed description of the scene contents of the input image. Image analysis operations involve dividing image spatial domain into meaningful parts or regions [10]. This way of dividing images is referred to as *image segmentation*. The segmentation algorithms use image physical features to extract the meaningful regions. The performance of the algorithm is then measured by how meaningful the results are. The algorithms are based on one of the following approaches [10]:

a. Find areas with homogeneous feature(s) over a large region in the image.
b. Detect abrupt changes in the image feature(s) within a small neighborhood.

In the first approach, the result image contains regions with homogeneous common features. The second approach results in borders between two regions with different features in the image, e.g., the borders between an object and its background. The second approach is referred to as edge detection, which is explained in section 2.3.2.

The next section introduces one of the most important segmentation algorithms, which employs the image segmentation approach (a), i.e., Thresholding.

2.3.1 Thresholding

Thresholding or binarization is the simplest way of image segmentation which involves the operation of dividing the image into several segments. Segmentation implies reduction of image data to simplify further processing operations. Binarization transforms an image from 8/16/32 bit image to 1-bit image (0 or black, 1 or white). A good segmentation can give a good idea about the shape of the object(s) in an image and makes future processing much easier [7, 9].

Thresholding indicates the comparison between the image pixels and the threshold value T=t. The pixel value is set to 0 (black) if it happens to be less than the threshold value, otherwise it set to 1 (white). The thresholding result is a binary image with white and black pixels. The effect of thresholding on the cheetah's image in Fig 2.1 a) is shown in Fig. 2.2.

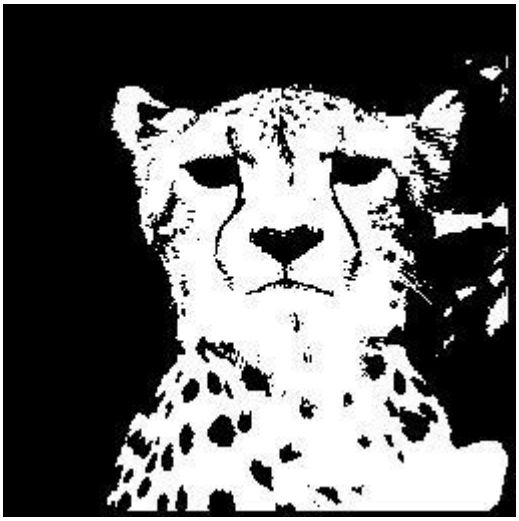


Figure 2.2 The effect of thresholding

Pixels in dark areas in the cheetah picture, i.e., the picture background, cheetah's spots, eyes and nose have values less than the value of the threshold, so they turn into black in the output image. The brighter areas in the image will all be white as their pixel values are larger than the threshold value.

A proper threshold value determines how good the segmentation result is. The value differs from image to image depending on many parameters, e.g. image brightness. A value of 128 is adequate when the image has two clearly different objects that are dark and bright, respectively. Otherwise, it will not give an ideal segmentation. Setting a fixed value for all images is not an efficient way for thresholding because it has no consideration for images frequencies [9].

An image histogram (graphical representation of the luminal distribution in a digital image that plots the number of pixels in each tone) can be used to manually choose a threshold value according to the image requirements. The threshold value will be defined as the lowest peak in the histogram. Unfortunately, the user has to define the threshold manually for each image and it is not easy to determine the exact threshold value from a plotted histogram.

A better way is to use the Basic Global Threshold (BGT) method. The method is able to calculate the threshold value automatically within the algorithm [7] and make it easier to use thresholding in image processing modules [9]. The BGT method performs a series of iterations in order to calculate the threshold value. The iterations

consist of the following steps:

i. Pick an initial thresholding value t (e.g. 128), or use the mean value of the image pixels.
ii. Calculate mean values of the pixels (m_1) below and (m_2) above the threshold t.
iii. Calculate new threshold value $t_{new} = (m_1 + m_2)/2$.
iv. If the threshold has stabilized ($t = t_{new}$) then this is the new threshold value, T. Otherwise T becomes t_{new} and reiterate from step 2.

2.3.2 Image Edge Detection

Image edge detection is one of the most important basic operations in image processing. It is considered as a type of image segmentation techniques that determines the presence of edges in an image and outlines them in an appropriate way [11]. An edge can also be defined as an abrupt change in the image intensity [7]. Edge detection algorithm produces an image with only the edges contained in the original image. Edge detection is important for most of subsequent higher-level vision tasks such as boundary detection, motion detection/estimation, texture analysis, segmentation, and object identification [7].

In digital image processing, images are quantized into pixels. In grayscale images, the values of the pixels indicate the brightness of the picture at each pixel position. Since edges are areas within the image with high contrast, the existence of an edge in a pixel may be obtained by examining abrupt changes in the brightness of neighborhood pixels. If the neighborhood pixels have almost the same brightness, probably there is no edge in that pixel. Otherwise, the pixel may contain an edge.

Edge detection can be accomplished in a wide range of methods. All edge detection methods can be categorized into two categories: gradient and Laplacian. The gradient method uses the maximum and minimum of the first derivative of the image. The Laplacian detects edges by looking for zero crossing in the second derivative of the image.

The simplest way to obtain image gradient is to find the differences in pixel values in the horizontal and vertical directions as follows:

X difference is calculated as $| f(x+1, y) - f(x, y) |$

Y difference is calculated as $| f(x, y+1) - f(x, y) |$

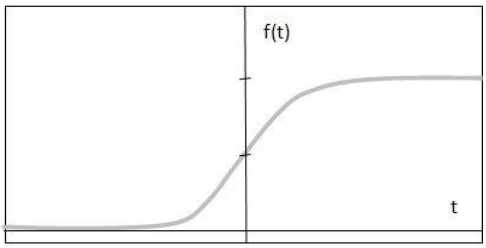
where: $f(x,y)$indicates the intensity values of the pixels [0-255] at (x,y) position [10].

A digital image may contain variety of contents, e.g., solid objects, lines and single points (noise). Each of which represents a different form of edges. Edges in digital images may have three different forms, Step, ramp, and roof edges. Roof edges occur in digital images that contain thin objects (e.g., lines) that have different intensity than their background. Roads in satellite images can be a good example of roof edges modeling. A step edge is modeled by an ideal transition of image intensity occurring by a distance of one pixel [7]. That implies only in computer generated digital images that are free of noise. In real life, digital images are noisy and blurred which makes the transition between a solid object and its background a bit blunt. This smooth transition is modeled by a ramp shape edge. As illustrated in Fig. 2.3 (a), an edge with one dimensional shape of ramp function is demonstrated. Calculating the derivative of the image can highlight edges locations in the image [7]. The gradient of the ramp function shown in Fig. 2.3 a) is the first derivative with respect to t (in one-dimension). This results in the maximum value of the edge signal as shown in Fig. 2.3 b). The first derivative is positive at the start of the ramp and on the ramp curve and zero otherwise. The derivative shows a maximum value located at the center of the edge original signal. The second derivative of a gray-level profile is positive at the leading edge of the transition, negative at the trailing part of the edge, and zero otherwise. From the above, the following facts may be concluded. First, the magnitude of the first derivative can be used to localize edges in digital images. Second, the second derivative produces two different values for each edge and the sign of the edge determines the edge pixel location (on the dark side or the bright side of the edge). This characteristic of the second derivative can be utilized to locate the centers of thick edges.

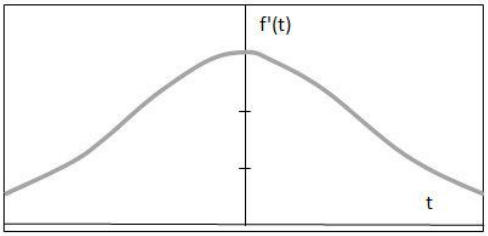
First derivative is a characteristic of the "gradient family" of the edge detection filters. The pixel is considered as an edge if its gradient value exceeds some threshold value. On the other hand, the second derivative of a maximum first derivative will

result in zero which enables detecting edges by finding the zero of the second derivative. This method is called Laplacian and the result graph is shown in Fig.2.3 (c). Laplacian is infrequently used for edge detection directly because it is too sensitive to noise and it results in doubled edges and it has no consideration for the edge direction. Laplacian is usually combined with other edge detection methods to create extra powerful edge detection effect. For example, the double edges produced by Laplacian can be used to locate the edge between double edges.

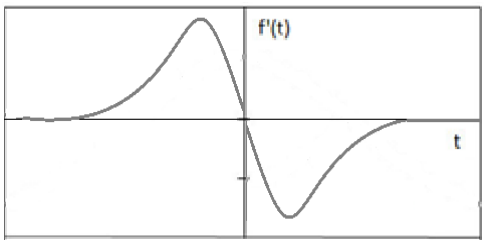
This thesis concentrates on the edge detection operators that use the first derivative gradient to locate the edges. Therefore, no further explanation of edge detection by Laplacian is included.



a) Original edge signal (ramp)



b) First derivative (Maximum value)



c) Second derivative (zero crossing)

Figure 2.3 The first and Second derivatives of a ramp function

The next section describes the mathematical procedure of calculating the gradient of an image and the commonly used gradient edge detection operators.

2.3.2.1 *Gradient Operators*

The gradient of an image *f*(*x*,*y*) at a location (*x*,*y*) is the vector G that its components are the partial derivatives of *f* along *x* and *y* axes as in equation (2.4):

$$\text{G}= \nabla f = \begin{bmatrix} Gx \\ Gy \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \tag{2.4}$$

The gradient vector points to the max rate of change of the image *f* at the location (x,y). The magnitude of the vector is given by the equation (2.5) and its direction is given by equation (2.6).

$$G= \text{mag}(\nabla f) = \sqrt{Gx^2 + Gy^2} \tag{2.5}$$

$$\theta = \tan^{-1}\left[\frac{\text{Gy}}{\text{Gx}}\right] \tag{2.6}$$

The magnitude equals to the maximum rate of increase *f*(*x*,*y*) per unit distance in the direction of ∇f. The gradient can also be described by the sum of absolute magnitudes of *Gx* and *Gy* as shown in equation (2.7), which makes it easier to be implemented on hardware.

$$G= \text{mag}(\nabla f) \approx |Gx| + |Gy| \tag{2.7}$$

The gradient of an image is computed by obtaining the partial derivative of *x* and *y* as $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$, respectively, at each pixel location. Since derivatives are linear and shift-invariant, gradient calculation is usually achieved by convolution. That is realized using an edge detection filter (operator). The operator consists of a small matrix whose elements are the filtering coefficients. This matrix is called the filter mask. The filter mask will be convolved over the image pixels to calculate the gradient for each pixel. The result is a gradient image as the same size as the original image as shown in Fig 2.4. Thresholding is then implemented on the gradient image

to produce the final edge detection result.

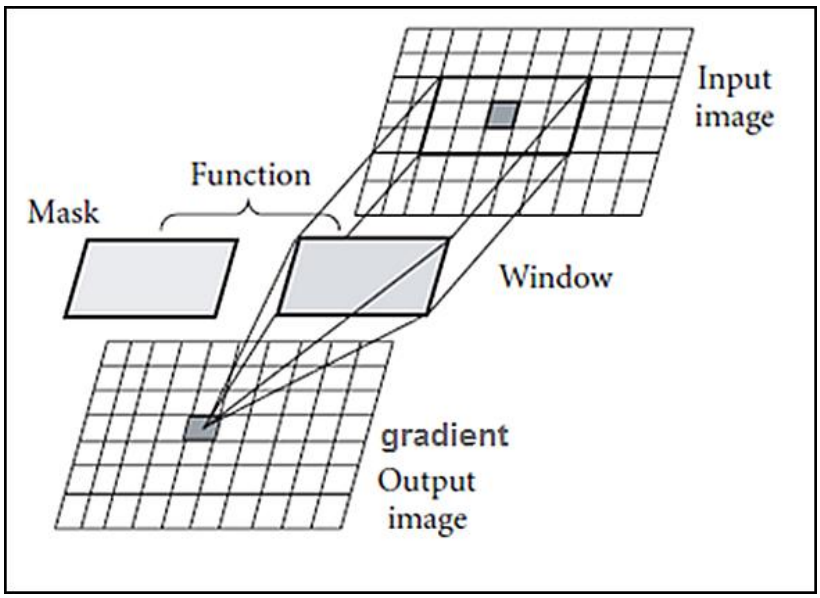


Figure 2.4 Obtaining image gradient [12]

It is important to mention that obtaining image derivative enhances noise. Therefore, it is important to take into account that noise suppression is to be introduced before applying the edge detection operation [7, 10]. Some of the widely used gradient operators are introduced in the following subsections.

a. Roberts Operator

Since the image edges can be obtained by the partial derivatives of the discrete values of the image pixels, the partial derivative of a certain pixel may be approximated over its neighborhood [7]. Previously, it has been shown in section 2.3.2 that the first derivative of the brightness can be substituted by the first difference in the horizontal and vertical directions [10]. The two gradients are defined using equations (2.8) and (2.9), respectively:

$$Gx = \frac{\partial f(x,y)}{\partial x} = f(x+1,y) - f(x,y) \tag{2.8}$$

$$Gy = \frac{\partial f(x,y)}{\partial y} = f(x,y+1) - f(x,y) \tag{2.9}$$

The equations can be applied over all *x* and *y* of the image *f* by filtering the image *f*(*x*,*y*) with a 2-D Roberts mask. Roberts's cross-gradient operator was invented to find edges across diagonal axes by [13]. The operators consist of two masks to be rolled over the image pixels in order to find the differences between each two diagonal ones. Consider the 3x3 sub-image shown in Fig 2.5 a). The two Roberts masks in Fig. 2.5 b) attempt to compute the diagonal differences as shown in equations (2.10) and (2.11):

Z1	Z2	Z3
Z4	Z5	Z6
Z7	Z8	Z9

a) 3x3 sub-image

-1	0
0	1

0	-1
1	0

b) Roberts gradient masks

Figure 2.5 Robert operator

$$Gx = \frac{\partial f(x,y)}{\partial x} = (Z9 - Z5) \tag{2.10}$$

$$Gy = \frac{\partial f(x,y)}{\partial y} = (Z8 - Z6) \tag{2.11}$$

The edge value after the convolution operation is computed as (*x-1/2* and *y-1/2*) because the edge corresponds to the central point. The 2x2 masks are simple but they are not efficient as they do not examine the nature of data on the opposite sides of a central point which increases the masks sensitivity to noise.

b. Prewitt Operator

Unlike Roberts operator, Prewitt operator [14-15] tests the edge strength at the central point of a 3x3 neighborhood, i.e., *Z5* in Fig.2.5 a). The test of the edge at a central point carries more information regarding the edge direction [7]. The Prewitt masks are shown in Fig. 2.6

-1	-1	-1	-1	0	1
0	0	0	-1	0	1
1	1	1	-1	0	1

Figure 2.6 Prewitt vertical and horizontal masks

The two masks are slid over the image pixels row by row to compute the horizontal and the vertical gradients for the central pixel in the neighborhood *Z5* as in equations (2.12) and (2.13), respectively. The masks give equivalent weights to all the pixels around the candidate pixel Z5. Edges produced by Prewitt operators are expected to be more accurate than those produced by Roberts [7].

$$Gx = \frac{\partial f(x,y)}{\partial x} = (Z7 + Z8 + Z9) - (Z1 + Z2 + Z3) \tag{2.12}$$

$$Gy = \frac{\partial f(x,y)}{\partial y} = (Z3 + Z6 + Z9) - (Z1 + Z4 + Z7) \tag{2.13}$$

The total gradient is calculated by equation (2.5) or (2.7) for all the image pixels to result in the gradient image. The gradient pixels are compared with the threshold value for binarization.

c. Sobel Operator

The Sobel operator is a widely used operator for image edge detection [16-17]. It improves the edge detection response of Prewitt by assigning higher weights to the pixels close to the candidate pixel. Utilizing double weights will smooth the image noise [10]. The Sobel gradients equations are shown in equation (2.14) and (2.15).

$$Gx = \frac{\partial f(x,y)}{\partial x} = (Z7 + 2Z8 + Z9) - (Z1 + 2Z2 + Z3) \tag{2.14}$$

$$Gy = \frac{\partial f(x,y)}{\partial y} = (Z3 + 2Z6 + Z9) - (Z1 + 2Z4 + Z7) \tag{2.15}$$

Sobel operator is based on the differencing operations [1 0 -1] and averaging operation [1 2 1] in order to smooth the noise developed by the derivatives. By convolving these two operations together, the two 3x3 Sobel masks can be obtained.

Like all gradient operators, Sobel computes the first derivative of a grayscale input image by convolving the 3x3 masks on the image (Fig. 2.7). Since the borders cannot be calculated with the 3x3 mask, they have to be blacked out, partially calculated or calculated with zero extensions as partial neighborhoods. Although Sobel operator has a smoothing effect, it is still considered sensitive to noise; it brings noise in the images as edges. Another disadvantage of the operator is that, edges found by Sobel operators are wider than usual due to the two consecutive intensity changes within three pixels [7]. The gradient image obtained using the Sobel masks is shown in Fig. 2.8. Fig. 2.9 illustrates the edge detected image after thresholding.

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

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

Figure 2.7 Vertical and Horizontal masks of Sobel

In this thesis, Sobel operator has been selected to implement a dedicated edge detection processor. Further information on this will be discussed in chapter 3 and 4.

Figure 2.8 Sobel Gradient image

Figure 2.9 Sobel edge detection after thresholding

d. Canny Operator:

Canny operator [18] combines the edge detection with noise reduction. Canny aims to achieve a superior edge detection results by decreasing the error rate, best localization of edges and minimizing the response to single edge points [7, 10]. Unfortunately, the operator requires complex calculations, so it is not suitable for dedicated hardware.

2.4 Parallelism in Image/Video Processing Operations

It has been mentioned in section 2.1 that image processing can be classified into two levels: low-level and high-level processing. The former includes the operations that convert one form of image data into another form of image data. Whereas, high–level processing interprets the pre-processed image contents to perform object recognition. Applications at both levels are computational and data intensive. The amount of computations for each pixel is determined by the depth of the image processing algorithm used. Since most of the algorithms are iterative in nature, meaning the exact computations are to be repeated for each pixel in the image frame, increasing the image frame size will increase the computational burden significantly. For example, the NTSC video standard requires 30 frames per second, with approximately 0.25

mega pixels per frame, resulting in total of 7.5 mega pixels per second [19]. PAL video standard requires 25 frames per second with a larger frame size. For algorithms like image edge detection using Canny operator, the algorithms requires approximately 130 operations per pixel. High resolution image and video standard have 4 times more pixels per frame which will increase the computational load even further [19].

An efficient way to deal with enormous amount of data is by the use of parallel processing. The key to efficient implementations of image and video processing systems are always achieved by exploiting data or instruction parallelism within the image processing algorithms. The data level parallelism (DLP) is exploited by applying the same operation to multiple sets of data while the instruction level of parallelism (ILP) is accomplished by scheduling the execution of multiple independent operations in a pipeline manner.

Low-level processing operations entail nested looping with the innermost loop applying an operator (neighborhood, point and global operators) to obtain the output pixels forming the output image. Mixed with the large amount of data that requires high memory bandwidth, the nature of the low-level operations increases the possibility of utilizing data level parallelism when implementing their application. High-level operations are control-intensive and are more sequential in nature than parallel. With irregular instructions structure and low data and bandwidth requirements, high-level operations are meant for instruction level parallelism [2].

When implementing real-time image/video processing systems on resource constrained platform, the massive amount of input data makes developing an algorithm that is computational, memory, power efficient a very difficult task. Since the algorithms are usually developed under "no resource limitations" environment, they have to be optimized in order to meet real-time performance.

2.5 Optimization of Image Processing Systems

While variety of hardware and software optimization techniques are used to realize real-time versions of image / video processing algorithms, simplifying the algorithm

can result in high performance gain. The simplification rationalizes the algorithm back to the core functionality, which reduces the computational complexity, the memory usage and power consumption [2]. By applying simplification strategies to the algorithm in the research development environment, the optimized algorithm is able to meet real-time constraints when it is implemented on limited resource platform such as FPGAs. Simplifying the algorithms of image /video processing reveals three basic schemes:

- Reducing the number of operations.
- Reducing the amount of processed data.
- Using alternative simpler algorithm.

2.5.1 Reduction of Computation

Since the low-level image and video processing operations are able to perform large amount of computations on a massive data amount, every operation counts. The reduction of the amount of operation to perform in the algorithm implies reducing the number of calculations to be performed on each pixel. This will increase the performance of the system to match the real-time constrains.

The reduction of operations can be classified into two categories; pure reduction and reduction via approximation. The pure operation reduction entails actions that reduce the number of operations without changing the final numerical results. If the final result is changed, the method then is categorized under the approximation or suboptimal solution category. Low-level operations that involve symmetrical computations or successive computational redundancies are perfect candidates for this reduction strategy. This strategy is usually implemented by revealing the mathematical identities and properties using manual expansion of the calculation steps.

Edge detection is a perfect example of low-level window operations with high computational redundancy and symmetry. The sliding window operations entailed in the gradient edge detection operations involve symmetrical coefficients which can be employed to reduce the computations by rearranging and regrouping operations in order to reduce the occurrence of time-consuming calculations. Chapter 3 shows how

the proposed Sobel architecture employs the symmetries within the operator masks to reduce the number of calculations and hence computation time.

2.5.2 Reduction of Data

Reduction of the amount of data to be processed is a major source of performance gain. Since the computations are performed on each pixel in the image frame, reducing the number of pixels is essential for achieving real-time performance and reducing the memory usage. Reduction in processed data can be applied in many ways; all aim to process a certain subset of the image pixels at each time. These include spatial-temporal down sampling which involves processing of certain pixel subsets and skipping the others or skipping certain frames in the video processing systems. It can also be achieved by tiling and partitioning of the image frame into nonoverlapping or overlapping blocks and applying the expensive operations to the interested blocks. This method is not applied in this thesis since edge detection is a very low-level operation that requires processing all the pixels in the image frame.

2.5.3 Alternative Simple Algorithm

The key in designing real-time resource-friendly systems is to base those systems on computationally simple algorithms. The simple algorithms may not be the perfect computational solution if they are considered from a mathematical point of view. However, they are convenient for embedded real-time systems. In the embedded world, a system's storage requirements are very important for the choice of a feasible solution which can indirectly affect the power consumption of the system. Therefore, an optimum real-time embedded solution should be computationally simple and memory and power efficient.

In general, to achieve optimal performance in image processing algorithms, designers attempt to match the hardware to the problem [5]. Various techniques are used to squeeze more performance from the machine. These techniques involve manual tuning of the compiler output in order to find the best time/space trade-off. Time-space techniques are directly applied to the imaging algorithm because imaging applications tend to be memory intensive [5]. These techniques comprise the following:

2.5.4 Basic Optimization Techniques

Most of the optimizations techniques used to optimize image processing systems for real-time attempt to find time-wasting computations and try to avoid them in order to reduce the overall execution time [5]. These techniques are utilized to optimize the designs of modern compilers to achieve the best time-space trade-off. This implies if the optimization attempts to minimize the execution time, the memory usage will increase accordingly and vice versa. That is due to the increment of the architecture parallelism which will accordingly increase the logic and memory consumption. In order to apply this concept on image processing applications, the optimization techniques need to be applied on the image processing algorithm to reduce the overall execution time [5]. Here are some of those techniques that are used to attain real-time performance in image processing systems.

2.5.4.1 *Arithmetic Identities*

Significant execution-time reduction can be achieved by common sense application of arithmetic identities, for example, the avoidance of multiplication by 1 and addition by 0 [5]. Moreover, a division operation executes slower than a multiplication operation in some computers. So it will be more efficient to replace it by multiplication with a negative exponent. This will save a considerable number of cycles. Chapter 3 will demonstrate how this technique is used to optimize the Sobel algorithm which involves a lot of multiplications by 1 and 0.

2.5.4.2 *Reduction in Strength*

Replacing arithmetic operations with equivalent but faster operations is referred to as reduction in strength [5]. Replacing a multiplication and division by the power of 2 with shifting operations is generally faster [2, 5]. Substituting multiplication and division operations with shifting will save not only several clock cycles but also considerable amount of logic resources required to implement a parallel multiplier or a divider by using a simple shift register. In this thesis, the proposed architecture is optimized to replace all multiplications by 2 with shift operations.

2.5.4.3 *Common Subexpression Elimination*

It is an optimization technique that searches for instances of identical expressions and replaces them with a single variable that carries the computed value. This necessitates examining whether or not the replacement is worthy, i.e. by examining whether saving the value of the variable is more expensive than recalculating it in terms of memory, logical resources and performance. Common subexpression elimination can be divided into two kinds:

- Local Subexpression elimination: A simple optimization that works within a simple basic block.
- Global Subexpression elimination: Implies analyzing the entire procedure by the use of the data flow analysis. For example, the following Verilog® code is considered:

```
assign Z= pi * x + omega * y

assign Q= omega * (pi + y)
```

The value (`y*omega`) represent a common subexpression and it might be assigned to an intermediate variable `T`. the code may be rewritten as follows:

```
assign T= y * omega

assign Z = pi * x + T

assign Q = pi + T
```

The value `T` will be saved in a register for further loop calculations. The effect of the common sub expression elimination is good enough to be used as a common compiler optimization.

2.5.4.4 *Loop Invariant Removal*

A common loop optimization technique is performed by the exclusion of code that does not change inside an iterative loop. The removal of the loop invariants can save the time required to recalculate the invariants inside the loop. The following Verilog®

code is an example of removal of a loop invariant:

```
for ( i= 1; i < 100; i = i+1) begin
    X[i]= X[i] +2 *omega *t;
end
```

By eliminating the invariant (`2 * omega * t`), the code can be rewritten as:

```
Z = 2 * omega * t
for ( i= 1; i < 100; i +1) begin
X [i] = X [i] + Z
end
```

2.5.4.5 *Loop Transformations*

Loop transformations attempt to improve performance by rewriting loops to make better use of memory system [20-23]. Optimization of memory usage relies on the fact that the same data will be used repeatedly. The reused data are kept in a local or embedded memory which has low access latency. The reuse of the data in the embedded memory improves the performance by reducing accesses to main memory and hence reduces the memory access latency. Two types of data reuse are explained here [24]:

- Spatial reused: When several data items within the same embedded memory line are reused in the loop, the loop exhibits spatial reuse.
- Temporal reuse: If the same data is used in more than one loop iteration then the loop demonstrates temporal reuse.

The loop transformations are usually applied by changing the order of the execution of those nested loops. They aim to uncover the massive amount of fine-grain parallelism and to endorse the data reuse by the improvement of both the spatial and temporal data reuse. The interaction between loop transformation methods

enables a wide variety of designs with different area and time trade-offs [22]. The loop transformations include the following types:

i. Loop Unrolling

The process of expanding the loop to remove the loop overhead is called loop unrolling and it can save considerable execution time [5, 25-28]. Loop unrolling is often used to change a linear algorithm (e.g., edge detection) into a parallel architecture supporting format in order to achieve a higher throughput. Unrolling the loop implies the replication of its body corresponding to consecutive iterations which enables high-level of parallelism and reduces the number of memory accesses by reusing data of the unrolled loop (if kept in registers) [21]. Generally speaking, if an algorithm consisting of n nested loops is "unrolled", the parallel implementation of the system will inherit a performance gain – system throughput - by a factor of n [29]. As an example for loop unrolling, consider the next iterative Verilog® code:

```
for (i=0; i<3; i++) begin
   for (j=0; j<3; j++)
      begin
         X[i]= X[i]+ Y[i][j] *Z[i]
       end
    end
```

The unrolled version of the code is shown as follows:

```
for (i=0; i<3; i++) begin
       X[i] = X[i] + Y[i][j]   *Z[i]
       X[i] = X[i] + Y[i][j+1] *Z[i]
       X[i] = X[i] + Y[i][j+2] *Z[i]
end
```

Using loop unrolling, the high throughput is accompanied by an increment of the area and power; due to high memory and logic utilization [30]. Partial loop unrolling is considered in case of large loops to reduce the logic utilization and power consumption [30].

ii. Loop Blocking (Tiling)

Loop blocking (also called strip mining or loop tiling) which replaces a single loop with two loops [20, 22, 31]. The inner loop works on a portion of the original loop iteration space with the same increment as the original loop. The outer loop navigates across the original iteration space with an increment equal to the block size. The inner loop will be fed by an outer loop with blocks of the original iteration space to execute the original loop block by block. When applying loop blocking on a nested loop with a depth n, the new nested loop depth will be anything between n+1 and 2n. Loop blocking divides the large loop iteration space into smaller blocks which allows the accessed data to fit in the size-limited embedded or local memory. Moreover, loop blocking is used to minimize the external memory accesses by enhancing the reuse of the data in the embedded memory. It permits the processor to perform as many operations as possible on the array elements in the embedded memory and reduces the references to external memory [24, 32]. For instance, consider the following VHDL® code:

```
//loop before Blocking
  for (i=0; i< max; i++) {
          for (j=0; j< max; j++){
              a[i ] [ j] += a [i , j]+ b[j , i];
          }
  }
```

The code consists of two nested loop `i` and `j`. To apply loop blocking, the each of the two loops is replaced with two loops, an outer loop and an inner loop. The outer loops `i` and `j` have the same size of the original `i` and `j` loops (before tiling), respectively. Although, both the new loops `i` and `j` have a step size equal to the required block size. The inner loops `ii` and `jj` have sizes equal to the block size and a step size equal to the original `i` and `j` loops before the loop blocking, respectively.

```
// the loop after loop blocking
for (i=0; i< max; i+= block_size)  {
   for (j=0; j< max; j+= block_size){
      for (ii= i; ii<i+block_size; ii++){
        for (jj=j; jj<j+block_size; jj++){
           a[ii][jj] += a [ii][jj]+ b[jj][ii];
        }
      }
   }
}
```

iii. Loop Fusion:

Loop fusion combines instructions from several adjacent loops that use the same memory locations into a single loop [32]. This enhances the data reuse by preventing the data from being reloaded and hence reduces total processing time. The next Verilog® codes show the effect of loop fusion on two loops:

```
// the original code
for (i=1; i< n; i++)
  begin
    b[i] = a[i]+2*c;
   end
 for (i=1; i< n; i++)
   begin
     d[i]= b[i]+2*c;
   end
```

```
//after loop fusion
for (i=1; i< n; i++)
  begin
    b[i] = a[i]+2*c;
    d[i] = b[i]+2*c;
   end
```

2.5.4.6 *Scalar replacement*

Scalar replacement or register promotion is an optimization technique that aims to reduce accesses to external memory by replacing repeated array references with scalar temporaries [20]. The replacement is achieved by storing the repeatedly referred array data in registers. In FPGAs, scalar replacement can be implemented using internal registers or embedded RAM blocks. In nested loop computations (e.g., the image processing operations) multiple references are made to the same data in the same or consecutive loop iterations.

Scalar replacement for reconfigurable architectures differs from cache based architectures in the data replacement and localization strategies. The data replacement and localization are made explicitly, i.e., every datum has to be stored in a specific location in order to be reused by the designer, or discarded when it is no longer in use [22, 33].

Chapter 4 discusses the application if these optimization techniques on Sobel edge detection algorithm to create an optimized Sobel processor.

2.6 Sobel Processors on FPGA

General purpose processors (GPP) do not take advantage of the parallelism within the computations in image processing algorithm, neither can handle the I/O requirements

for these systems which make them inefficient for implementing real-time image processing systems [34]. It was shown in section 1.3 how FPGAs satisfy the performance requirements and real-time constrains for image processor architectures design. In this section, several parallel architectures for Sobel edge detection processor are reviewed.

Generally, implementing a Sobel edge detection system on FPGA is the same as a 2-Dimentional (2-D) image filter where line buffers are usually utilized as a delay line to feed the input pixels to the filter. Each line buffer is as wide as a whole frame line and feeds the input pixels simultaneously with the other line buffers to the filter nodes as shown in Fig. 2.10. The line buffers allow each input pixel to be loaded once to the FPGA from the external memory. This reduces the memory read latency and enables the filter to achieve real-time performance constraints [35]. Each pixel is multiplied by the filter coefficient (the Sobel masks in our case) and the multiplication results are added together in an adder tree. The output of the adder tree is the output pixel of the filter [19, 35]. Since this design depends mainly on multiplications, the number of multipliers determines design complexity. For filters with non-symmetric coefficients, the size of the mask m*m generates a design with m^2 complexity. For 2-D filters with symmetric masks, the complexity can be significantly optimized to [(m+1) + (m+3)]/8 and the reduction can be increased up to (m+1) for separable filters, i.e., 1-D filter in the horizontal direction and another 1-D filter in the vertical direction (such as Sobel filter). Separable designs enable parallel processing for the horizontal and vertical oriented gradients on FPGA [19]. Figure 2.10 and 2.11 demonstrate the non-symmetric and symmetric implementation of the 2-D filters, respectively. Since Sobel masks are symmetric to each other, the next chapter shows how the proposed architecture is optimized to take advantage of the symmetries within the masks.

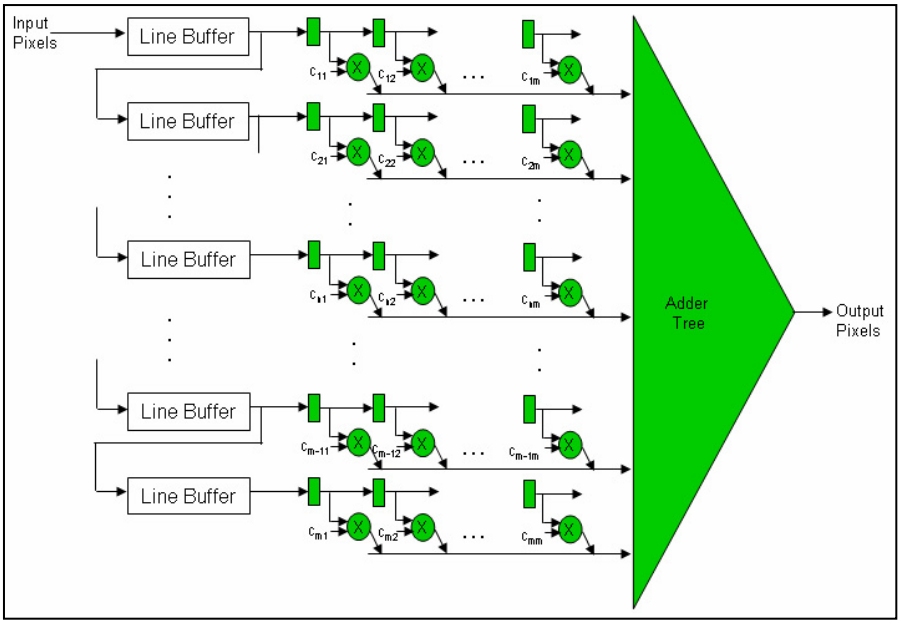


Figure 2.10 Hardware Implementation of non-symmetric 2-D filter [19]

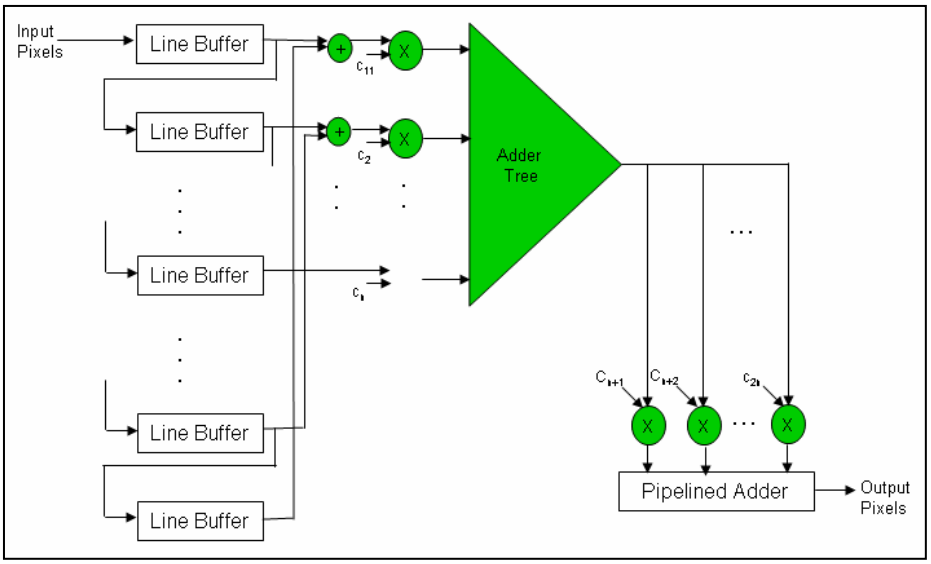


Figure 2.11 Hardware implementation of symmetric separable 2-D filter [19]

The following subsections describe several parallel architectures for implementing Sobel edge detection system on FPGAs.

2.6.1 Sobel Edge Detection on FPGA Using Pipelined and Parallel Architectures

Vinu Thomas and his students designed a SOBEL edge detection processor and implemented it on Xilinx Spartan 3 XC3S400 FPGA [36]. The row image data are converted into grayscale and sent to the FPGA board via bidirectional parallel port. The pixel information extracted is then processed with the Sobel processor and calculation results are sent back to the host computer. After that the data are manipulated to reconstruct the edge detected image. The processor is designed using Verilog® HDL. Open source electronic design automation tools, GPLCver and GTKWave are employed for compilation and post processing of the simulation results consecutively.

The edge detection system in [36] is shown in Fig. 2.14. The system has 8-bit bidirectional bus to transfer and receive image data. The *bus_rw* signal is used to choose either read or write processes. To control the data transfer operations, *data_strobe* and *mode_strobe* signals are used. The *clk* signal is connected to the FPGA internal clock.

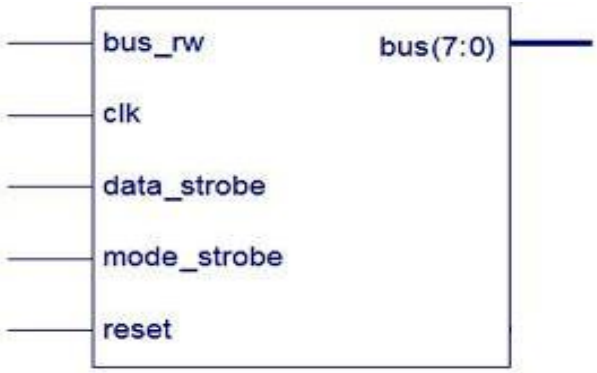


Figure 2.12 The edge detection system implemented on Xilinx Spartan [36]

0	0	0	0	.....	0	0	0
0	0	0	0	.....	0	0	0
A_0	A_1	A_2	A_3	.....	A_{32}	A_{33}	A_{34}

0	0	0	0	.....	0	0	0
A_0	A_1	A_2	A_3	.....	A_{32}	A_{33}	A_{34}
B_0	B_1	B_2	B_3	.....	B_{32}	B_{33}	B_{34}

A_0	A_1	A_2	A_3	.....	A_{32}	A_{33}	A_{34}
B_0	B_1	B_2	B_3	.....	B_{32}	B_{33}	B_{34}
C_0	C_1	C_2	C_3	.....	C_{32}	C_{33}	C_{34}

Figure 2.13 RAM modules [36]

The system consists of 32 Sobel instances connected to 3 sets of RAM array, each of which is 34 byte. The RAM arrays can be loaded serially and shifted in parallel; they are used to store the image data transferred from the host computer.

As illustrated in the Fig 2.13, the third RAM module is serially loaded first. Its contents are then shifted in parallel to the second RAM module and so on. When all three modules are fully loaded, the contents of the RAM modules are the inputs of the 32 Sobel instances.

It is mentioned before that the 32 Sobel instances are connected to the three RAM modules. Each of the instances consists of six signed subtractors, two shift registers, two modulus operators, three adders and a magnitude comparator. Fig. 2.16 demonstrates the block diagram of the Sobel instance. Eight image pixels are read from the RAM modules as an input for the Sobel instance. The instance calculates the horizontal and the vertical gradients according to the equations (2.16) and (2.17). Finally the total gradient G is compared with a predefined threshold value. The final answer is either 0 if the gradient is less than the threshold or 1 otherwise.

$$Gx = (Z3 - Z1) + 2(Z6 - Z4) + (Z9 - Z7) \tag{2.16}$$

$$Gy = (Z7 - Z1) + 2(Z8 - Z2) + (Z9 - Z3) \tag{2.17}$$

The design utilizes 80% of the total number of slices in the Xilinx Spartan 3 FPGA, 11% of slice flip flops, 69% of the 4 input Look-Up Tables (LUTs) and 22% of the bonded input/output blocks (IOBs). The disadvantage of this design is that the total logic utilization of the design is very high due to the large amount of registers used for storing the 34 bytes of pixels to produce 32 output pixels. Notice that the data in the later two RAM modules(B_1- B_{34} and C_1- C_{34}) are loaded again for the calculation of the next image line which increases the memory read redundancy, increase the overhead of reloading the data in the RAM modules before stating the next calculation and hence skew the design speed. Moreover, the partial results of the gradient are computed again for each pixel instead of being reused. The system produces 32 output pixels at the same time and those pixels are multiplexed to be written back to the external memory which prevents the processor from achieving real-time performance constrains. Our project uses this design as a reference and optimizes the architecture for less logic utilization and more performance gain. More information on this will be discussed in the next chapters.

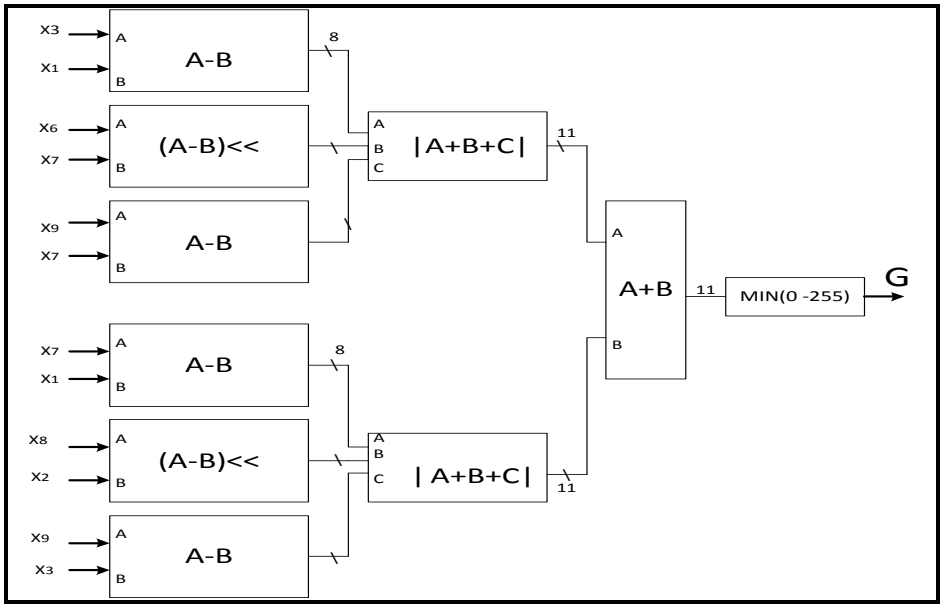


Figure 2.14 The Sobel instance

A Sobel processor architecture was designed by T. A. Abbasi and M. U.Abbasi in [37] and implemented on Xilinx Spartan3 FPGA. Their processor architecture is shown in Fig. 2.15. The inputs of the system are 8 pixels, each of which is 8-bit wide fed by an 8 bytes bus and 8-bit threshold value to be compared with the total output gradient. The output is an 8-bit pixel. The value of the output pixel depends on the value of the input threshold (either 0 or 255) which determines the accuracy of the edge detection output image.

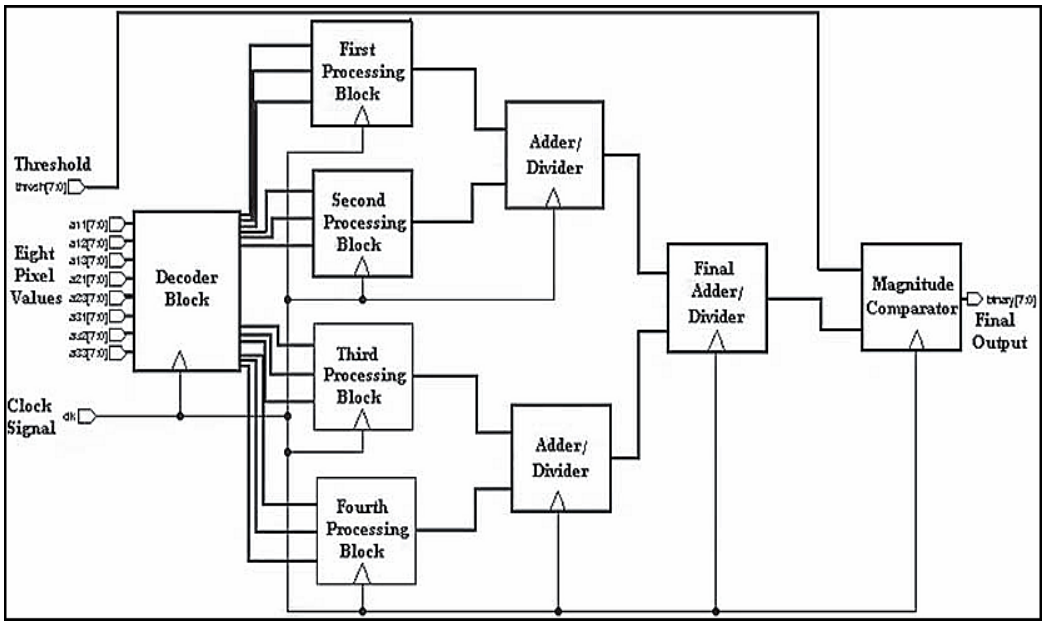


Figure 2.15 The Sobel architecture proposed by [37]

As shown in Fig.2.15, the processor consists of different blocks. The blocks functions are explained here:

1. Decoder block: to decode the eight input pixels into twelve pixels to be fed to four processing blocks, three pixels to each processing block.
2. First processing block: applies the weight [-1 -2 -1] on its three input pixels, add weighted pixel values and down scale the resultant by a factor of four. Performing these operations will take three CLK cycles.
3. Second processing block: applies the same function of the first block but with the weight of [1 2 1].
4. Third processing block: functions similarly as the first block.
5. Fourth processing block: is similar to the second block.

6. Adder divider blocks: add the two output pixels of the processing blocks and divide the result by a value of two. These blocks require three clock cycles to produce their output.
7. Final adder divider block: adds the two 8-bit output pixels of the two adder/divider blocks and divide the resultant by a factor of two. That takes two clock signals latency.

8. Comparator block: compares the resultant gradient value with the input threshold value. This results in the edge detected binary image having only two pixel values 0 and 225.

The architecture is synthesized for Xilinx Spartan 3 using Verilog® HDL. The processor works on 134.756 MHz and produces the edge detected binary image of a 512x512 8-bit grayscale image in 1.95ms. The architecture uses 26% of the total number of slices in Xilinx Spartan 3 XC3S50 -5PQ208, 18% slice flip flops, 13% of four input LUTs and 65% of the bonded IOBs. The design utilizes a lot of logic resources for only one Sobel instance, because it uses several dividers and multipliers which will limit the possibility of increasing design parallelism and hence limit design speed. It is noticed that the design does not employ any sort of buffer or embedded RAM blocks to reuse the input fetched data which increases the external memory redundancy and slow the design performance.

Another pipelined implementation of Sobel operator on FPGA is introduced by [38] as a part of template matching system using distance transforms. The Sobel processor is included in the preprocessing (pp) part of the system that consists of edge detection, noise removal, and distance transform computation. The Sobel processor uses Block RAM of the FPGA to store two lines of the input image. The 3x3 window of the processed data are accessed in parallel through 3x3 shift register arrays (SRA). The data is shifted to the left (SL) to replace the multiplication by 2 in the Sobel confidents. The horizontal and vertical gradients are calculated by pipelined horizontal and vertical arithmetic units (AU), respectively. The sum of both gradients S_Y and S_X is compared with the threshold to obtain image edges. The architecture is shown in Fig.2.16.

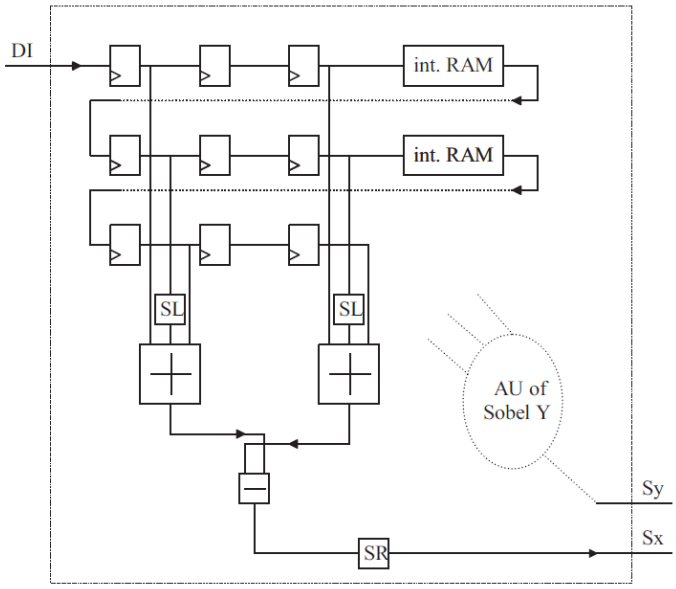


Figure 2.16 Hardware implementation of Sobel operator by [38]

The architecture is implemented on Xilinx XC2V3000 and requires 165 Slices and 1 Block RAM while the whole pp system consumes 1712 slices and 4 block RAMs. The pp system is able to process 512x512 image in 5.5ms using a 96MHz clock frequency.

Park and Diniz in [22] implemented an optimized Sobel processor that utilizes scalar replacement of the input pixels. Virtex4 of Xilinx FPGA is utilized to implement the processor design which is shown in Fig. 2.17. The design consists of data path and a memory interface. The data path contains the combinational logic of Sobel algorithm and a finite state machine (FSM) that handle the data accesses control signals. The memory interface handles the data transfers protocols and address generation. The design is pipelined in both the data interface and data path components. In [22], five architectures are discussed. In this thesis, two of those architectures are compared with the optimized Sobel processor. The first Sobel processor utilizes data reuse within the horizontal gradient loop only with the use of scalar replacement. The second processor utilizes data reuse within vertical and horizontal gradients using the scalar replacement. It is shown in section 4.3.2 how the optimized Sobel processor exceeds the performance of both processors proposed by [22] with the use of partial loop unrolling and scalar replacement in only the horizontal gradient calculation.

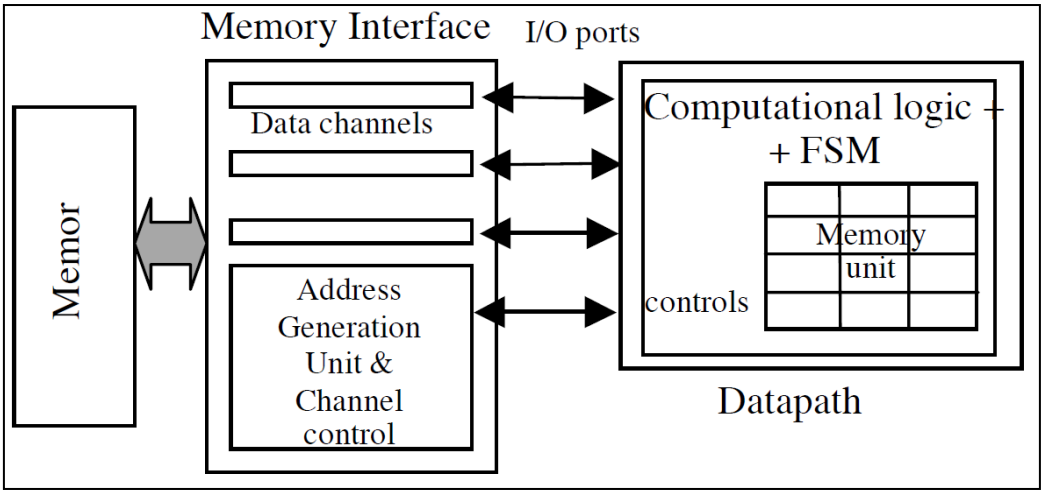


Figure 2.17 Sobel Architecture by [22]

2.6.2 Sobel Edge Detection System on FPGA Using Distributed Arithmetic

Distributed arithmetic (DA) is a computational algorithm used to implement weighted sum of products operation performed by digital filters [20]. Since Sobel operator is a basic two dimensional Finite Impulse Response (FIR) digital filter, DA represents a feasible way of implementing Sobel on FPGA.

Fig. 2.18 shows how Sobel edge detection filter is implemented using the DA algorithm. The architecture consists of a set of parallel to serial registers (PSR) and shift registers (SR), a pair of look up tables (LUT) addressed by the serial output of the shift registers, and an adder /subtractor accumulator register. The main idea is that, multiplying the pixel value (signal) with a coefficient (constant) produces the products that have to be accumulated to get the pixel gradient. In DA algorithm, instead of calculating each product separately, all possible cumulative partial products of all terms are saved in the LUT and addressed by the multiplier bits. When the most significant sign bits address the LUT, this causes the outcome to be subtracted from the accumulated partial products.

The design works at 14.36 Mega pixels per second for 800x600 images. Thirty seven configurable logic blocks (CLBs) are utilized to implement the design on Xilinx XC3042 FPGA using VHDL. Equations (2.14) and (2.15) of Sobel can be rewritten to benefit from the coefficient symmetries which reduces the sum of product terms. The pixel variables with the same coefficient values are grouped as sum of difference

terms. This reduces the number of calculations required for calculating the gradient and save silicon area. The new form is shown in equations (2.16) and (2.17). The processor is unable to satisfy the real-time performance constraints with its current configuration. To meet real-time constraints, the design has to be duplicated five times to compute five pixels simultaneously. Moreover the clock frequency must be increased to 60MHz.

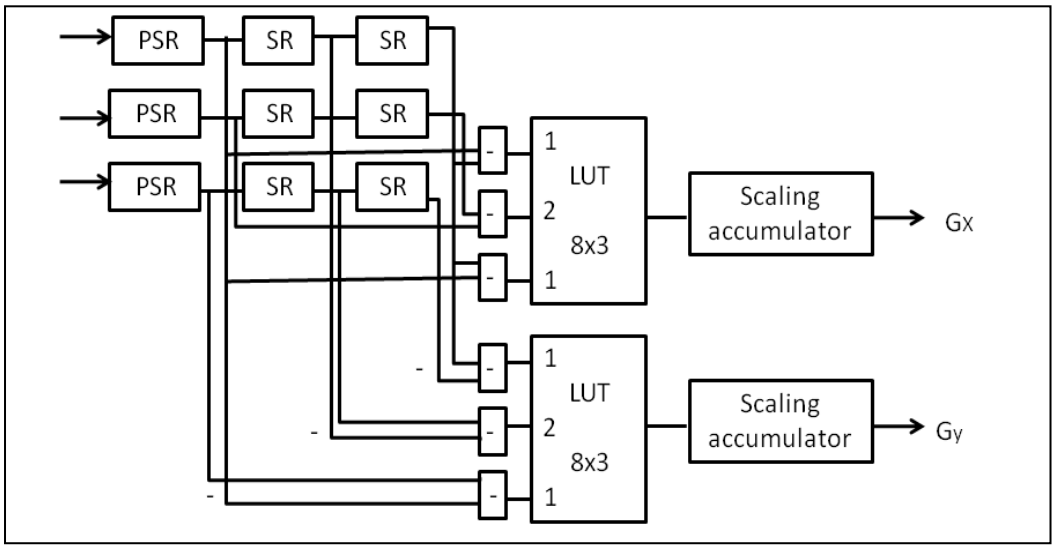


Figure 2.18 Edge detection filter is implemented using the distributed arithmetic algorithm

Another FPGA based implementation of Sobel processor using DA is presented by Chivapresscha and Dejhan in [39]. The processor was implemented using VHDL and synthesized on Altera FLEX10K FPGA. The design uses a frame management system which works as a delay line to shift the pixels that are buffered to the processor module. The design utilizes the two LPM-ROM (library of parameterized modules- read only memory) in the FLEX10K to store the horizontal and vertical masks coefficients. Each ROM module has 9 inputs. The processor has 8-bit input pixel produces the vertical and the horizontal gradients as two 8-bit outputs data. The design requires 280 logic cells and 12288 memory bits to be implemented on the FLEX10K FPGA. The processor can operates by a 37MHz clock and can meet real-time image processing applications requirements.

2.6.3 Sobel Edge Detection System on FPGA Using Systolic Arrays

Systolic arrays (systolic algorithm) is one of the widely used algorithm for implementing convolution and digital filters [40-41]. FPGA based systolic array systems are well-matched for real-time window based operations, e.g. edge detection [12]. The systolic array consists of an array or processing elements (PEs). The PEs array is fed by a memory that contains the image data. The system is implemented on Splash 2 which is the processor attached to SPARCstation. The Splash-2 board consists of 17 PEs based on Xilinx 4010 FPGA named X_0 - X_{16}. The processing element X_0 is used to control the data flow in the processor and the cross bar connections between the PEs. Each PE is connected to a 512KB memory that is utilized to store the intermediate results and lookup tables. To implement Sobel on the Splash 2 board, the two-dimensional (2-D) convolution of the 3x3 mask of Sobel is converted to linear or 1-D convolution. The 1-D convolution implies that a mask of k coefficients is represented by k PEs. Each PE multiplies the pixel that is transferred from the memory or from an adjacent PE with the coefficient, adds the partial sum to it, and passes it to the next PE. The 2-D implies that the k x k mask is represented by k x N assigning 0's to the coefficients of the extra (N - k) PEs. To feed the data to the processor (N - k) stages of shift registers are used. Fig 2.19 shows a 2-D convolution with shift registers and memory lookup tables and the Sobel processor on Splash 2 is shown in Fig. 2.20.

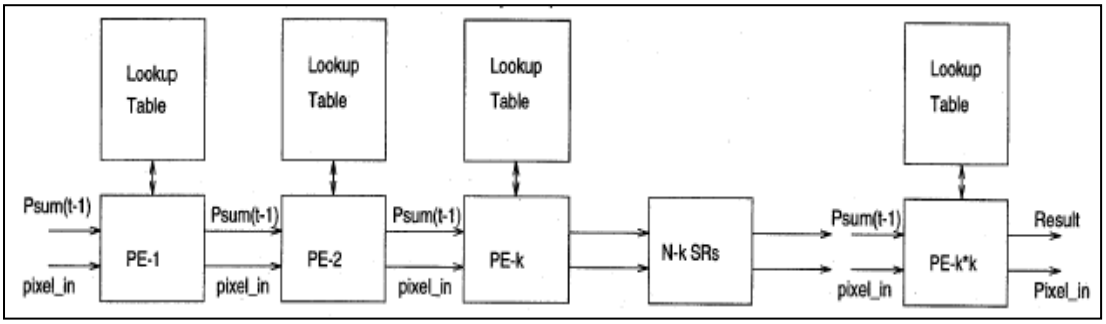


Figure 2.19 Systolic array for 2-D convolution with shift registers and memory lookup tables [12]

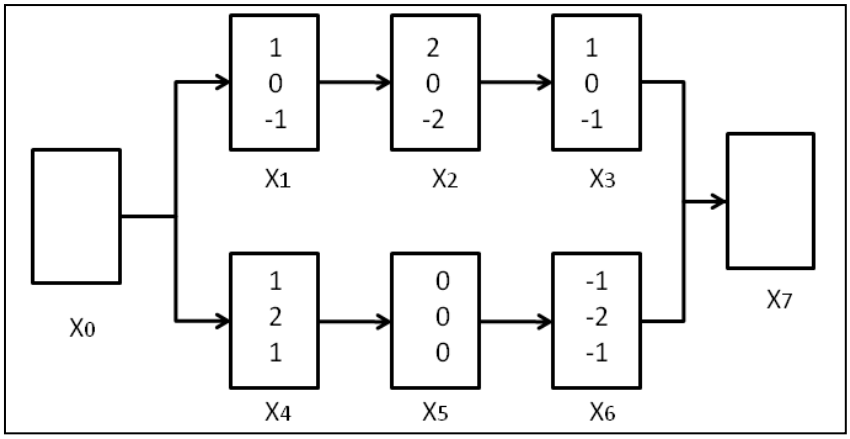


Figure 2.20 Sobel filter on Splash 2 using Systolic array

The processor is 32 pixels in width to reduce the logic utilization when calculating larger images. Any disarrangement of the *k* x *N* mask elements produces results that are not parts of the desired output. Moreover, the processor size increases unnecessarily for larger image widths. Therefore, the image has to be divided into smaller images of a size equals to the appropriate processor width. Although the processor elements are multiplier less and the multiplication values are stored in the lookup tables, accessing lookup tables takes skews the execution time for one clock cycle to get the multiplication result. The design has a frame rate up to 100fps which is 11 times less than that of the processor proposed in this thesis.

2.7 Chapter Summary

This chapter represents image edge detection as an important image segmentation technique. Some of the most famous edge detection operators have been discussed. Furthermore, several optimization techniques used to optimize the image processing system are reviewed in order to achieve embedded real-time systems requirements. Several architectures for Sobel edge detection processor are also discussed.

CHAPTER 3

THE PROPOSED ARCHITECTURE

3.1 Introduction

In the previous chapter, several FPGA-based Sobel edge detection systems are reviewed. It is also mentioned that FPGAs are a better platform for this kind of applications due to their high computational power, flexibility, parallelism and input/output capability. However, the logic and memory resources of the FPGAs are limited which make the implementation of image processing systems on FPGAs more challenging [2, 5]. Since the image processing systems are well known for their large amount of computations and data, designers should be aware of limitations while meeting the system requirements. This can be achieved by matching the hardware to the algorithms. This requires optimization of the algorithms to minimize the logic and memory space utilization required to perform the calculations and store the data, respectively. At the same time, designers should maintain a performance level that meets the time requirements for real-time system.

In this chapter, a processor architecture for edge detection based on Sobel operator is presented. The processor is optimized to solve the previously mentioned problems and attain a real-time performance on FPGA platform. The processor combines the optimization techniques mentioned in chapter 2 to minimize the logic and memory utilization, reduce the external memory read redundancy, avoid the recalculation of the same data and hence minimize the overall execution time.

The chapter starts by implementing a basic Sobel processor design on FPGA, analyzing the system to find the factors that degrade the performance of the design.

The design is then optimized by eliminating these performance degradation factors using several optimization techniques discussed in section 2.4. This results in an optimized Sobel edge detection processor. After that, the optimized processor is integrated in a complete edge detection system and implemented on FPGA to be tested for real-time performance. The implementation of the processor as a complete edge detection system on FPGA is detailed later in this chapter. The flowchart in Fig. 3.1 details the steps of this research methodology. The flowchart in Fig. 3.2 explicates the optimization process based on the data reuse enhancement. The exploit of data reuse in the optimization process is divided into two parts; the data reuse within one image line and the data reuse within multiple image lines.

The data reuse within the same image line is achieved by using the optimized Sobel processor that consists of a single row of Sobel instances. Each Sobel instance calculates one output pixel. Therefore, the number of Sobel instances determines the number of pixels the processor can achieve in a certain amount of time, i.e., the processor throughput. The single row processor avoids unnecessary recalculations by reusing the calculation within the processor block.

Since the optimization of the processor depends mostly on increasing the processor parallelism, the processor needs to read several input pixels in parallel in order to produce multiple pixels in on clock cycle. The use of the optimized Sobel processor calls for the design of a line buffer specially suited for the optimized Sobel processor. The processor line buffer is designed and integrated with the processor to handle the data coming from the external memory (SRAM) in a way that matches the execution requirements of a single-row-optimized Sobel processor.

The optimized edge detection system is implemented on the Altera Cyclone DE2-70 board and is able to achieve real-time system performance by processing a video stream from a D5M camera and the edge detected output stream is then displayed on a computer monitor. The optimized Sobel processor architecture performance is examined and compared with other designs [22, 37, 42].

The second part of the processor optimization flowchart shows the implementation of the data reuse within multiple image lines. It results in an optimized Sobel processor with multiple rows of Sobel instances and is able to calculate output data from different image lines at the same time. The processor is synthesized on Altera Cyclone II DE2-70 FPGA in order to see the effect of the optimization on the logic utilization percentage of the processor. The logic utilization of the processor is compared with the basic Sobel design. The design of the processor line buffer and the implementation of the full system on FPGA are not included in this thesis as highlighted in Fig. 3.2 and are left for future completion of this research work.

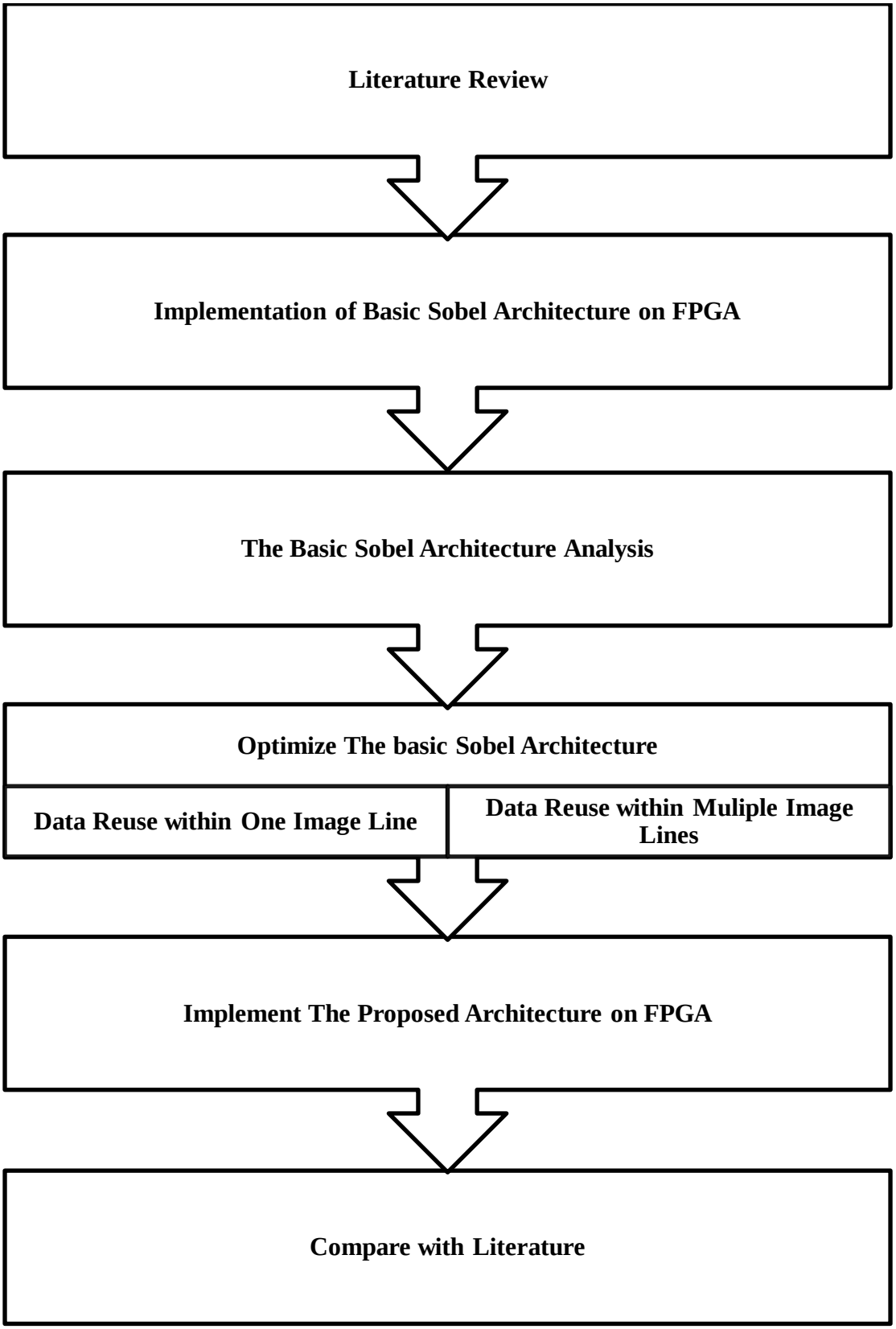


Figure 3.1 Research Methodology

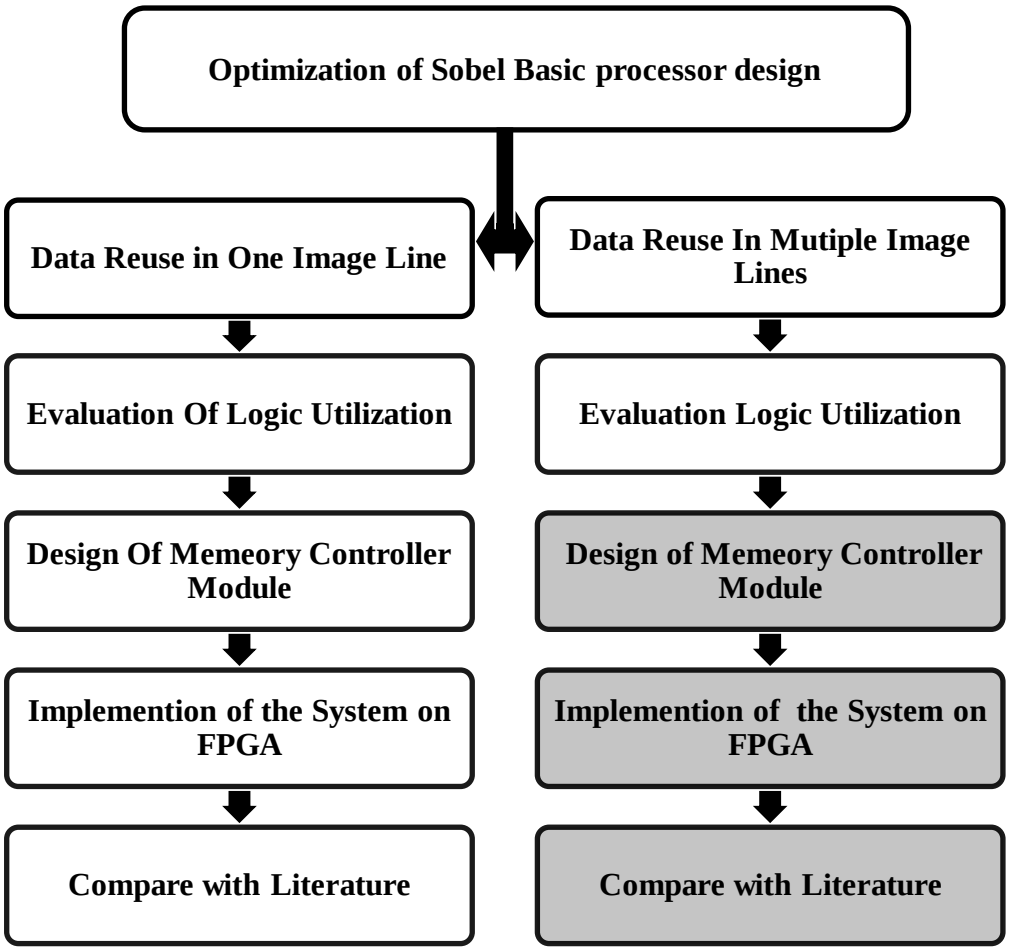


Figure 3.2 Flowchart of the Optimization Process

3.2 Basic Sobel Edge Detection Processor architecture

The basic Sobel architecture is inspired by the architecture mentioned in [36] which is explained in section 2.6.1. It has been discussed that the architecture attempts to increase the system performance by utilizing the FPGA parallelism and using the embedded memory to store the input data. The processor consists of 32 Sobel modules that are able to calculate the 32 output pixels in parallel wired to 3x34 bytes of FPGA embedded RAM that stores the input data. In section 3.3, an analysis of the system functionality is clarified to show the degrading factors of the system. Based on those factors, the optimization of the system will aim to reduce the effect of these factors and improve the system performance.

3.3 Basic Sobel Edge Detection Processor Analysis

In this section, an analysis of the basic Sobel processor design is performed. The analysis shows high redundancy in the calculations. The architecture calculates the same data repeatedly due to the high data dependency of the iterative process of the edge detection.

For simplicity, a small image of 5x8 grayscale pixels is assumed to be the input of the basic processor as shown in Fig. 3.3. To calculate the Sobel gradient for this image using the basic Sobel processor architecture, a row of 6 Sobel instances is required to calculate the gradient of a single image line in parallel. In Fig. 3.3, the X_{nm} values are the input image pixels in grayscale and the squares represent the input data of each Sobel instance in the processor row, where *n* and *m* represent the row and column order in the input image matrix, respectively. Note that in reality the image is always padded with zero frames to enable performing window operations on the first and last columns and rows and so maintain the dimension of the input image [43]. Since Sobel gradients G_x and G_y (refer to section 2.6.1) are calculated in parallel by one Sobel instance, the number of squares in Fig. 3.3 is twice the number of window operations to be performed in one row. The calculations of the 6 output pixels from Row 1, Row 2, and Row 3 are performed as in Table 3.1. The output image is a 3x6 image of gradient pixels.

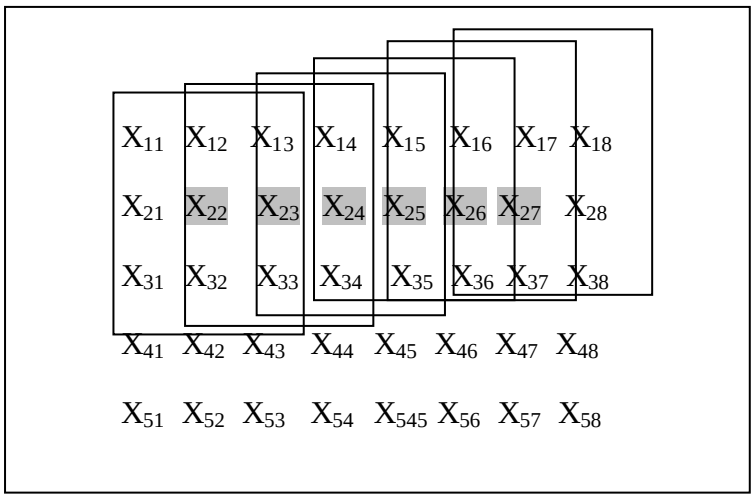


Figure 3.3 The input image

Table 3.1 Window operations in Image line (Row) 1, 2 and 3 using the basic Sobel processor

Output	Row 1	Row 2	Row 3
G_1	$Gx = (X_{13} - X_{11}) + 2(X_{23} - X_{21}) + (X_{33} - X_{31})$ $Gy = (X_{31} - X_{11}) + 2(X_{32} - X_{12}) + (X_{33} - X_{13})$	$Gx = (X_{23} - X_{21}) + 2(X_{33} - X_{31}) + (X_{43} - X_{41})$ $Gy = (X_{41} - X_{21}) + 2(X_{42} - X_{22}) + (X_{43} - X_{23})$	$Gx = (X_{33} - X_{31}) + 2(X_{43} - X_{41}) + (X_{53} - X_{51})$ $Gy = (X_{51} - X_{31}) + 2(X_{52} - X_{32}) + (X_{53} - X_{33})$
G_2	$Gx = (X_{14} - X_{12}) + 2(X_{24} - X_{22}) + (X_{34} - X_{32})$ $Gy = (X_{32} - X_{12}) + 2(X_{33} - X_{13}) + (X_{34} - X_{14})$	$Gx = (X_{24} - X_{22}) + 2(X_{34} - X_{32}) + (X_{44} - X_{42})$ $Gy = (X_{42} - X_{22}) + 2(X_{43} - X_{23}) + (X_{44} - X_{24})$	$Gx = (X_{34} - X_{32}) + 2(X_{44} - X_{42}) + (X_{54} - X_{52})$ $Gy = (X_{52} - X_{32}) + 2(X_{53} - X_{33}) + (X_{54} - X_{34})$
G_3	$Gx = (X_{15} - X_{13}) + 2(X_{25} - X_{23}) + (X_{35} - X_{33})$ $Gy = (X_{33} - X_{13}) + 2(X_{34} - X_{14}) + (X_{35} - X_{13})$	$Gx = (X_{25} - X_{23}) + 2(X_{35} - X_{33}) + (X_{45} - X_{43})$ $Gy = (X_{43} - X_{23}) + 2(X_{44} - X_{24}) + (X_{45} - X_{25})$	$Gx = (X_{35} - X_{33}) + 2(X_{45} - X_{43}) + (X_{55} - X_{53})$ $Gy = (X_{53} - X_{33}) + 2(X_{54} - X_{34}) + (X_{55} - X_{35})$
G_4	$Gx = (X_{16} - X_{14}) + 2(X_{26} - X_{24}) + (X_{36} - X_{34})$ $Gy = (X_{34} - X_{14}) + 2(X_{35} - X_{15}) + (X_{36} - X_{16})$	$Gx = (X_{26} - X_{24}) + 2(X_{36} - X_{34}) + (X_{46} - X_{44})$ $Gy = (X_{44} - X_{24}) + 2(X_{45} - X_{25}) + (X_{46} - X_{26})$	$Gx = (X_{36} - X_{34}) + 2(X_{46} - X_{44}) + (X_{56} - X_{54})$ $Gy = (X_{54} - X_{34}) + 2(X_{55} - X_{35}) + (X_{56} - X_{36})$
G_5	$Gx = (X_{17} - X_{15}) + 2(X_{27} - X_{25}) + (X_{37} - X_{35})$ $Gy = (X_{35} - X_{15}) + 2(X_{36} - X_{16}) + (X_{37} - X_{17})$	$Gx = (X_{27} - X_{25}) + 2(X_{37} - X_{35}) + (X_{47} - X_{45})$ $Gy = (X_{45} - X_{25}) + 2(X_{46} - X_{26}) + (X_{47} - X_{27})$	$Gx = (X_{37} - X_{35}) + 2(X_{47} - X_{45}) + (X_{57} - X_{55})$ $Gy = (X_{55} - X_{35}) + 2(X_{56} - X_{36}) + (X_{57} - X_{37})$
G_6	$Gx = (X_{18} - X_{16}) + 2(X_{28} - X_{26}) + (X_{38} - X_{36})$ $Gy = (X_{36} - X_{16}) + 2(X_{37} - X_{17}) + (X_{38} - X_{18})$	$Gx = (X_{28} - X_{26}) + 2(X_{38} - X_{36}) + (X_{48} - X_{46})$ $Gy = (X_{46} - X_{26}) + 2(X_{47} - X_{27}) + (X_{48} - X_{28})$	$Gx = (X_{38} - X_{36}) + 2(X_{48} - X_{46}) + (X_{58} - X_{56})$ $Gy = (X_{56} - X_{36}) + 2(X_{57} - X_{37}) + (X_{58} - X_{38})$

By reviewing the calculations above, the following conclusions can be obtained:

For each row calculations:

i. For any vertical gradient Gx: Two out of three subtraction operations are repeated in the Gx calculation of the next row.

ii. For any horizontal gradient Gy: Each window operation performs two subtractions that were calculated in the previous window calculations or will be used in the next two window calculations.

iii. For a bigger size image, repeating the calculation in the next image line (Row 2 and Row 3) requires the data to be read again from the external RAM to the embedded RAM modules. This increases the memory read redundancy and hence memory latency.

iv. Recalculation requires more subtractors to exist in the architecture unnecessarily which increases the logic utilization of the processor.

In this thesis, the solution of all these design problems is accomplished by the use of known optimization techniques (refer to section 2.5) to design an architecture that is able to acquire real-time performance with a minimum amount of logic.

3.4 Optimizing Sobel Edge Detection Processor

The architecture proposed in this thesis is an optimization of the basic Sobel architecture analyzed in the previous section. The optimization is a combination of the optimization techniques mentioned in the chapter 2. The optimization is meant to increase the processor throughput, reduce the logic utilization and enable the processor to attain real-time performance at the same time. The optimization can reduce the memory read redundancy of the input data to a minimum and avoid the recalculations of the same data by using a processor that calculates several window operations in parallel. This way, the data are calculated once and reused several times inside the the processor block.

The Sobel processor is built out of several Sobel instances; each of them is able to produce one output pixel as shown in Fig. 3.4. The instance is built out of simple mathematical operators; subtractors (SUB), adders (ADD), shift operators (<<) and modulus operators (ABS). The difference between this Sobel instance and the basic Sobel instance is that the shift operator (<<) is separated from the subtractor (SUB). This enables reusing the subtractor result in future calculations.

The Sobel instance utilizes the loop fusion optimization technique by calculating the vertical and the horizontal gradient in parallel in the same instance. The instance is optimized using arithmetic identities technique to avoid multiplication by 1 and the addition by 0 during the calculation of the gradient. Avoiding unnecessary calculations will reduce the number of calculations and hence reduce the processing time and the logic utilization. While calculating the gradient, instances substitute the multiplication by 2 in the Sobel masks with a shift operation, i.e., reduction of strength. The shift operation is faster in execution and can be implemented with fewer logic elements than a multiplier would require. That makes the proposed processor a multiplier free processor. The comparison of the Sobel gradients with the threshold value is processed outside the Sobel instance as a loop invariant operation. The thresholding will increase the logic utilization if kept as part of the Sobel instance calculations because it requires placing a comparator in each Sobel instance.

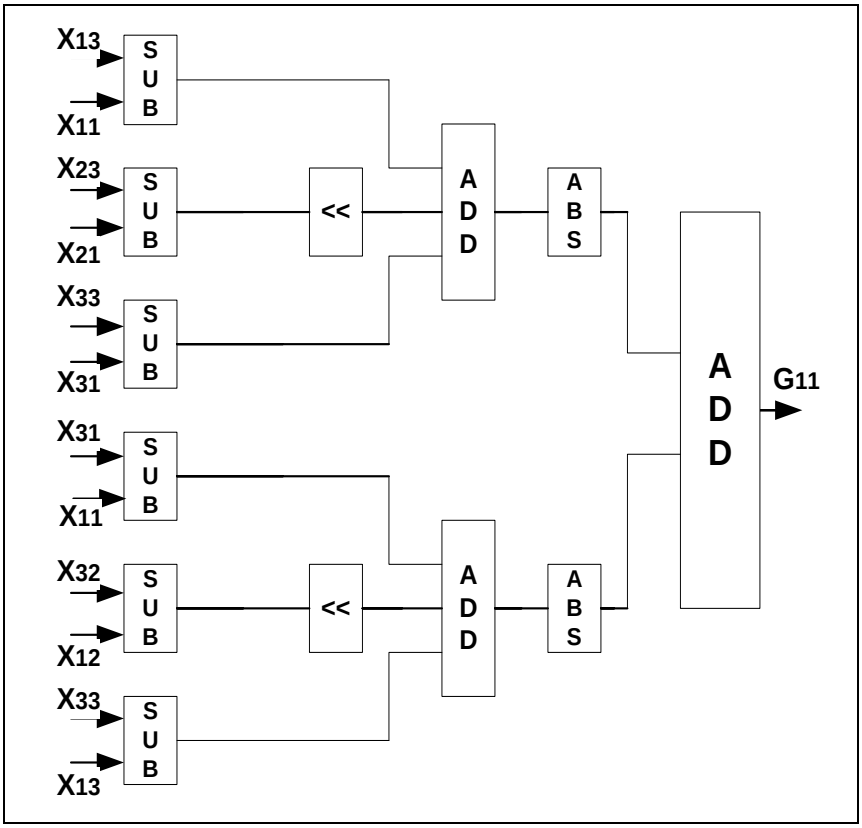


Figure 3.4 Basic Sobel Instance

It has been mentioned in section 2.5.4.5 that a linear algorithm such as edge detection can be altered to support parallel architecture format by the use of partial loop unrolling. This means, duplicating the Sobel instance to form one combinational block that can calculate several outputs in parallel. Fig.3.5 illustrates how the optimized Sobel processor block is partially unrolled by a factor of 4 to form a 1x4 Sobel processor block. The four Sobel instances are connected in a way that exploits FPGA parallelism and data reuse within one line of the input image. The thick vertical lines indicate the data reuse within the same image line. Depending on the number of instances in the Sobel processor block, the processor will gain performance by a factor equal to the number of instances used. Notice that the number of the Sobel instances in the optimized Sobel processor row can be increased as needed and is limited by the amount of logic available on the FPGA. As mentioned in the previous section, 2/3 of the subtractions performed while obtaining horizontal gradient G_y are repeated in the calculations of the next window operations within the same image line due to the high data dependency in the Sobel edge detection operator. Since the data of G_y in each

window calculations are reused in the next two window operations in the same line, the 1x4 processor-row fully reuse the data of the first two instances and partially reuse the data of the third one. Therefore, the number of four Sobel instances is chosen as the default number of the Sobel instances in the processor row. Increasing the number of instances in the processor row allows more opportunities for data reuse and increases the processor throughput but increases the logic utilization consequently. Using a parallel architecture and partial loop unrolling, the previous calculations can be reused within one image line. The vertical black lines in Fig 3.5 show how the calculations of G_y are reused in the following window operations within the same row. This way, 1x4 Sobel processor block eliminates recalculation within the same image line; i.e. implementing common subexpression elimination technique. The elimination of the common subexpressions reduces the number of subtractors in the design which reduces the total logical utilization.

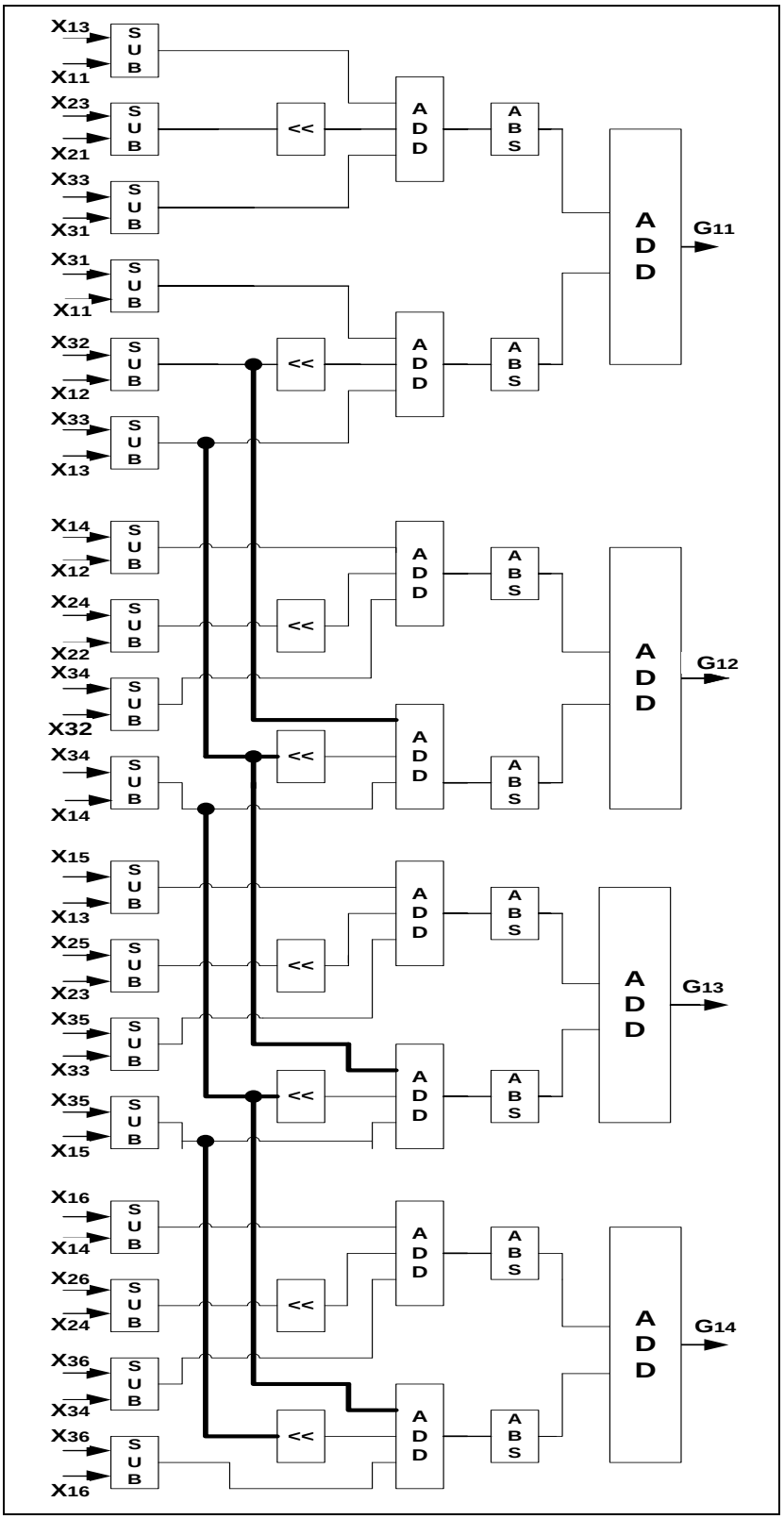


Figure 3.5 1x4 Sobel processor or one Sobel row

The concept of loop blocking can also be applied on the proposed Sobel processor architecture if we consider the data reuse of G_x calculations since 2/3 of the subtractions within G_x calculations are repeated in the calculations of other G_x calculation in the next row due to the data dependencies within the calculations of G_x. Combined with partial loop unrolling, loop blocking can be exploited to enable the data reuse within the calculations of multiple image lines at the same time. By exploiting the data reuse within two image lines, the processor instances are connected as shown in Fig. 3.6. Every four vertical consecutive instances are considered as a Sobel Processor row. The rows are connected to each other to form a Sobel processor block and exploit the data reuse. Fig. 3.6 details the connection of the first and second rows of the Sobel instances. The thick black vertical connection lines indicate data reuse within the same row (G_y calculation). The thicker black horizontal connection lines indicate computation data reuse from previous rows (G_x calculation).

In a 2x4 optimized Sobel processor block shown in Fig. 3.6 it can be noticed that, the number of subtractors in the second processor row is less (10 subtractors) than that of the first row (18 subtractors) which reduces the logic resources used. The optimized Sobel processor block may contain more than two rows. The following rows of instances are all similar to the second row and all are connected in the same way. The number of Sobel rows can be increased as needed and is limited by the available FPGA resources. Fig. 3.7 shows 4x4 optimized Sobel processor which is capable of producing sixteen 8-bit output pixels in one clock cycle.

Optimizing the design with loop unrolling and loop blocking requires the input data to be available at the execution time on the FPGA board which is solved by the use of scalar replacement technique that stores the required image lines as sets of shift registers connected as tap-delay-lines, i.e. line buffers. Scalar replacement is implemented using the embedded M4K RAM blocks that are available on the FPGA core [44]. The line buffer contents can be accessed concurrently to feed the Sobel processor block with input pixels enabling the calculation of multiple outputs in parallel every clock cycle. It should be noted that, increasing the size of the line buffers enables the exploit of more data reuse but will increase the logic utilization significantly. Additionally, a large line buffer has the disadvantage of a long overhead of filling the tabs before execution.

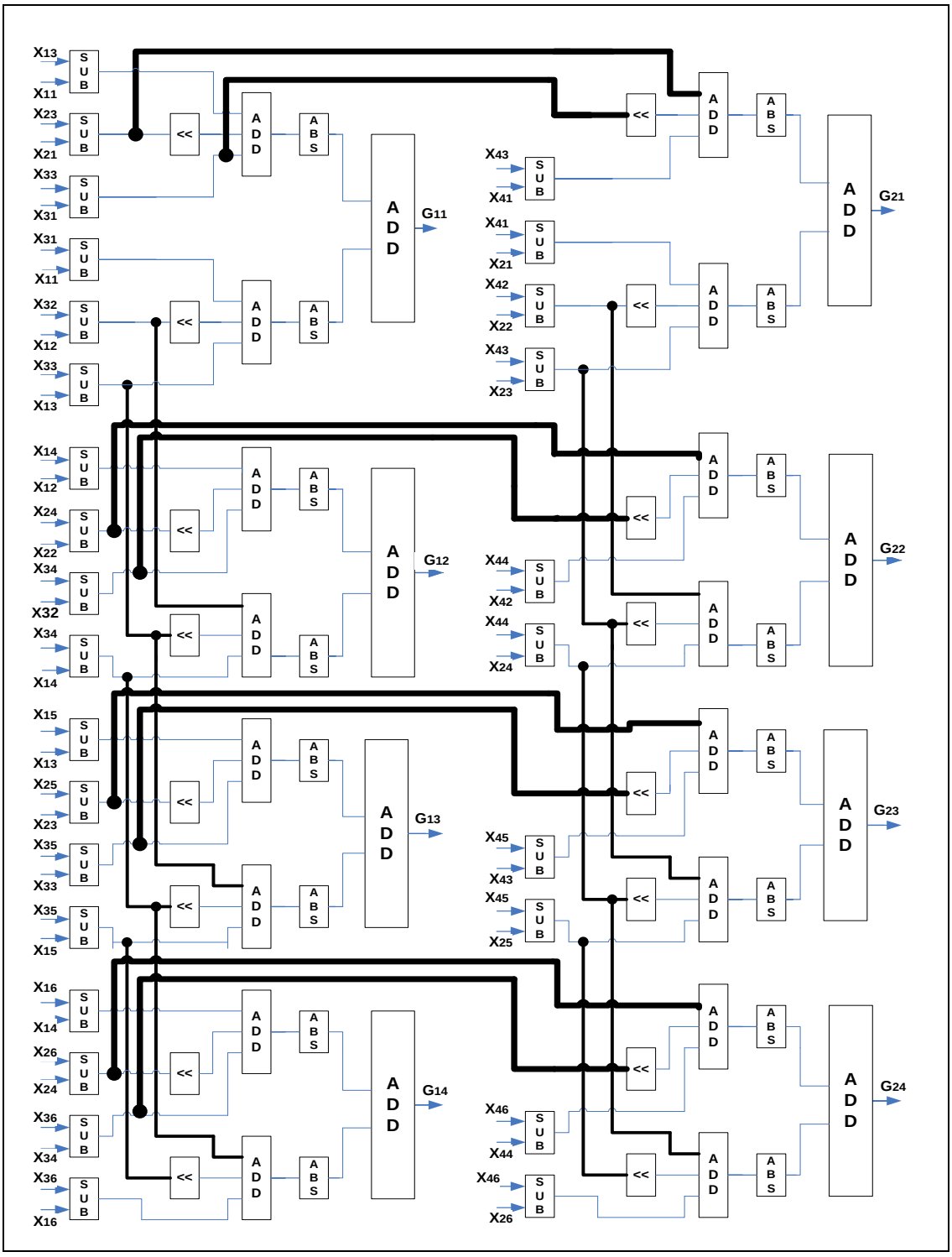


Figure 3.6 Data reuse within two consecutive image lines (2 Sobel processor rows)

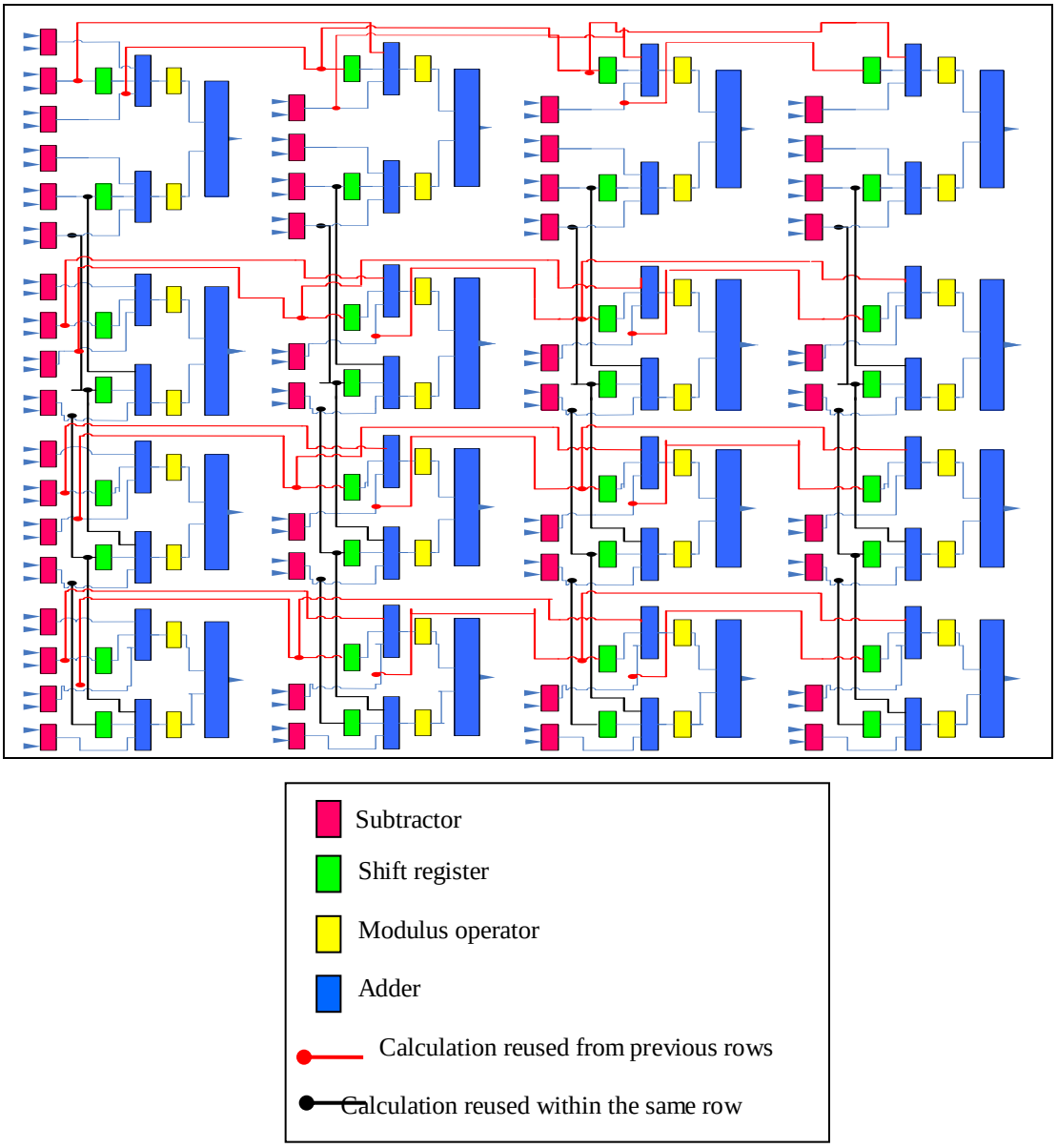


Figure 3.7 4x4 optimized Sobel processor

Unlike the basic Sobel architecture, the 4x4 Sobel block requires only 6x6 sub image to be available in the embedded FPGA memory to be fed to the Sobel block as inputs. Notice that, this is applicable only when the optimized Sobel processor is implemented with RAM modules like the ones mentioned in [36]. With the use of RAM modules, the architecture reduces the amount of on-chip input data that is needed to calculate the same number of output pixels by the basic architecture. For example, if two 4x4 Sobel blocks where used in parallel (32 outputs), only 6*10 pixels have to be stored in the RAM modules, i.e. 6 RAM modules containing 10 pixels (60 pixels) in each module. If the optimized processor uses the same number of pixels that the basic Sobel design uses (102 input pixels) in 6x17 RAM modules, the optimized processor will be able to produce 6x15 pixels in each line or 90 output pixels. Notice that, by the use of the partial loop unrolling and loop blocking only 2/6 lines will be reloaded for the calculations of new input pixels.

By using the same example of the 5x8 grayscale image used to analyze the basic Sobel architecture in section 3.3, an optimized Sobel processor can be built to match the size of the image. The processor should contain 3 rows of instances with 6 Sobel instances in each (3x6 block). This way, the processor can calculate the gradient of the image in one clock cycle. It can be shown how a 3x6 optimized Sobel processor block reuses the data in Table 3.2. The table shows the gradients (G) calculations for the first, second, and third image lines (rows). The optimized Sobel processor instance separates the shifting (<<) operator from the subtractor (SUB) which enables the reuse of the result of the SUB operation in other window operations or other rows. Each basic Sobel instance has six subtractors; three for each of the gradients.

Table 3.2 Window operations in Image line 1, 2 and 3 using optimized Sobel processor

Output	Row 1	Row 2	Row3
G_1	$Gx = (SUB_{11}A) + (SUB_{11}B<<) + (SUB_{11}C)$ $Gy = (SUB_{11}D) + (SUB_{11}E<<) + (SUB_{11}F)$	$Gx = (SUB_{11}B) + (SUB_{11}C<<) + (SUB_{21}C)$ $Gy = (SUB_{21}D) + (SUB_{21}E<<) + (SUB_{21}F)$	$Gx = (SUB_{11}C) + (SUB_{21}C<<) + (SUB_{31}C)$ $Gy = (SUB_{31}D) + (SUB_{31}E<<) + (SUB_{31}F)$
G_2	$Gx = (SUB_{12}A) + (SUB_{12}B<<) + (SUB_{12}C)$ $Gy = (SUB_{11}E) + (SUB_{11}F<<) + (SUB_{12}F)$	$Gx = (SUB_{12}B) + (SUB_{12}C<<) + (SUB_{22}C)$ $Gy = (SUB_{21}E) + (SUB_{21}F<<) + (SUB_{22}F)$	$Gx = (SUB_{12}C) + (SUB_{22}C<<) + (SUB_{32}C)$ $Gy = (SUB_{31}E) + (SUB_{31}F<<) + (SUB_{32}F)$
G_3	$Gx = (SUB_{13}A) + (SUB_{13}B<<) + (SUB_{13}C)$ $Gy = (SUB_{11}F) + (SUB_{12}F<<) + (SUB_{13}F)$	$Gx = (SUB_{13}B) + (SUB_{13}C<<) + (SUB_{23}C)$ $Gy = (SUB_{21}F) + (SUB_{22}F<<) + (SUB_{23}F)$	$Gx = (SUB_{13}C) + (SUB_{23}C<<) + (SUB_{33}C)$ $Gy = (SUB_{31}F) + (SUB_{32}F<<) + (SUB_{33}F)$
G_4	$Gx = (SUB_{14}A) + (SUB_{14}B<<) + (SUB_{14}C)$ $Gy = (SUB_{12}F) + (SUB_{13}F<<) + (SUB_{14}F)$	$Gx = (SUB_{14}B) + (SUB_{14}C<<) + (SUB_{24}C)$ $Gy = (SUB_{22}F) + (SUB_{23}F<<) + (SUB_{24}F)$	$Gx = (SUB_{14}C) + (SUB_{24}C<<) + (SUB_{34}C)$ $Gy = (SUB_{32}F) + (SUB_{33}F<<) + (SUB_{34}F)$
G_5	$Gx = (SUB_{15}A) + (SUB_{15}B<<) + (SUB_{15}C)$ $Gy = (SUB_{13}F) + (SUB_{14}F<<) + (SUB_{15}F)$	$Gx = (SUB_{15}B) + (SUB_{15}C<<) + (SUB_{25}C)$ $Gy = (SUB_{23}F) + (SUB_{24}F<<) + (SUB_{25}F)$	$Gx = (SUB_{15}C) + (SUB_{25}C<<) + (SUB_{35}C)$ $Gy = (SUB_{33}F) + (SUB_{34}F<<) + (SUB_{35}F)$
G_6	$Gx = (SUB_{16}A) + (SUB_{16}B<<) + (SUB_{16}C)$ $Gy = (SUB_{14}F) + (SUB_{15}F<<) + (SUB_{16}F)$	$Gx = (SUB_{16}B) + (SUB_{16}C<<) + (SUB_{26}C)$ $Gy = (SUB_{24}F) + (SUB_{25}F<<) + (SUB_{26}F)$	$Gx = (SUB_{16}C) + (SUB_{26}C<<) + (SUB_{36}C)$ $Gy = (SUB_{34}F) + (SUB_{35}F<<) + (SUB_{36}F)$

Referring to the calculations in Table 3.1, the subtractors for calculating the vertical gradient Gx are named $SUB_{nm}A$, $SUB_{nm}B$, $SUB_{nm}C$, where n and m represent the row and column in the input image matrix. The subtractors for the horizontal gradient Gy are named $SUB_{nm}D$, $SUB_{nm}E$, $SUB_{nm}F$ as shown in Table 3.2. Naming the subtractors enables the optimized Sobel processor to calculate every subtraction operation once and reuse the value in the other instances in the Sobel block. In this example, the output of the optimized processor is 3x6 gradient pixels calculated in one clock cycle.

In case of an input image with larger frame size, the optimized Sobel processor will have a smaller size than the image line and it has to be slid over the image pixels. The size of the processor is only limited by the logic resources and it is possible to use a single row processor or multiple processor rows as shown in Fig. 3.5 and Fig. 3.7, respectively. Implementing a single-row optimized Sobel processor with a certain size requires a special line buffer to be designed according to the number of Sobel instances inside the optimized Sobel block. The line buffer handles the data from an

external memory and feeds the required amount of pixels to the processor determined by the processor size. In case of using optimized Sobel processor that contains multiple rows of instances, another line buffer should be designed and integrated with the Sobel processor to feed the data back to the external memory in a raster order (row by row) [34]. Therefore, the multiple rows optimized Sobel processor will calculate the gradients for a certain number of pixels from several image lines depending on the number of rows in the processor block and the number of instances in each row.

By reusing the calculation results of previous window operations, our proposed architecture reduces the number of subtraction operations of the basic architecture [36] by 50%. The data reuse method reduces the processing time because each pixel needs to be read from memory at most twice instead of up to six times.

3.5 The Line Buffer Module

In this thesis, the one-row 1x4 Sobel processor that exploits the data reuse within one image line is implemented on FPGA as a complete edge detection system. A line buffer was designed to feed the required number of input pixels to the processor in every clock cycle. This enables the process to achieve real-time performance.

The line buffer module is utilized to feed the grayscale 8-bit pixels from the input image stored in the SRAM to the Sobel processor. As illustrated by Fig. 3.8, the line buffer has the capacity to hold 3 lines of pixels. Each line consists of 160 memory locations; each location is 32-bit in length; to store a total of 640 grayscale pixels (4 pixels in each location). The three lines accommodate a total of 1920 pixels in 480 locations. The line buffer is fed by a FIFO [45] that shifts a 32bit long word (4 pixels) from the SRAM to the line buffer in every clock cycle. The line buffer has one write pointer (wp) and three read pointers (rp0, rp1 and rp2) to read from each of the three image lines stored in the line buffer. All the pointers are set to point to the 0 location whenever the asynchronous clear (aclr) signal is asserted to reset the buffer. When the write pointer reaches the location number 479, it will point back to location 0 to fill up the first location again. The read pointers rp0, rp1, and rp2 will point to the start of each image line before the reading is started, i.e. rp0= 0, rp1= 160 and rp2= 320. The

read pointers are incremented to point to the next memory location in every clock cycle. To read the image pixels in the line buffer, the memory location which is pointed by the read pointer will be stored in a 32-bit register (pixout). Since six pixels are needed from each image line to calculate four output pixels, two (pixout) registers are needed for each image line. At every rising edge of the clock, the pixels in the first register, e.g. (pixout0) will be replaced by the previous values of the second register (pixout1). The registers are wired to the line buffer output taps (taps00 to taps25).The taps are wired to the Sobel processor to feed the pixels for calculating the edges.

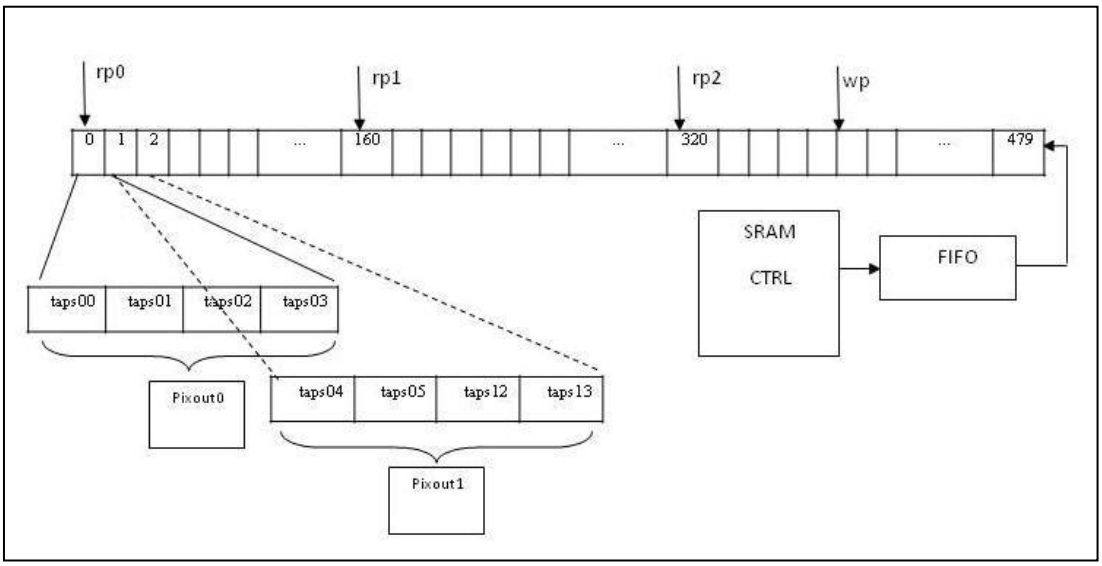


Figure 3.8 Line buffer for a 1x4 optimized Sobel processor

3.6 Implementing the Optimized Processor on FPGA

In order to implement the single row optimized Sobel processor on FPGA to run in real-time, the processor has to be integrated with a digital camera, an external memory, and a PC monitor to work as a complete edge detection system on the FPGA board. As illustrated in Fig. 3.9, the digital camera captures a video stream which represents the system input. The input frames are stored in the FPGA board static random access memory SRAM memory. The line buffer of the optimized Sobel processor is connected to the SRAM in order to pass the input pixels to the optimized Sobel processor. The Sobel processor writes the output pixels back to the SRAM. The monitor reads the processor output pixels from the SRAM to be shown on the screen.

The SRAM connects to the Sobel processor by the use of a multiport memory controller module. The multiport memory controller module connects the SRAM to the camera by a camera controller module. The camera controller will transform the raw image pixels into RGB and change the RGB to grayscale pixels and write them to the SRAM. This way, 2/3 of the SRAM space is saved.

Four FIFO modules designed by [45] were used to synchronize the SRAM with the other devices. The FIFO1 is used to connect the camera controller module with the SRAM multiport memory controller. The FIFO2 synchronizes the SRAM controller with the line buffer of the Sobel processor. The FIFO3 enables the Sobel processor to write the output pixels back to the SRAM by passing them to the multiport memory controller module. The FIFO4 connects the multiport memory controller module to the video graphic adapter (VGA) controller module that connects the monitor to the multiport SRAM controller.

All the system modules are designed using the Verilog hardware description language (HDL). The functionality of the system was tested using Modelsim-Altera 6.1g software. Altera Quartus® II web edition version 9.1 software was utilized to run the timing simulation of the design and to download the system design files on Altera Cyclone® II DE2-70 EP2C70F896C6 FPGA. The Altera DE2-70 board has a 2MB SRAM [46] which is used as the external memory and a frame buffer of the edge detection system. The TRDB_D5M 5 Mega pixel digital camera from Terasic was used for video stream capture which can be simply installed on any Altera DE2-70 board [47].

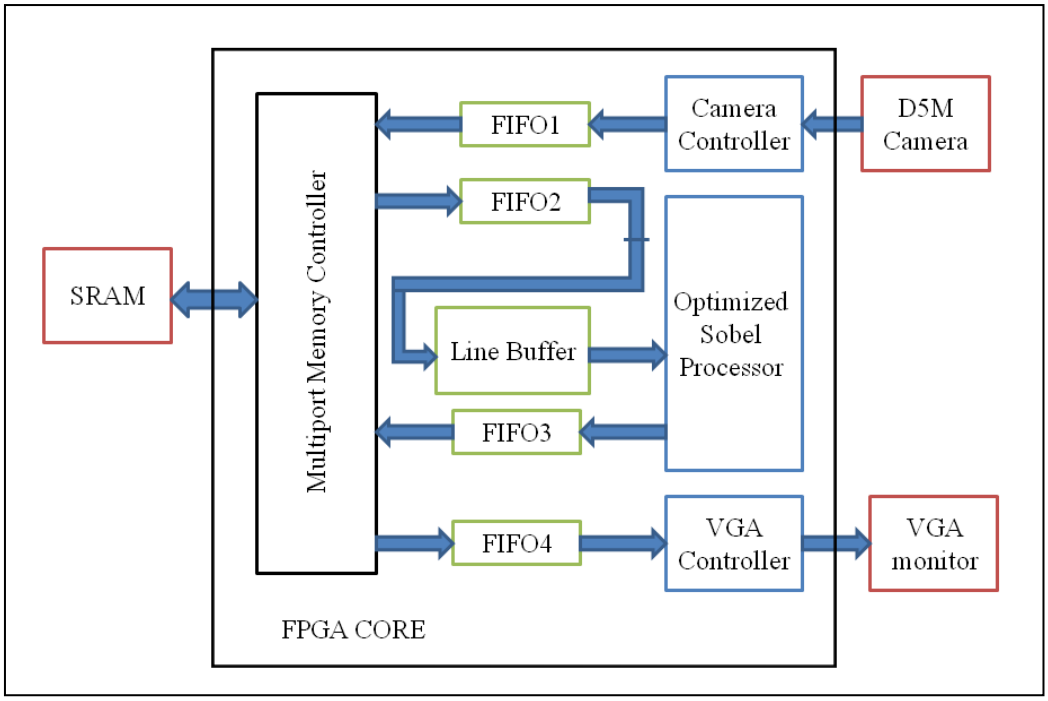


Figure 3.9 Edge detection system

In the next chapter, the system implementation on Cyclone II FPGA is discussed and the system specifications are listed. The evaluation of the processor logic utilization and performance is also included in the next chapter. The chapter also shows how the proposed processor can satisfy real-time performance constrains.

3.7 Chapter Summary

In this chapter the proposed Sobel edge detection processor is presented. It is discussed how the proposed architecture is optimized by the use of the proper optimization technique based on the basic Sobel processor analysis. The chapter also discussed how the optimized Sobel processor utilizes the data reuse in one image line and how it can be constructed to utilize the data reuse within multiple image lines. A line buffer designed specifically to work with a 1x4 Sobel processor row was proposed. Finally, the chapter summarized how the optimized Sobel processor along with its line buffer can be integrated on an FPGA board to work as a real-time edge detection system.

CHAPTER 4

RESULTS AND DISCUSSION

4.1 Introduction

In chapter 3, the optimized Sobel edge detection processor is discussed. The processor architecture enhances the reuse of data to reduce the logic utilization and satisfy real-time systems constraints. It is also explained how the Sobel optimized processor is integrated with a line buffer and other controlling modules to be able to perform as a complete real-time edge detection system on an FPGA board.

This chapter discusses the implementation of the proposed Sobel processor architecture on an FPGA. It starts with the evaluation of the logic utilization of the Sobel processor. The evaluation aims to show the effect of the optimization applied on the processor in comparison with the referenced Sobel architecture [36]. The performance of the optimized Sobel processor as part of a real-time edge detection system is discussed. Finally a comparative performance analysis is carried out between the optimized processor and other processors mentioned in section 2..

4.2 Logic Utilization of the Optimized Sobel Processor

The proposed Sobel processor architecture was synthesized on Cyclone II DE2-70 development kit EP2C70F896C9 FPGA to evaluate the logic utilization of different sizes of the proposed processor. The evaluation is meant to show the effect of the optimization carried out on the processor in terms of reduction in the logic utilization, memory usage, and the processor execution time. The logic utilization includes the number of logic elements occupied when implementing the processor

combinational block on the Cyclone DE2-70 board. In this step, the evaluation measures the number of the logic elements occupied by the processor combinational circuit and excludes registers and line buffers that are part of the Sobel processor.

The logic utilization evaluation is carried out using several sizes of the Sobel processor combinational block with different number of Sobel instances in each block. It starts by estimating the effect of the data reuse within one line of the input image by evaluating the logic utilization of a single-row optimized Sobel processor. Different sizes of the single-row processor are examined for logic utilization. In the next stage, the data reuse within multiple image lines is estimated using an optimized Sobel processor block that consists of multiple rows of the Sobel instances. A total of four Sobel instance has been chosen as the default number of instances to build one row of the processor block as explained in section 3.4. Different sizes of the optimized Sobel processor block were also examined for logic utilization percentage to evaluate the effect of the optimization on each processor configuration.

For a single row optimized Sobel processor, a processor row consists of four Sobel instances and requires 548 logic elements (LEs) or less than 1% out of the 68,416 LEs available on the Cyclone II FPGA. A row of 16 Sobel instances occupies 2,146 (3.1%) LEs. An optimized Sobel processor that consists of 32 instances requires 4,275 (6.2%) LEs. Table 4.1 summarizes the logic utilization of a single row optimized Sobel processor.

Table 4.1 The logic utilization of different sizes of a single row optimized Sobel processor

Number of Sobel instances in one row	**Number of logic elements**
4 Optimized Sobel instances	548 (<1%)
16 Optimized Sobel instances	2,146 (3.1%)
32 Optimized Sobel instances	4,275 (6.2%)

Table 4.2 shows the logic utilization for different configurations of an optimized Sobel processor block with multiple rows of Sobel instances. Two rows of Sobel instances (2x4 block) need 1,020 LEs (1.4%) to be implemented on DE2-70 board and process eight pixels in one clock cycle. An optimized Sobel processor block with

four rows (4x4 block) requires 1,992LEs (2.9%) to produce 16 output pixels. While a block of 32 instances enclosed in an 8x4 optimized Sobel processor block consumes 5.3% (3,668 LEs) of the total logic elements.

Table 4.2 The logic utilization of different sizes of optimized Sobel processor block which consist of multiple rows

Number of rows in the optimized Sobel processor block	**Number of logic elements occupied**
2 rows (2x4 Sobel instances)	1,020 (1.4%)
4 rows (4x4 Sobel instances)	1,992 (2.9%)
8 rows (8x4 Sobel instances)	3,668 (5.3%)

Using a single row processor allows the data to be reused only in the calculation of the horizontal gradient Gy. Therefore, the logic utilization of the optimized Sobel processors that use multiple rows of instances is less than that of the optimized Sobel processors that use a single row of Sobel instances. Multiple rows Sobel processor reuses not only the horizontal gradient Gy calculations but the vertical gradient Gx calculations as well. The data reuse will reduce the number of subtractors in the optimized Sobel block. The result of a certain subtraction operation is reused in up to two subsequent window operations within the same image line or in next two image lines.

In a 4x4 optimized Sobel processor block, the processor exploits the data reuse in the vertical and the horizontal gradient calculations. As illustrated in Fig. 3.7, all the rows of the 4x4 optimized Sobel block have 10 subtractors except for the first row which has 18 subtractors since there are no vertical gradient data reuse in the calculation of the first image line. The data reuse reduces the number of subtractors in the processor significantly and hence reduce the size of the processor combinational block. Table 4.3 and Table 4.4 show the number of subtractors in the different sizes of the optimized Sobel processor in case of the data reuse within one image line (single-row optimized Sobel processor) and within multiple image lines (optimized Sobel processor block with multiple rows), respectively.

Reviewing both tables we can conclude that, the data reuse within multiple lines within the input image reduces the number of subtractors by ((66-48)/66*100% =27%) using a 4x4 optimized Sobel processor block compared to that of a single row optimized Sobel processor which has the same number of instances. The reduction reaches 32% in the case of 8x4 block size (32 Sobel instances) compared to a single optimized Sobel processor with the same number of Sobel instances. Note that the percentages is calculated using the formula [(a-b)/b]*100%, where b > a.

Table 4.3 The number of subtractors in different sizes of single row optimized Sobel processor (data reuse in one image line)

Number of instances in the optimized Sobel row	**Total number of subtractors**
4 (1x4) Sobel instances	18
16 (1x16) Sobel instances	66
32 (1x32) Sobel instances	130

Table 4.4 The number of subtractors in different sizes of multiple rows optimized Sobel processor block (data reuse in multiple image lines)

Number of instances in the optimized Sobel processor block	**Total number of subtractors**	**Reduction from table 4.3**
8 (2x4) Sobel instances	28	_
16 (4x4) Sobel instances	48	27%
32 (8x4) Sobel instances	88	32%

In this chapter, the optimized Sobel processor is evaluated by comparing with other Sobel processors mentioned in the literature. Initially, the logic utilization is evaluated by comparing with the referenced processor proposed by [36]. The processor is synthesized on the Altera DE2-70 board. Several sizes of the Sobel processor proposed by [36] were examined for logic utilization for comparison with the proposed architecture. The size of the processor is changed by changing the number of the Sobel instances contained in processor. The results in Table 4.5 show the resource utilization for different sizes of the referenced processor [36]. For a

processor with four Sobel instances, the number of logic elements required to implement the processor combinational block is 574 LEs, 2,242 LEs are occupied by a processor with Sixteen Sobel instances and 4,467 LEs are required for 32 Sobel instances processor. On the other hand, the proposed processor reduces the logic resources to 548 LEs (4.5%) for a processor with 4 Sobel instances, 1,992 LEs (11.2%) for a 16 Sobel instances processor, and 3,668 (17.88%) for 32 Sobel instances.

Table 4.5 The logic utilization of different sizes of the parallel Sobel processor in [36]

Sobel processor size	**Number of logic elements occupied by [36]**	**Number of logic elements occupied by the proposed processor**
4 Sobel instances	574 (<1%)	548 (<1%)
16 Sobel instances	2,244 (3.2%)	1,992 (2.9%)
32 Sobel instances	4,467 (6.5%)	3,668 (5.3%)

The logic resources comparison between the optimized Sobel processor and the referenced Sobel processor [36] is illustrated by the bar chart in Fig. 4.1. The X axis represents the number of instances in the processor and the Y axis represents the number of LEs occupied by each processor with different configurations. The graph shows how the difference in logic elements between the two processors increases as the size of the processor is increased. The difference in the number of logic elements occupied by the processors increase from 26 LEs difference in the case of 4 Sobel instances, to 250 for 16 Sobel instances and reaches 799 LEs for 32 Sobel instances.

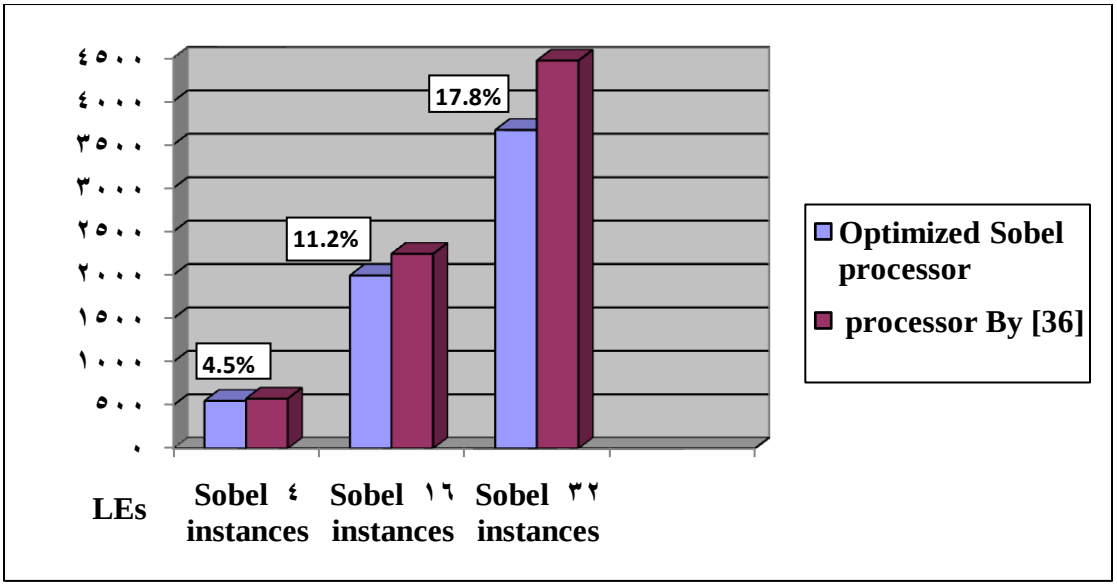


Figure 4.1 The logic utilization (in LEs) comparison between the proposed optimized Sobel processor and the referenced Sobel processor in [36]

The optimized Sobel processor has fewer subtractors in its architecture than the Sobel processor in [36] as summarized in Fig. 4.2. The X axis represents the number of subtractors in the processor block and the Y axis represents the processor configurations. It is mentioned in [36] that the basic Sobel instance has 6 subtractors which makes the total number of subtractors equals to 192 in a Sobel processor that consists of 32 instances. A processor with 16 instances contains 96 subtractors and 24 subtractors are enclosed in a processor with 4 instances. In comparison, the optimized Sobel processor contains 54% less subtractors (88 subtractors) for a processor with 32 instances, 50% less when a processor with 16 instances is utilized. An optimized processor with 4 instances acquires (18 subtractors) 25% reduction in the number of subtractors over a processor of the same configurations by [36]. The reduction in the number of subtractors in the processor reduces the processor size which allows adding more Sobel instances in the processor block. Adding more instances enables the processor to process more pixels in parallel and hence increases the processor throughput and the overall frame rate.

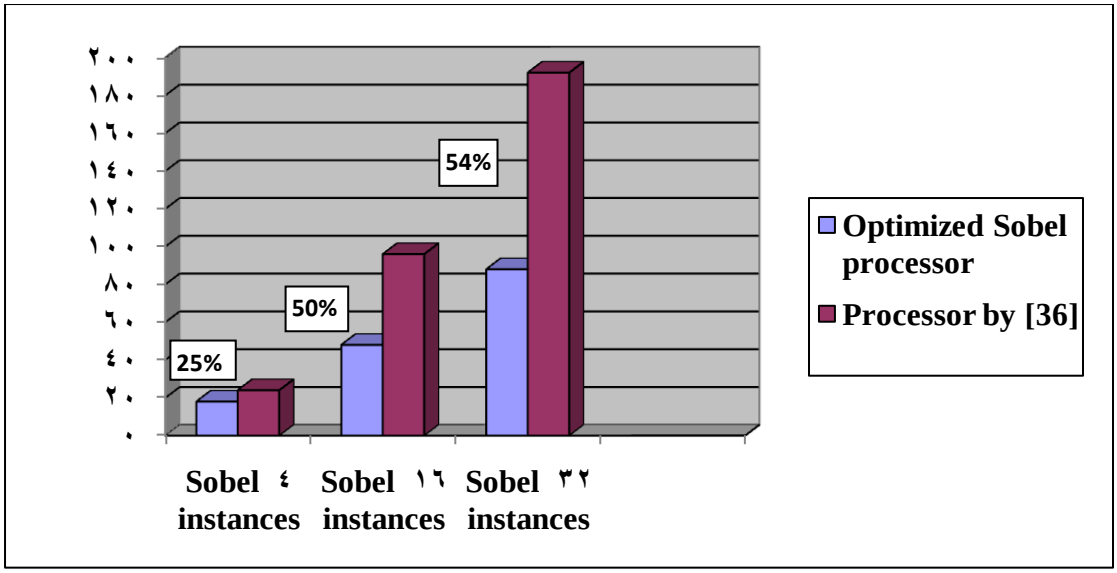


Figure 4.2 The comparison between the optimized Sobel processor and the referenced Sobel processor in [36] in terms of the number of subtractors

The effect of the optimization on the memory usage is evaluated. The proposed Sobel processor is implemented using the RAM modules that were used by [36] as mentioned in section 2.6.1. The processor optimizes the use of the input data required to produce the out pixels by exploiting the loop unrolling and loop blocking. For a 32 instances (4x8) optimized Sobel processor, the processor requires 60 input pixels (60 Bytes) to calculate the 32 output pixels (6 RAM modules with only 10 pixels each). However, the Sobel processor in [36] requires 102 input pixels arranged in 3 RAM modules with 34 pixels each (102 Bytes) to calculate 32 output pixels. By calculating the reduction percentage, the optimized Sobel processor uses 41% less RAM modules registers compared with [36] to produce the same number of output pixels (32 pixels). The processor of 16 instances can reduce the memory utilization to 36 bytes instead of 54 bytes in the case of [36]. The comparison result is shown in Table 4.6.

It should be considered that the optimized Sobel processor does not use the RAM modules and it uses a line buffer specially designed to handle the data in a special way that matches the optimized Sobel processor and to assure a performance that satisfies the real-time constraints. The line buffer architecture is described in section 3.5.

Table 4.6 Memory utilization comparison between the proposed processor and the processor in [36]

Sobel processor size	**Memory utilization for [36] (Bytes)**	**Memory utilization for the proposed processor (Bytes)**
4 Sobel instances	18	18
16 Sobel instances	54	36
32 Sobel instances	102	60

4.3 The Optimized Sobel Processor Performance Evaluation

In this section, an evaluation of the proposed Sobel processor is demonstrated. The following subsection (4.3.1) states the specifications of the edge detection system that the optimized Sobel processor is integrated in. This is followed by a comparative analysis between the optimized processor performance and the performance of Sobel processors from this research literature in subsection 4.3.2.

4.3.1 The Edge Detection System Specifications

In section 3.6, it is shown how the optimized Sobel processor is integrated in a complete edge detection system and implemented on a DE2-70 FPGA board. The implementation aims to test the ability of the proposed architecture to satisfy real-time image processing systems constraints.

The D5M Terasic digital video-camera operates with a 25MHz frequency to provide the system with input frames of 640x480 pixels. A 640x480 frame contains 307200 pixels represented by four colors (green1, green2, red, and blue) [36]. This provides a camera data rate (DR_{CCD}) of 25 mega pixel per second (MP/s) and a camera frame rate of (25MP/307200/4=20) 20 frames per second (fps). The frames are stored in the 2MB SRAM on the DE2-70 board which operates with a frequency of 100MHz [46]. The SRAM reads 4 pixels (32bits) from the D5M camera every clock cycle which allows a maximum data rate (DR_{SRAM}) of 400MP/s (4x100=400). The VGA controller runs at 28MHz frequency and has a 28MP/s data rate (DR_{VGA}) [46]. The SRAM, VGA, and camera controller specifications are summarized in Table 4.7.

Table 4.7 System Specifications summary

Component	Data rate (MP/s)	Frequency (MHz)
SRAM controller	400	100
VGA controller	28	28
CCD camera controller	25	25

The optimized 1x4 Sobel processor row and its line buffer are running with a 10MHz clock frequency. With this frequency, one Sobel instance processor can calculate 10MP/s. The processor row consists of 4 Sobel instances that enable the processor to produce 4 pixels in each clock cycle, which increases the processor throughput (the number of calculated pixels per second) to 40MP/s. Since the D5M camera provides a 640x480 frame size that contains a total of 307200 pixels, dividing 40MP/s by 307,200 pixels per frame results in 130.2fps (40M/307200=130.2). The processor specifications are summarized in Table 4.8. It should be considered that this is not the maximum clock frequency of the optimized Sobel processor, neither its maximum throughput. This frequency is chosen to match the SRAM maximum data rate and frequency. Moreover, a frequency of 10MHz can produce a frame rate that satisfies the real-time constraints (minimum 30 fps).

Table 4.8 Optimized Sobel processor specifications summary

Processor size	Frequency (MHz)	Throughput (MP/s)	Frame rate (fps)
(1x4) instances	10	40	130.2

The maximum frequency the optimized processor can achieve and its maximum throughput and frame rate are reported by the Quartus II Classic Time Analyzer and are summarized in Table 4.9. The processor has a maximum throughput of 367.77 MP/s and a maximum frame rate of 1196.02fps.

Table 4.9 Optimized Sobel processor maximum performance specification

Processor size	**Maximum Frequency (MHz)**	**Maximum Throughput (MP/s)**	**Maximum Frame rate (fps)**
(1x4) instances	91.86	367.44	1196.02

The FIFO modules designed by [45] are utilized in the system to synchronize each module with the others. Therefore, the FIFO read clock matches the read source clock and the write clock should match the write destination clock frequency. As illustrated in Fig. 3.9, FIFO1 connects the camera controller with the multiport SRAM controller. Therefore, it has a read clock similar to the camera controller frequency (25MHz) and writing frequency equals to the SRAM controller clock (100MHz). FIFO2 reads with a 100MHz clock frequency (SRAM) and writes to the line buffer with a 10MHz clock frequency. FIFO3 read clock follows the optimized Sobel processor clock (10MHz) and writes to the SRAM with a 100MHz clock frequency. FIFO4 follows the SRAM clock frequency (100 MHz) for reading and the VGA controller clock frequency (28MHz) for writing. The FIFOs specifications are summarized in Table 4.10.

Table 4.10 System FIFOs specifications

FIFO	**Read Clock (MHz)**	**Write Clock (MHz)**
FIFO1	25	100
FIFO2	100	10
FIFO3	10	100
FIFO4	100	28

The complete edge detection system, when compiled with Altera Quartus II software for Cyclone II DE2-70, consumes 2,151 LEs (3%) out of 68,416 LEs available on the DE2-70 board. A total of 1,973 LEs were consumed by the total combinational functions and 962 LEs were implemented as dedicated logic registers. The design has a total number of 56,320 memory bits which is 5% out of the 1,152,000 memory bits implemented using the M4K blocks on the DE2_70 board. The logic utilization of the system is detailed in Table 4.11.

Table 4.11 Logic utilization summary of the proposed edge detection system based on 1x4 optimized Sobel processor

Module	**Combinational logic (LEs)**	**Registers (LEs)**	**M4K Memory Bits (Bit)**
Camera controller	340	226	5120
SRAM controller	456	137	0
VGA controller	110	46	512
Sobel processor	548	0	0
Sobel line buffer	72	145	49152
FIFO1	28	40	512
FIFO2	64	73	512
FIFO3	29	45	512
FIFO4	29	73	512

The edge detection system timing analysis is performed using the Quartus II Classic Timing Analyzer tool. The worst-case clock setup time (t_{SU}) which is the length of time for which data that feeds a register via its data or enable input(s) must be present at an input pin before the clock signal that clocks the register is asserted at the clock pin is reported by the timing analyzer. The worst-case clock to output delay (t_{CO}) which is the maximum time required to obtain a valid output at an output pin after a clock transition at an input clock pin, directly or through a register is also calculated by the timing analyzer. The Timing Analyzer reports a t_{CO} of 18.726ns and t_{SU} of 11.051ns. The worst-case of the propagation delay (t_{PD}) is estimated to be 10.630ns which is the time required for a signal from an input pin to propagate through combinational logic and appear at an external output pin.

The Quartus II Classic Timing analyzer is used to analyze the performance of module that combines the Optimized Sobel processor with the line buffer. The timing analyzer report shows the best-case timing model for the processor. The report is summarized in Table 4.12. The timing model reports a worst-case t_{SU} of 5.477ns. The worst-case t_{CO} is 6.678ns. The fast timing model shows a maximum frequency of the line buffer clock of 91.86MHz with a 10.886ns clock period which is the maximum frequency of the processor.

Table 4.12 Timing analysis summary for the optimized Sobel processor

Type	Timing analysis
worst-case t_{SU}	5.477ns
worst-case t_{CO}	6.678ns
Clock setup "line buffer Clock"	91.86MHz (10.886ns CLK period)

The system generates an edge-detected real-time video stream successfully. A snapshot of the output video stream on the VGA monitor is shown in Fig. 4.3 and Fig 4.4. Fig. 4.3 shows a snapshot of Susan Lordi's sculpture with black background. The system does not contain any sort of noise reduction and therefore, the result snapshots are noisy. Therefore, the comparison between the gradient and the threshold image is inverted to produce 0 (black) when the gradient value is greater than the threshold value (an edge) and 1 (white) otherwise (not an edge). The inversion of the thresholding result enables easier observation of the black edges in a white background in the presence of noise.

Figure 4.3 Grayscale snapshot

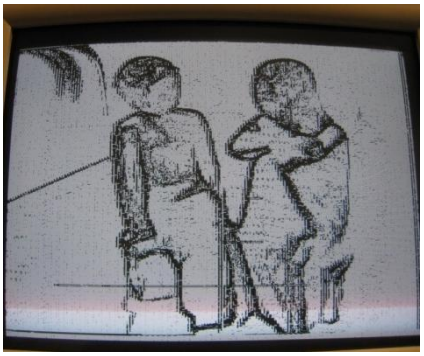


Figure 4.4 Sobel edge detection snapshot

4.3.2 Comparative Analysis of Sobel Processors

This section conducts a comparative analysis between the proposed image edge detection system based on the optimized Sobel edge detection processor and other Sobel edge detection systems performances.

The comparison is performed between the 1x4 optimized Sobel processor and four Sobel processors proposed by [22, 37, 42]. The comparison is based on two characteristics: clock frequency and processor throughput. These two characteristics determine whether or not the system can satisfy real-time systems constraints. A system with a high clock frequency is desirable for its high processing speed which increases the system throughput accordingly. On the other hand, a high clock frequency increases the power consumption of the system which makes it less efficient for mobile application [29, 48-49]. The comparison shows that the optimized Sobel processor is able to achieve a significantly high throughput with a considerably low frequency.

Since the optimized Sobel processor is implemented using the Altera Cyclone II DE2-70 [51], while the other processors were implemented on Xilinx FPGA (Spartan 3 for [37], XC3042 for [42] and Virtex 4 for [22], the architecture differences between the two vendors must be considered before the evaluation. For instance, the Altera Cyclone II DE2-70 FPGA consists of logic array blocks (LABs) that contain 16 LEs in each. Each logic element contains a single 4 inputs type look up table (4-LUT) and other logic for multiplexing, addition and some registers [44, 50], While the Xilinx FPGAs from Virtex 4 family and earlier families (includes Spartan 3 family) consist of slices, each of which consists of two 4-LUTs and other logic for multiplexing,

adding and registers. The Xilinx XC3042 FPGA contains Configurable Logic Blocks CLBs. The CLB architecture is based on 32x1 LUT that can be separated into two 4-LUTs to implement two logic functions with 4 inputs each [51]. Therefore, the comparison between the two vendors should be based on the number of LUTs occupied by the architectures in each FPGA platform because the architecture of the LUT is nearly the same.

The comparison between the five architectures is shown by Table 4.13. It should be considered that the logic utilization comparison between the different platforms is a best estimate because the logic utilization does not take into account the number of flip flops, multiplexers, and routing logic which are essential parts of the LE, slice, and (CLBs).

Table 4.13 Logic utilization comparison

Sobel Processor	**FPGA type**	**Architecture configurations**	**Number of 4-LUTs (equivalent)**
The optimized Sobel processor	Cyclone II DE2-70	1x4	548
Processor proposed by [37]	Xilinx Spartan 3	1x1	202
Processor 1 proposed by [22]	Xilinx Virtex 4	1x1	570
Processor 2 proposed by [22]	Xilinx Virtex 4	1x1	701
Processor proposed by [42]	Xilinx XC3042	1x1	140

It is seen from Table 4.13 that the optimized Sobel processor requires 548 4-LUTs since the LE of DE2-70 board consists of one 4-LUT. The processor proposed by [37] occupies 202 4-LUTs of the Xilinx Spartan 3. The architecture has less LUTs than the optimized Sobel processor because it consists of a single Sobel module that calculates one output pixel in every clock cycle. Processor 1 proposed by [22] uses the data reuse by exploiting scalar replacement technique within the horizontal calculations by saving the reused data in dedicated logic registers to be used in the next clock cycle.

Therefore the logic utilization is close to that of the proposed architecture. Processor 2 proposed by [22] utilizes the data reuse in the horizontal and vertical direction and saves the reused data in the block RAM of the Xilinx Virtex 4 FPGA. The processor occupies slightly more LUTs than the optimized Sobel processor. The Sobel processor presented by [42] is implemented on Xilinx XC3042 device with 70 CLBs which are equivalent to 140 LUTs. Considering the configurations of the processors in the table, the optimized Sobel processor contains 4 Sobel instances to calculate 4 output pixels at the same time while all the other processor contain only one Sobel instance. This makes the optimized processor the least logic utilization in comparison with the other processor in Table 4.13.

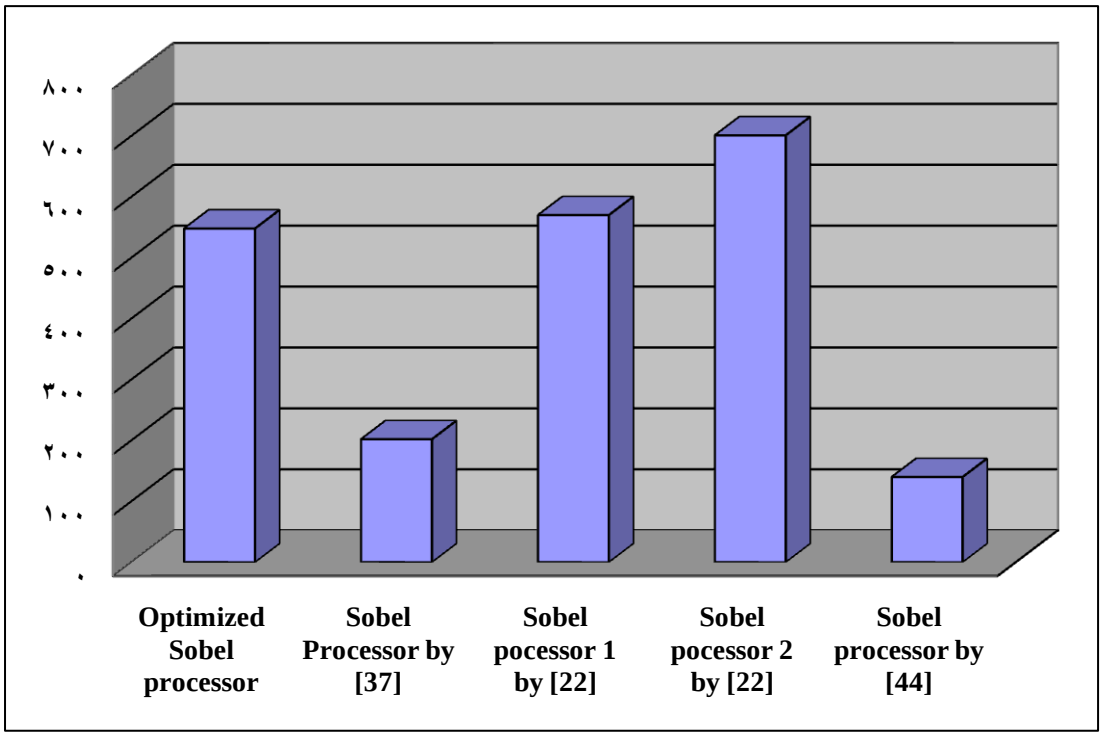


Figure 4.5 The logic utilization comparison

The comparison between the Sobel processors in terms of frequency and throughput is summarized in Table 4.14. The maximum performance of the optimized Sobel processor is compared with the other Sobel processor mentioned in the literature. The throughputs and frame rates of each processor are recalculated for a 640x480 frame size to match the frame size of the optimized Sobel processor.

For instance, the Sobel processor proposed by [37] operates in 134.756MHz and

is able to calculate the edges for an image size of 512x512 (262,144) pixels in 1.95ms. This results in a throughput of (262,144*1000/1.95=134.432x10^6) 134.432MP/s. For a frame size of 640x480 (307200) pixels, the processor can achieve a frame rate up to (134.432x10^6/307200=437.6) 437.6fps. Similarly, the Sobel processor 1 proposed by [22] is capable of processing 128x128 (16,384) pixels in 0.271ms with a frequency of 253MHz which results in a throughput of (16,384*1000/0.271=60.4x10^6) 60.4MP/s. The frame rate for a 640x480 frame size can be calculated as 60.4x10^6 /307200= 196.8fps. The same calculations are performed to calculate the frame rate for processor 2 proposed by [22] that can process 16,384 pixels in 0.226ms (72.5MP/s throughput) running with a 264MHz clock. Finally, the Sobel proposed by [42] reports a throughput of 4.4MP/s, which can produce (4.44x10^6 /30720) 14.45fps for a 640x480 frame size.

Table 4.14 Processors performance comparison

Processor	**Clock frequency (MHz)**	**Throughput (MP/s)**	**Frame rate (fps)**
The optimized Sobel processor	91.86	367.44	1196.02
Processor proposed by [37]	134.756	134.432	438.6
Processor 1 proposed by [22]	253	60.4	196.85
Processor 2 proposed by [22]	264	72.52	336.09
Processor proposed by [42]	40	4.44	14.45

As reported in Table 4.14, the optimized Sobel processor has a maximum clock frequency of 91.86 MHz which is considerably low compared to the other Sobel processors in the table except for the processor in [42].The low frequency can help decrease the power consumption of the processor and enables the use of the processor in mobile and embedded systems [29, 48-49] The optimized Sobel processor has the

highest throughput and the highest frame rate with 367.44 MP/s and 1196.02 fps, respectively. The high throughput is caused by the partial loop unrolling of the design to produce 4 Sobel edge detection pixels in one clock cycle. This increases the frame rate concurrently to 1196.02 fps. The high frame rate and high throughput exceeds the real-time systems constrains and the low frequency makes the optimized Sobel processor a better choice for embedded systems. The comparison results are illustrated by the Fig. 4.6.

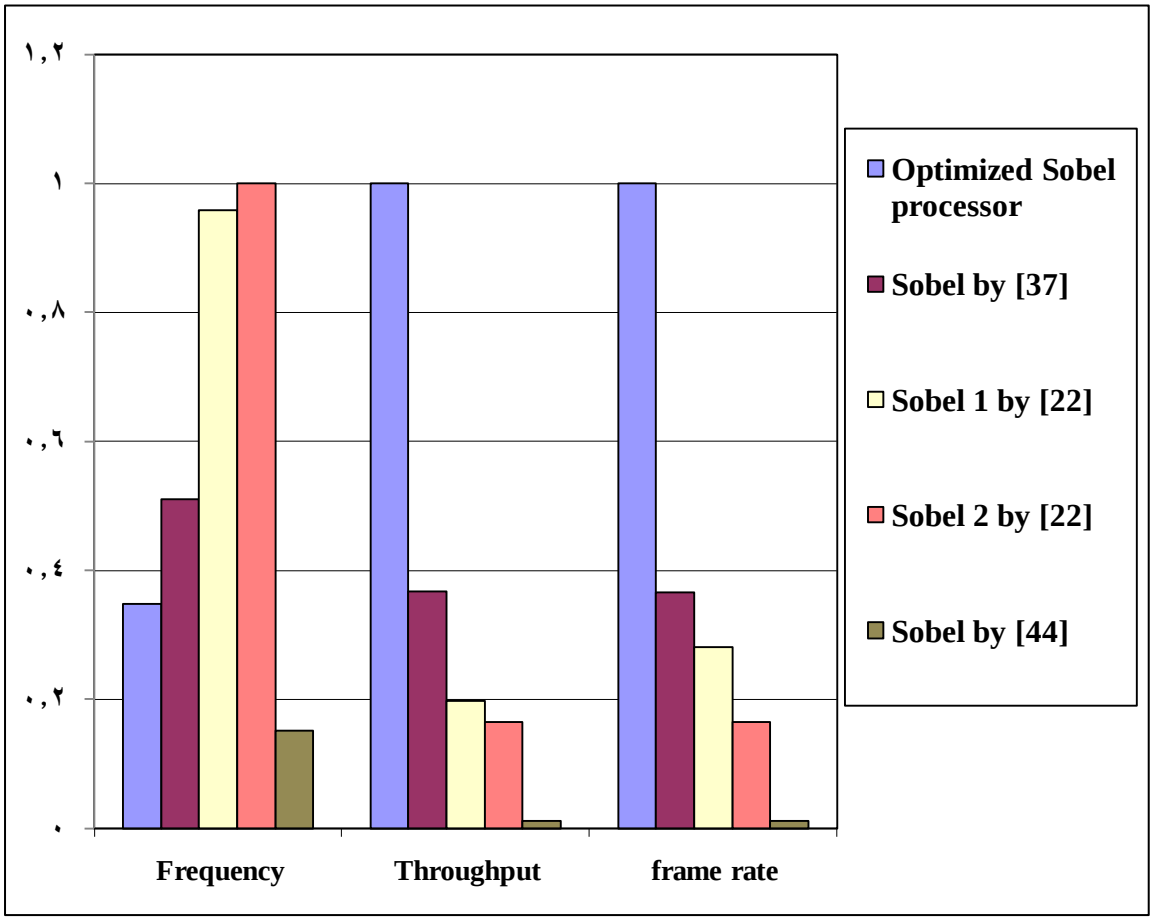


Figure 4.6 Performance comparisons between the optimized Sobel processor and literature processors using normalized values with respect to the maximum value.

4.4 Chapter Summary

This chapter presents an evaluation of the logic utilization of the optimized Sobel processor and its performance. A comparative analysis is carried out based on the processor performance versus four different Sobel processors in terms of throughput, frame rate, and frequency. The results show that the optimization performed on the proposed processor enables a massive increase in the processor throughput and hence frame rate even when the system operates with a relatively low clock frequency.

CHAPTER 5

CONCLUSION AND RECOMMENDATIONS

5.1 Conclusion

This thesis proposes a dedicated Sobel processor architecture on FPGA. The processor is optimized by applying several optimization techniques to reduce the logic utilization and increase the throughput. The processor is integrated with other controlling modules in order to work as a complete edge detection system on Altera Cyclone II DE2-70 FPGA. The performance of the processor was compared to other Sobel edge detection processors.

The proposed Sobel processor utilizes data reuse by taking advantage of the parallelism in the edge detection algorithm and the parallelism offered by the FPGA as the implementation platform. A single row of the Sobel processor utilizes the data reuse within one image line in the calculations of the horizontal gradients. The processor that consists of multiple rows utilizes the data reuse within the same image line and within multiple image lines to enable data reuse in both the calculations of the horizontal and the vertical gradients, respectively. The data reuse results in 54% in the number of subtractors in the processor block compared with other architecture.

The effect of the optimization carried out on the processor is significant in terms of logic utilization and performance. In terms of logic utilization, the optimization reduces the processor size, allowing more instances to be placed inside the processor block. Consequently, the processor gains performance with a factor equals to the number of the Sobel instances contained in the processor block. The comparative analysis shows that the Sobel processor is able to achieve a very high throughput even when operating with a low clock frequency compared to other Sobel processors discussed in the literature.

With the help of the specially designed line buffer, the proposed processor is able to produce several output pixels every clock cycle and hence satisfy real-time systems constraints. The low logic utilization of the Sobel processor along with its significantly high throughput and low operating frequency make it a good choice for mobile and embedded systems.

5.2 Future Work

In this thesis, a single row optimized Sobel processor is integrated in a complete edge detection system and evaluated for logic utilization and real-time performance. The processor performance can be further improved by the integration of noise suppression system within the processor block to improve the quality of the processor output. Moreover, the optimized Sobel processor can be incorporated with other higher-level image processing operations in order to build an integrated image processing system for a specific application.

REFERENCES

[1] B. A. Draper, *et al.*, "Accelerated image processing on FPGAs," *Image Processing, IEEE Transactions on*, vol. 12, pp. 1543-1551, 2003.

[2] N. Kehtarnavaz and M. Gamadia, *Real-time image and video processing: from research to reality*: Morgan & Claypool Publishers, 2006.

[3] J. G. Ackenhusen, *Real-time signal processing: design and implementation of signal processing systems*: Prentice Hall PTR, 1999.

[4] N. K. Guy, "Real-time image processing," in *Dictionary of film and digital photography* ed: PhotoNotes.org, 2002.

[5] E. Dougherty and P. Laplante, *Introduction to real-time imaging*: SPIE Optical Engineering Press, 1995.

[6] Altera. (March 2007). *Video and Image Processing Design Using FPGAs, White paper*. Available: http://www.altera.com/literature/wp/wp-video0306.pdf

[7] R. González and R. Woods, *Digital image processing*: Pearson/Prentice Hall, 2008.

[8] S. Dikbas, *et al.*, "Chrominance Edge Preserving Grayscale Transformation with Approximate First Principal Component for Color Edge Detection," in *IEEE International Conference on Image Processing, ICIP* 2007, pp. II - 261-II - 264.

[9] A. S. P. Ram, "Design of a recognition system for special Malaysian car plates using stroke analysis," Master of Engineering Electrical engineering, Universiti Teknologi Malaysia 2005.

[10] B. Chanda and D. Majumder, Digital image processing and analysis: Prentice-Hall of India, 2004.

[11] T.-H. H. Lee, "Edge Detection Analysis," Graduate Institute of Communication Engineering, National Taiwan University, Taipei, Taiwan, ROC2007.

[12] C. Torres-Huitzil and M. Arias-Estrada, "FPGA-based configurable systolic architecture for window-based image processing," *EURASIP J. Appl. Signal Process.*, vol. 2005, pp. 1024-1034, 2005.

[13] L. G. Roberts, "Machine Perception Of Three-Dimensional Solids," PhD, Department of electrical engineering, Massachusetts Institute of Technology, Cambridge, MA, 1963.

[14] S. C. Wu and J. M. S. Prewitt, "An Application of Pattern Recognition to Epithelial Tissues," in *Proceedings of the Annual Symposium on Computer Application in Medical Care*, 1978, pp. 15-25.

[15] J. M. S. Prewitt, "Object enhancement and extraction," *Picture Processing & Psychopictorics, Academic Press, New York,* pp. 75-149, 1970.

[16] I. Sobel and G. Feldman, "A 3x3 Isotropic Gradient Operator for Image Processing," *presented at a talk at the Stanford Artificial Project, unpublished but often cited, orig. in Pattern Classification and Scene Analysis, Duda,R. and Hart,P., John Wiley and Sons,'73, pp271-2,* 1968.

[17] N. Kanopoulos, *et al.*, "Design of an image edge detection filter using the Sobel operator," *IEEE Journal of Solid-State Circuits* vol. 23, pp. 358-367, 1988.

[18] J. Canny, "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, pp. 679-698, 1986.

[19] H. S. Neoh and A. Hazanchuk, "Adaptive Edge Detection for Real-Time Video Processing using FPGAs," presented at the The International Conference for Embedded Signal Processing Applications, GSPx 05, Santa Clara, CA, 2005.

[20]B. So and M. Hall, "Increasing the Applicability of Scalar Replacement," in *Compiler Construction*. vol. 2985, E. Duesterwald, Ed., ed: Springer Berlin / Heidelberg, 2004, pp. 2732-2732.

[21]J. M. P. Cardoso and P. C. Diniz, "Modeling Loop Unrolling: Approaches and Open Issues," in *Computer Systems: Architectures, Modeling, and Simulation*. vol. 3133, A. Pimentel and S. Vassiliadis, Eds., ed: Springer Berlin / Heidelberg, 2004, pp. 224-233.

[22]J. Park and P. C. Diniz, "Partial data reuse for windowing computations: performance modeling for FPGA implementations," presented at the Proceedings of the 3rd International Conference on Reconfigurable Computing: Architectures, Tols and Applications, Mangaratiba, Brazil, 2007.

[23]N. Baradaran, *et al.*, "Compiler reuse analysis for the mapping of data in FPGAs with RAM blocks," in *Proceedings of IEEE International Conference on Field-Programmable Technology*, 2004, pp. 145-152.

[24]Y. N. Srikant and P. Shankar, *The Compiler Design Handbook: Optimization and Machine Code Generation*: CRC PRESS, 2007.

[25]B. Buyukkurt, *et al.*, "Impact of Loop Unrolling on Area, Throughput and Clock Frequency in ROCCC: C to VHDL Compiler for FPGAs," in *Reconfigurable Computing: Architectures and Applications*. vol. 3985, K. Bertels, *et al.*, Eds., ed: Springer Berlin / Heidelberg, 2006, pp. 401-412.

[26]O. S. Dragomir, *et al.*, "Loop unrolling and shifting for reconfigurable architectures," in *International Conference on Field Programmable Logic and Applications, FPL*, 2008, pp. 167-172.

[27]O. S. Dragomir, *et al.*, "Optimal Loop Unrolling and Shifting for Reconfigurable Architectures," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, pp. 1-24, 2009.

[28] J. Mellor-Crummey and J. Garvin, "Optimizing Sparse Matrix–Vector Product Computations Using Unroll and Jam," *International Journal of High Performance Computing Applications*, vol. 18, pp. 225-236, 2004.

[29] S. Kilts, *Advanced FPGA Design: Architecture, Implementation, and Optimization*: John Wiley & Sons, 2007.

[30] F. Hannig, *et al.*, "Parallelization Approaches for Hardware Accelerators – Loop Unrolling Versus Loop Partitioning," in *Architecture of Computing Systems – ARCS 2009*. vol. 5455, M. Berekovic, *et al.*, Eds., ed: Springer Berlin / Heidelberg, 2009, pp. 16-27.

[31] K. Jung Sub, *et al.*, "An Automated Framework for Accelerating Numerical Algorithms on Reconfigurable Platforms Using Algorithmic/Architectural Optimization," *Computers, IEEE Transactions on*, vol. 58, pp. 1654-1667, 2009.

[32] U. Meyer, *et al.*, *Algorithms for memory hierarchies: advanced lectures*: Springer, 2003.

[33] N. Baradaran, *et al.*, "Extending the Applicability of Scalar Replacement to Multiple Induction Variables," in *Languages and Compilers for High Performance Computing*. vol. 3602, R. Eigenmann, *et al.*, Eds., ed: Springer Berlin / Heidelberg, 2005, pp. 455-469.

[34] P. M. Athanas and A. L. Abbott, "Real-time image processing on a custom computing platform," *Computer*, vol. 28, pp. 16-25, 1995.

[35] K. Benkrid, *et al.*, "High level programming for real time FPGA based video processing," in *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing. ICASSP '00*, 2000, pp. 3227-3230.

[36] A. Emmanuel, *et al.* (2009, the Sobel Edge detector using FPGA Project. Available: http://edge.kitiyo.com/

[37]T. A. Abbasi and M. U. Abbasi, "A proposed FPGA based architecture for Sobel edge detection operator," *Journal of active and passive electronic devices*, vol. 2, pp. 271–277, 2007.

[38]S. Hezel, *et al.*, "FPGA-based template matching using distance transforms," in *Proceedings of the10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002, pp. 89-97.

[39]S. Chivapreecha and K. Dejhan, "Hadware implementation of Sobel-Edge detection Distributed Arithmatic Digital Filter " presented at the 25th ACRS, Asian Conference on Remote Sensing, Chiang Mai, Thailand, 2004.

[40]N. K. Ratha, *et al.*, "Convolution on Splash 2," in *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, 1995, pp. 204-213.

[41]R. W. Means and H. J. Sklar, "Systolic array image processing system," United States Patent, 1989.

[42]L. Xue, *et al.*, "FPGA based Sobel algorithm as vehicle edge detector in VCAS," in *Proceedings of the 2003 International Conference on Neural Networks and Signal Processing*, 2003, pp. 1139-1142 Vol.2.

[43]A. Y. Zomaya, *et al.*, "Genetic scheduling for parallel processor systems: comparative studies and performance issues," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, pp. 795-812, 1999.

[44]Altera, *Cyclone II Device Handbook* vol. 1. San Jose, CA: Altera Corporation, 2008.

[45]C. E. Cummings, "Simulation and Synthesis Techniques for Asynchronous FIFO Design," presented at the SNUG 2002, Synopsys Users Group Conference, San Jose, CA, 2002.

[46]Terasic. (2009, Altera DE2-70 board user manual. Available: http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=53&No=226&PartNo=4

[47] Terasic. (2009, D5M 5 Mega Pixel Digital Camera Package. Available: http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=68&No=281&PartNo=3

[48] R. Woods and J. McAllister, *FPGA-based implementation of signal processing systems*: John Wiley & Sons, 2008.

[49] H. Saar. (2006, *Minimizing FPGA power consumption*. Available: http://www.eetimes.com/design/automotive-design/4004640/Minimizing-FPGA-power-consumption

[50] J. Voelmle, "Investigation of Altera DE2 Development and Education Board," Florida Gulf Coast University2009.

[51] Xilinx. (1998, XC3000 Series Field Programmable Gate Arrays (XC3000A/L, XC3100A/L) Product discribtion. Available: http://www.xilinx.com/support/documentation/data_sheets/3000.pdf

APPENDIX

Publications:

- Zahraa Elhassan M. Osman, Fawnizu Azmadi Hussin, Noohul Basheer Zain Ali," *Optimization of Processor Architecture for Image Edge Detection filter,"* UKSIM 12th International Conference on Computer Modeling and Simulation, Cambridge, United Kingdom, March 2010, pp. 648-652.

- Zahraa Elhassan M. Osman, Fawnizu Azmadi Hussin, Noohul Basheer Zain Ali," *Hardware Implementation of an Optimized Processor Architecture for SOBEL Image Edge Detection Operator*," 3rd International Conference of Intelligent and Advanced System, (ICIAS), Kuala Lumpur, Malaysia, June 2010, in press.