Efficient FPGA Architectures for Separable Filters and Logarithmic Multipliers and Automation of Fish Feature Extraction Using Gabor Filters

Arjun Kumar Joginipelly
arjunrao143@gmail.com

Efficient FPGA Architectures for Separable Filters and Logarithmic Multipliers and Automation of Fish Feature Extraction using Gabor Filters

A Dissertation

Submitted to the Graduate Faculty of the
University of New Orleans
in the partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Engineering and Applied Sciences
Electrical Engineering

by

Arjun Kumar Joginipelly
M.S. University of New Orleans, 2010

Aug 15 2014

Dedication

I dedicate this work to my parents, Joginipelly Raj Gopal Rao and J. Prabha Rani; to my grandparents Potlapally Bapu Rao and P. Bharatamma; to my uncles, P. Govind Rao and P. Mohan Rao; to my sister and brother, M. Sri Laxmi and J. Anil Kumar. The blessings, unconditional love, and constant support you all gave were always with me and encouraged me in stepping forward in life.

Acknowledgments

First and above all, I praise God for giving me this opportunity and granting me the capability to proceed successfully in completing my dissertation overseas. This dissertation would not have been possible without the guidance and support of several individuals who in one way or other contributed and extended their valuable assistance in the completion of this study.

I would like to express enormous gratitude to Dr. Dimitrios Charalampidis, my advisor, for his help, suggestions, and guidance throughout the course of my dissertation. I appreciate his direction, supervision of my work, and his patience which helped me to progress in the right direction. I would like to thank Dr. George Ioup, Dr. Juliette Ioup, Dr. Huimin Chen and Dr. Vesselin Jilkov, for their willingness to serve as members of my dissertation committee.

Most importantly, I would like to thank my parents and grandparents, for teaching me that responsibility in life is to learn, to be happy, and to know and understand myself; only then I could understand and help others. I am indebted to them for encouraging me in all of my pursuits and inspiring me to follow my dreams. I always knew that you believed in me and wanted the best for me.

I acknowledge my friend Mr. Rajesh Chary, for his valuable help and support throughout my dissertation. His patience, assistance, and constructive criticism helped me in shaping my personal and academic life for complex situations. I would like to thank Patricia Robbert, for giving me the opportunity to work as a teaching assistant in Physics Department. This helped me in improving speaking and presentation abilities, and support myself financially for continuing my studies at University of New Orleans.

I would like to thank Jeevan Allagar, for his time and patience in helping me to understand the importance of moving forward in life taking both success and failures along side with me. Finally, I would like to thank Bing & Michael Simpson, Janu, Nag Bhai, and each and everyone, for listening, offering me advice and supporting me throughout my dissertation.

List of Acronyms

FPGA – Field Programmable Gate Arrays

NOAA – National Oceanic and Atmospheric Administration.

ASIC – Application Specific Integrated Circuit

RAM – Random Access Memory

HDL – Hardware Description Language

VHDL – Very High Speed Integrated Circuits HDL

DSP – Digital Signal Processor

2D – Two Dimensional

EMB – External Memory Bandwidth

CC – Clock Cycle

OCDB – On-chip Data Buffers

FB – Full Buffering

PB – Partial Buffering

MWPB – Multi-window Partial Buffering

LUT – Look-Up-Table

ICR – Intermediate Convolution Result

FCR – Final Convolution Result

FIFO – First-In-First-Out

BRAM – Block RAM

SDRAM – Synchronous Dynamic RAM

DDRAM – Double Data Rate RAM

MUX – Multiplexer

DEMUX – Demultiplexer

CSC – Central Selection Controller

VR – Vertical Results

HR – Horizontal Results

LNS – Logarithmic Number System

OD – Operand Decomposition

MSB – Most Significant Bit

MSE – Mean Square Error

IM – Iterative Mitchell

NCE – Non-Controlled Environments

SEFSC – Southeast Fisheries Science Center

EM – Epinephelus morio or red grouper

OC – Ocyurus chrysurus or yellow tail snapper

ABC – Allowable Biological Catch

GF – Gabor Filters

MA – Moving Average

Table of Contents

List of Figures

List of Tables

Abstract

Convolution and multiplication operations in the filtering process can be optimized by minimizing the resource utilization using Field Programmable Gate Arrays (FPGA) and separable filter kernels. An FPGA architecture for separable convolution is proposed to achieve reduction of on-chip resource utilization and external memory bandwidth for a given processing rate of the convolution unit.

Multiplication in integer number system can be optimized in terms of resources, operation time and power consumption by converting to logarithmic domain. To achieve this, a method altering the filter weights is proposed and implemented for error reduction. The results obtained depict significant error reduction when compared to existing methods, thereby optimizing the multiplication in terms of the above mentioned metrics.

Underwater video and still images are used by many programs within National Oceanic Atmospheric and Administration (NOAA) fisheries with the objective of identifying, classifying and quantifying living marine resources. They use underwater cameras to get video recording data for manual analysis. This process of manual analysis is labour intensive, time consuming and error prone. An efficient solution for this problem is proposed which uses Gabor filters for feature extraction. The proposed method is implemented to identify two species of fish namely Epinephelus morio and Ocyurus chrysurus. The results show higher rate of detection with minimal rate of false alarms.

Keywords

Separable Convolution, Image Processing, Field Programmable Gate Array (FPGA), Logarithmic Multipliers, Gabor Filters, Automated Fish Classification and Feature Extraction.

Chapter 1

Introduction

1.1 Motivation

Embedded systems play an important role in digital image, video and signal processing with applications in wide areas such as digital cameras, traffic light controller systems [1], lane detection [2], medical devices, etc. In order to optimize the system performance to meet the ever increasing demand for efficient system performance, major metrics in terms of cost, resources, operation time, power consumption, resolution, and robustness are to be improved. Image processing systems which constantly demand for real-time processing power require new design methodologies and hardware accelerator architectures for hardware implementation.

In earlier times, image processing systems were mostly built with Application Specific Integrated Circuits (ASICs) which are not re-programmable (or re-configurable). A malfunction in one ASIC often results in a complete replacement of the faulty component. The ASIC's lack of flexibility to be reprogrammed is promoting their counterpart, namely the Field Programmable Gate Array (FPGA) chips.

FPGA's generally consist of a logic block-based system, which usually includes look-up-tables (LUT), flip-flops and some amount of random access memory (RAM), all wired together using a vast array of interconnects. All of the logic in an FPGA can be reconfigured with a different design as often as the designer likes. FPGAs can be configured by hardware engineers using a Hardware Design Language (HDL). The two principal languages which allows designers to design at various levels of abstraction are Verilog HDL (Verilog) and Very High Speed Integrated Circuits HDL (VHDL).

FPGAs can be developed to implement hardware design techniques such as parallelism and pipelining, which is not possible in dedicated Digital Signal Processing (DSP) designs. DSPs are a class of hardware devices that fall somewhere between an ASIC and a PC in terms of the performance and the design complexity. ASICs were traditionally preferred over FPGAs because of their speed, lower power consumption, and higher functionality. However, the improvement on FPGA technology in recent years closed this gap. ASIC design methods can also be used for FPGA design, facilitating gate level implementations, thereby decreasing development time and time-to-market. However, engineers usually use a hardware language, which allows for a design methodology similar to software design. Maintenance can be performed when an error is found in the implemented design, since the FPGA fabric can always be reconfigured. This software view of hardware design allows for a lower overall support requirements, lower cost, and design abstraction. The key advantages of FPGAs when compared to DSP implementations include performance, integration and customization using parallel and pipeline design techniques. Due to the support of parallelism, FPGAs may be able to achieve huge gains in performance compared to DSP implementations.

A thorough study on the capabilities of the FPGAs provided the motivation to utilize its features for implementing image processing algorithms [3] with an aim of minimizing time-to-market cost, enabling rapid prototyping of complex algorithms and finally simplifying the tedious process of debugging and verification. Some works of image processing applications using FPGAs have been reported such as edge detection [2], enhancement [4], and smoothing [5], convolution form the basic component. For instance, for image size of 256×256 with number of frames = 25/sec and kernel size of 3×3, real time image convolution requires 589,824 multiplications and 524,288 additions. Furthermore, as the image resolution and refresh rate increases, computational requirements of the convolution unit significantly increases. Hence, due to high performance requirements, FPGAs are an ideal choice for implementation of real time image processing algorithms. Given the importance of digital image processing and the significance of their implementations on hardware to achieve better performance, this dissertation addresses efficient architectures of separable convolution and logarithmic multiplication on FPGA using VHDL language.

"A picture is worth a thousand words." As human beings, we can extract information from a picture based on what we see and our background knowledge. Based on this analogy, we can train computers by building models to extract efficient visual features [6] from an image or video sequence. Hence, feature extraction is the fundamental step in any automated image analyzing system which is widely used in one of the areas of identifying and quantifying living marine resources.

Species detection is necessary to explore new species and their richness in any ecology. Many fisheries use manual accumulation and analyze large amounts of data obtained using underwater cameras for recognition and classification of species. This tedious process is easily prone to errors as it involves manual analysis. This provided the motivation to the proposed method on species-specific feature extraction using Gabor filters for extracting two species of fish namely Epinephelus Morio, and Ocyurus Chrysurus. The proposed method gave the desired results with a significant identification rate and negligible false alarms.

1.2 Dissertation Outline

The rest of the dissertation is organized as follows: Each chapter is allocated for a specific methodology explaining the contribution of dissertation work in improving on the existing methodologies.

Chapter 2 covers the convolution operation in detail and its functioning in filters and explains the need for optimizing in terms of resource utilization and external memory bandwidth. It then follows up with a thorough review on various existing FPGA separable convolution methods. The proposed FPGA architecture with separable filter kernels, its implementation, and the results are then presented.

Chapter 3 covers the multiplication operation in filters addressing the questions such as how its optimization can improve the filter performance in terms of metrics such as power consumption, operation time and resource utilization. One of the ways to achieve this is to convert integer multiplication to logarithmic multiplication. Various existing methods on logarithmic multipliers are then reviewed following up with the proposed method. The main objective of this method is to reduce the multiplication error by altering the filter weights.

Chapter 4 covers the proposed method to improve on manual fish feature extraction process currently in use by National Oceanic Atmospheric Administration (NOAA) fisheries group. An in depth need for the automation of the feature extraction and species classification process is explained. One of the solution is to use Gabor filters for feature extraction. The proposed algorithm using Gabor filter is implemented and the results obtained are then presented which show prominent success rate.

Chapter 5 gives the overall conclusions drawn from all the proposed methods and discusses on how they can be improved further in future.

1.3 Dissertation Contributions

In this dissertation, issues related to convolution and multiplication on FPGA hardware are resolved in an efficient way. With a major concern given to address external memory bandwidth, on-chip data buffers and power consumption which are common to most imaging applications. Architectural considerations as well as design methodology constitute the main scope of the dissertation research work. Efficient FPGA architectures are presented for separable convolution which provide a good balance between on-chip resource utilization and external memory bandwidth. An efficient logarithmic multiplier based on Mitchell [7] is proposed and implemented on FPGA. Automation of fish feature extraction using Gabor filters is proposed as an efficient solution to manual fish analysis problem. The list of publications which are accepted or currently in the process of submission from this dissertation research are as follows:

1. Arjun Kumar Joginipelly, Dimitrios Charalampidis, "Efficient Separable Convolution Using FPGA's", currently in the process of submission. [Chapter 2].
2. Arjun Kumar Joginipelly, Dimitrios Charalampidis, "Efficient Mitchell-based Logarithmic Multiplier", currently in the process of submission. [Chapter 3]
3. Arjun Kumar Joginipelly, Dimitrios Charalampidis, George Ioup, Juliette Ioup, Charles H Thompson, "Species-Specific Fish Feature Extraction Using Gabor Filters", proceedings of 66th Gulf and Carribbean Fisheries Institute, Corpus Christi, Texas, USA, Nov 2013. [Chapter 4]

Chapter 2

Efficient FPGA Implementation of Separable Convolution Architectures

2.1 Abstract

Two-dimensional (2D) convolution is an essential component of several image and video processing applications. In general, the convolution operation is computationally expensive. Field Programmable Gate Array (FPGA) architectures have been used to mitigate this problem owing to their parallel processing capabilities. Using separable filter kernels can further improve the convolution operation both in terms of resource utilization and speed. However, although several 2D convolution implementations have been presented in the literature, research on separable convolution using FPGA is limited. In this chapter, a new separable FPGA-based convolution architecture is proposed. The goal is to reduce both on-chip resource utilization and external memory bandwidth for a given processing rate of the convolution unit. Comparisons with existing separable convolution methods demonstrate the achievement of the stated goal.

2.2 Introduction

Two dimensional (2D) convolution is widely used in image processing applications such as edge detection [2], enhancement [4], and smoothing [5]. Spatial domain convolution using a kernel of size $K \times K$ requires K^2 multiplications and $(K^2 - 1)$ additions per pixel. The convolution output at image location (x, y) can be computed as the sum of products between kernel weights and corresponding pixel values located within the $K \times K$ image window centered at (x, y).

Real-time hardware-based convolvers require a large amount of resources including adder circuits, multipliers or multiplier-equivalent circuits, and internal and/or external memory. Major considerations include the external memory bandwidth (EMB), namely the number of pixels read from external memory or device per clock cycle (cc), the amount of on-chip data buffers (OCDB) where input or processed data is stored, and the amount of multiplier-related resources. Several schemes [8],[9],[10],[11] proposed in the literature address EMB and OCDB utilization.

In [8], a full buffering (FB) scheme was used for implementing 2D convolution. Delay line buffers were used to hold $(K - 1)$ rows of the input image. A partial buffering (PB) scheme was proposed in [9] to eliminate the line buffers by storing only small image portions in internal memory. However, achieving this goal resulted in an increased EMB. A multi-window PB (MWPB) scheme was proposed in [10] to overcome the problems in FB and PB schemes.

Similarly to PB, the MWPB scheme aimed at storing only small portions of the image data in internal memory. The main objective, however, was to reuse the common data shared by consecutive windows to the full extent possible. Therefore, the same image data did not have to be read K times as in the case of PB. This concept is illustrated in Fig. 2.1. Since K windows of size $K \times K$ ($W_k, k = 1, ..., K$) are available in $(2K - 1)$ consecutive image rows, K filter outputs can be produced by processing $(2K-1)$ rows. Thus, each pixel is read only $(2K-1)/K$ times, or practically, two times.

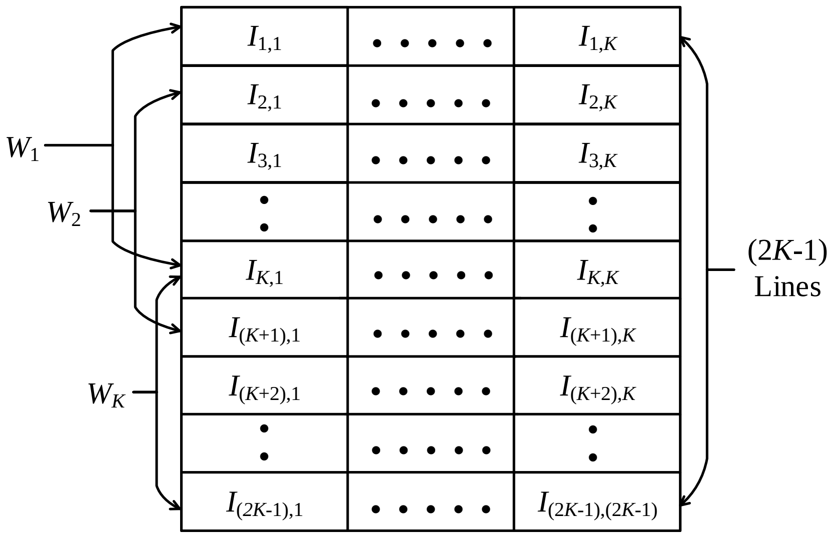

Figure 2.1: Illustration of available common rows for consecutive convolution windows W_1, W_2 and W_K.

The resources required for 2D convolution can be reduced by employing filters with separable kernels. A $K \times K$ separable kernel can be decomposed into a horizontal $1 \times K$ kernel and a vertical $K \times 1$ kernel, so that only $2K$ multiplications and $(2K - 2)$ additions are required per pixel. Although several hardware-based 2D convolution implementations have been investigated, there is limited work on separable convolution using FPGA or VLSI architectures. In [5], steerable filters [12] were implemented using a separable filter component. In [13], an image scaling architecture using separable filters was proposed. In [14], a separable implementation of Voltera filters was presented. In [15], singular value decomposition was used to approximate 2D filters as a cascade of separable and non-separable filters employing, respectively, embedded multipliers and reduced complexity LUT-based multipliers. In [16], FB-based separable convolution architectures and overlapping kernel techniques were investigated. Transform-based techniques were proposed [17] to perform 2D convolution in a separable manner.

Most existing works, including the ones mentioned above, mainly employ FB separable convolution schemes. Other works do not focus on separable convolution but only use it as one of the system components. Although FB schemes offer advantages with respect to non-separable techniques [16], they still require a large amount of resources due to line buffering. This chapter proposes an architecture that reduces both EMB and on-chip resource utilization for a given processing rate of the convolution unit. Section 2.4 presents the technical details of the proposed scheme.

Although it is not the objective of this work, employing LUT-based multiplier-equivalent circuits may further reduce resource utilization. Examples include $log2$ and inverse $log2$ modules [18], shift and accumulation block modules [19], optimized codes such as distributed arithmetic, output product coding, canonic signed digit coding, and binary sub-expression coding [20],[21], and common sub-expression elimination methods to replace constant multiplications with a network of adders and shifters [22]. Moreover, although the proposed work deals with linear convolution, cyclic convolution techniques have also been investigated in [23],[24].

This chapter is organized as follows: Section 2.3 discusses two separable convolution architectures. The first has been previously introduced in the literature [5],[14],[15],[25] and is based on the FB scheme. The second was implemented by the authors by modifying the PB scheme originally designed for non-separable convolution [9]. Section 2.4 introduces the proposed FPGA architecture. Section 2.5 presents performance analysis, and comparisons among the three architectures. Concluding remarks are presented in section 2.6.

2.3 Review of Separable Convolution Methods

This section discusses separable convolution architectures based on the FB and PB schemes. The FB separable convolution scheme has been previously introduced in the literature [5],[15],[14],[25]. The PB separable convolution architecture is designed in this work as a straightforward modification of the non-separable PB-based scheme. In what follows, parameter P describes the image pixel depth, which for grayscale images is equal to 8 bits. Moreover, the expression "shift register array of size R" implies a total of P shift registers, each consisting of R flip flops.

2.3.1 FB Scheme for Separable Convolution

In [5],[25], an FB scheme was used to implement separable convolution. A similar architecture was presented in [14],[15]. The FB separable convolution scheme is depicted in Fig. 2.2. In summary, input pixels are read from external memory, one at a time, in a row-wise manner. Pixels are pushed into internal line buffers until $(K - 1)$ image rows, each consisting of N pixels, are read. Each time a new pixel from the K-th row is read and stored in FIFO, a total of K pixels, forming a vertical $K \times 1$ window, are available and stored in K internal buffers. The products between the K pixels and the corresponding K vertical filter weights are obtained and summed together to produce the vertical intermediate convolution result (ICR). As pixels from the K-th row continue to be read, consecutive ICRs are stored in a shift register array of size K. The K ICRs are multiplied with the corresponding K horizontal filter weights and the results are summed together to obtain the final convolution result (FCR). The above process is repeated for all image pixels. Equivalently, the horizontal operation may be performed first followed by the vertical operation.

In a pipelined version of the FB scheme, one pixel is read and one FCR is produced per clock cycle, while a total of $2K$ multipliers should be available. Alternatively, in a non-pipelined version, the horizontal and vertical filters may share the same K multipliers. In this case, multiplexing is needed in order to select the appropriate inputs to the multipliers, namely vertical versus horizontal filter weights, and input data versus ICRs. In this case, one input pixel is read and one FCR is produced per two clock cycles. The advantage of the FB scheme is that

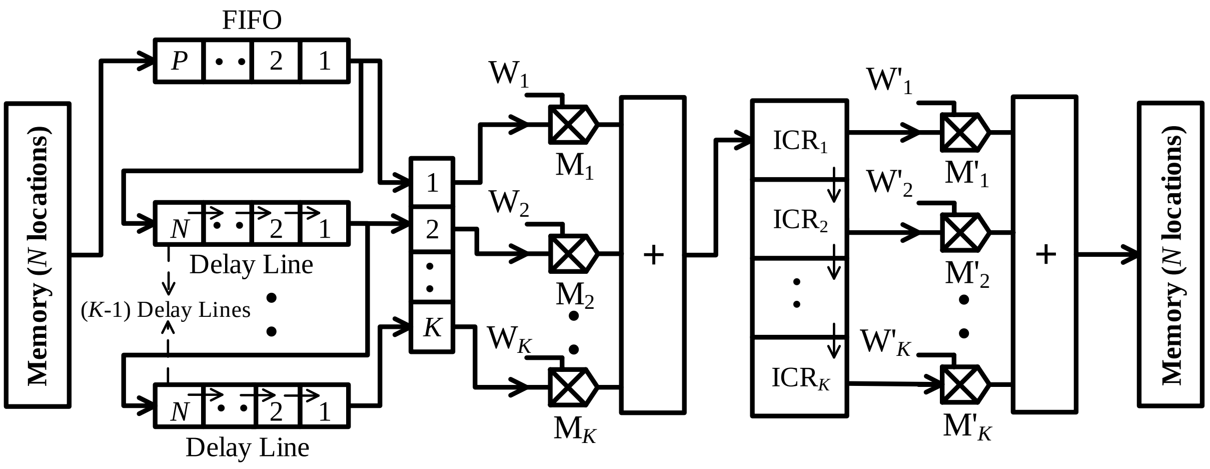

Figure 2.2: FB scheme for separable convolution

each pixel is read only once from external memory. Therefore, EMB equals only 1 or 0.5 pixels/cc using, respectively, the pipelined and non-pipelined versions of the scheme. The major disadvantage of this scheme is that shift register arrays of total size $[(K-1)N+K+1)]$ are used for buffering, which accounts for a substantial percentage of FPGA resources. The problem becomes significant for large input image and kernel sizes.

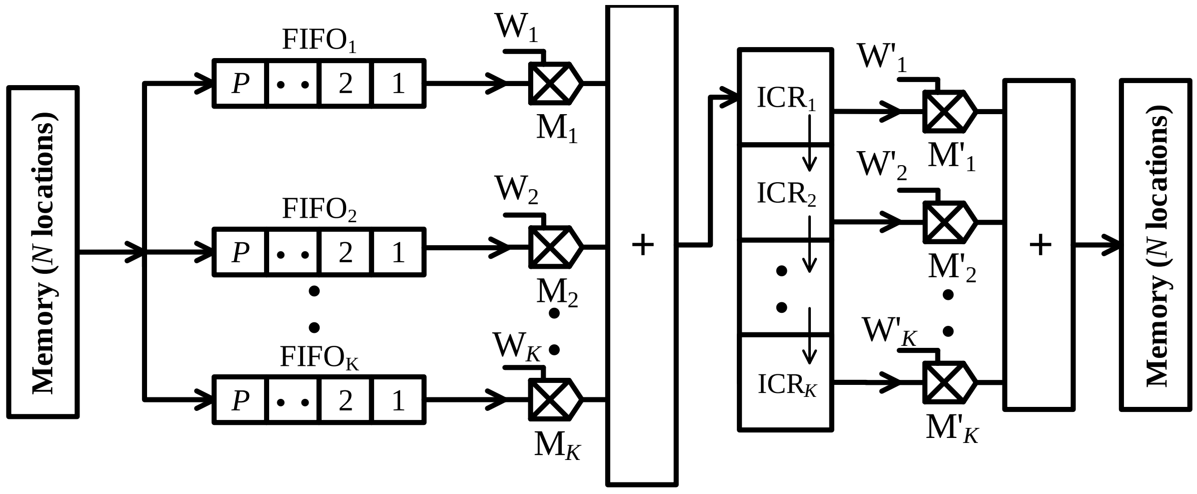

Figure 2.3: PB scheme for separable convolution

2.3.2 PB Scheme for Separable Convolution

The original PB scheme was developed for non-separable kernels [9] and is not presented here. Instead, a separable version of the PB scheme has been designed in this work, for the purpose of comparing with the FB separable convolution and the proposed schemes. In the separable version of the PB scheme, presented in Fig. 2.3, K pixels located in a window of size $K \times 1$ are read from external memory simultaneously and pushed in first-in-first-out (FIFO) blocks. The FIFO data are multiplied with the corresponding K vertical filter weights to produce the ICR. Similarly to the FB scheme, once K consecutive ICRs become available, they are multiplied with the corresponding K horizontal filter weights. The products are summed together to obtain the FCR. The process is repeated for all image pixels. Similarly to the FB scheme, a pipelined and a non-pipelined version may be considered.

The advantage of the PB separable convolution architecture is that only $2K$ data points has to be stored in internal buffers and shift register arrays. Therefore, the internal storage-related resources are reduced dramatically with respect to the FB scheme. However, each pixel is read K times from external memory resulting in a high EMB. The EMB equals K or $K/2$ pixels/cc using, respectively, the pipelined and non-pipelined versions of the scheme.

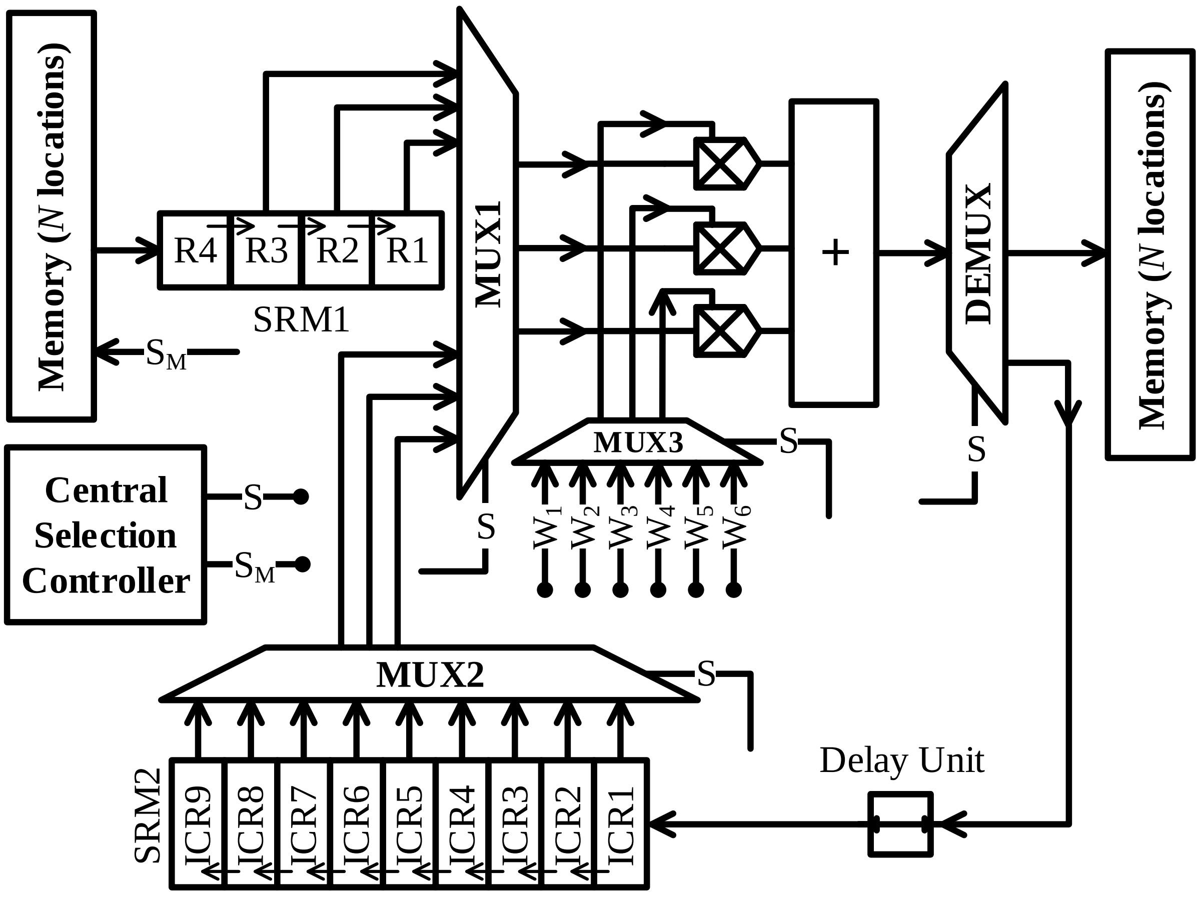

Figure 2.4: Proposed FPGA architecture for a kernel of size 3×3

2.4 Proposed Separable Convolution Scheme and Implementation

The objective of the proposed separable convolution technique is to reduce EMB, by reusing common data shared by consecutive processing windows, and OCDB, by eliminating line buffers. In this section, each module in the proposed separable convolution scheme is described in detail with reference to the FPGA implementation. The input image pixels are read from block RAM (BRAM), although other external memory elements such as synchronous dynamic RAM (SDRAM) or double date rate RAM (DDRAM) may also be used. Moreover, the implementation described next is what was referred earlier as the non-pipelined version, in which the same set of multipliers is shared by both vertical and horizontal 1D filters. The FPGA architecture of the proposed scheme with detailed schematic considering a kernel of size 3×3 is shown in Fig. 2.4.

2.4.1 Shift Register Modules

As described in the introduction and Fig. 2.1, EMB is reduced if information from $(2K - 1)$ rows is processed at a time. For this reason, pixels located in a $(2K - 1) \times 1$ window, $W_A^{i,j}$, are read one at a time before moving on to the next window. The superscript i, j indicates, respectively, the row and column position of the top pixel in $W_A^{i,j}$. In particular, the next window is $W_A^{i,j+1}$, if $j = 1, .., N - 1$ or $W_A^{(i+K,1)}$ if $j = N$, in which case the current window $W_A^{i,N}$ is located at the rightmost image column. The order in which pixels are read from the input image is presented in Fig. 2.5. In the proposed architecture, it is not necessary to store all $2K - 1$ rows of $W_A^{i,j}$ as in the case of the MWPB scheme. Only K pixels located within a $K \times 1$ subwindow, namely W_k, are needed at a time to produce a vertical convolution result. An additional pixel is read to be made available for processing the next subwindow W_{k+1}. Module SRM1, which consists of a shift register array of size $(K + 1)$, is used to hold and shift the input image pixels.

Each $(2K - 1) \times 1$ window provides K ICRs corresponding to K image subwindows $W_k, k = 1, ..., K$ (see Fig. 2.1) within the same window $W_A^{i,j}$. Once K consecutive $W_A^{i,j}$ windows are processed, a total of K^2 ICRs are produced. These results are stored in a second shift register array of size K^2 denoted as SRM2. For every new $K \times 1$ image window, K new ICRs are shifted in SRM2, one at a time, while the K oldest ICRs are shifted out of SRM2.

As opposed to the scan method presented here, [26] requires that a $(2K - 1) \times (2K - 1)$ image block is stored at a time, and mainly concentrates on how a single window is processed. However, it does not describe how processing should transition from one window to the next. Due to this uncertainty, [26] is not compared to the proposed work.

2.4.2 Multiplexer and Demultiplexer Modules

The proposed architecture utilizes three multiplexers (MUX1, MUX2, MUX3) and one demultiplexer (DEMUX) modules. In particular, MUX1 selects pixels from either SRM1 or SRM2, depending on the selection signal S generated from the central selection controller (CSC). If $S = HIGH$ then MUX1 selects pixels from SRM1. In this case, the multiplier and adder modules generate ICRs which are stored temporarily to SRM2 through DEMUX. When SRM2 is completely filled, S is set to LOW by the CSC, and MUX2 assigns K out of K^2 ICRs to MUX1. In this case, the multiplier and adder modules generate the FCRs. DEMUX sends the FCRs in another BRAM or other memory. It should be mentioned that ICRs are fed back to SRM2 from DEMUX with a 3-clock cycle delay. The multiplier and adder stages require 2 clock cycles to generate their output and another clock cycle is used to sync the data with SRM2. The delay produced is constant and does not depend on the input image or filter size.

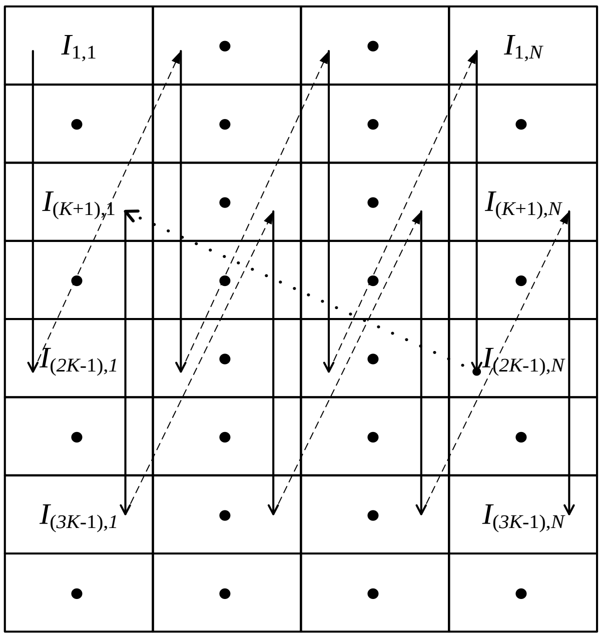

Figure 2.5: Order in which the image pixels are read

Depending on the selection signal S generated by the CSC, the role of MUX3 is to assign the appropriate set of weights to the multipliers corresponding to vertical or horizontal filter. In the more general separable filter case, such as in the case of a Gabor filter defined as $g(x, y) = Aexp\{-(x^2 + y^2)/2\sigma^2\}cos(w_o x)$, the vertical and horizontal filter weights differ. Of course, if the filter weights for the horizontal and vertical direction are identical, MUX3 can be completely eliminated.

2.4.3 Multipliers and Adder

In this design, embedded multipliers are employed using the Xilinx multiplier IP core [27]. Alternatively, LUT-based multiplication reduction methods, such as the ones discussed in the introduction section, could be employed. A total of K multipliers are required in the proposed implementation. If the kernel is not squared but rectangular of size $(K_y \times K_x)$ with $K_y > K_x$, then K_y multipliers are required.

Assuming that the weights are quantized to P bits, the result of each multiplication will consist of $2P$-bits. The final result obtained from the adder is scaled to ensure that the intensity value of the output pixel does not exceed P bits. This can be achieved by retaining only the P most significant bits generated by the adder.

2.4.4 Central Selection Controller

Signal S, which is generated by the CSC, is sent to MUX1, MUX2, MUX3 and DEMUX, while signal S_M is sent to memory. A counter, C, is used to set the value of signals S and S_M at each clock cycle. More specifically, $S = HIGH$ for $C = 0, ..., (K - 1)$, and $S = LOW$ for $C = K, ..., (2K - 1)$. Similarly, $S_M = HIGH$ for $C = 0, ..., (2K - 2)$, and $S_M = LOW$ for $C = (2K)$. Each window, $W_A^{i,j}$, consists of $(2K - 1)$ pixels, yet $(2K)$ clock cycles are needed to produce K convolution results. Thus, an image pixel is not read once every K clock cycles, when $S_M = LOW$.

The timing diagrams of signals S and S_M for a separable filter of size 3×3 are shown in Fig. 2.6. As described in Fig. 2.6, vertical results (VR) are produced during $S = HIGH$. During the period when $S = LOW$, horizontal results (HR) are produced. At the same time,

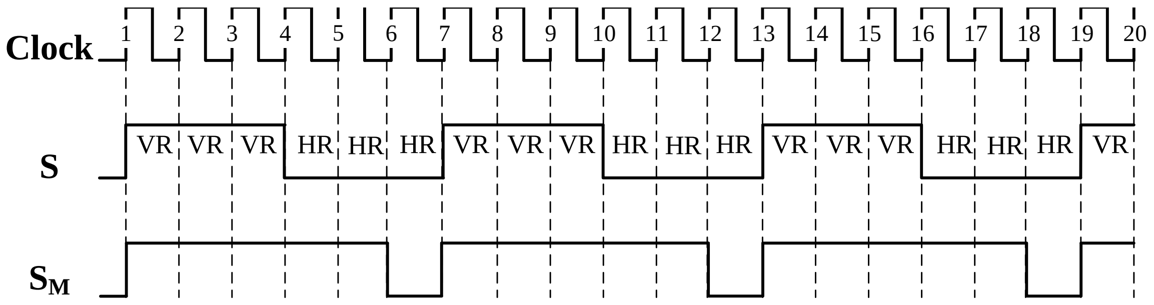

Figure 2.6: Central selection controller timing diagram

the architecture has the opportunity to read pixels from the next window $W_A^{i,j+1}$ or $W_A^{i+K,j}$ and populate SRM1 with new data.

2.5 Performance Analysis

The proposed technique is implemented on Xilinx Genesys Virtex 5 LX50T [28] FPGA board in VHDL language using the ISE 13.4 software. To the best of the authors' knowledge, only the FB scheme [5],[14],[15],[25] has been presented in the literature for separable convolution. In order to make additional comparisons, the authors designed a PB scheme for separable kernels, as a direct extension of the PB scheme for non-separable kernels found in [9]. The proposed scheme is compared with the FB scheme and the separable version of the PB scheme in terms of resource utilization (number of flip flops and LUT-flip flop pairs), EMB (pixels/cc) and processing rate (output pixels/cc). The non-pipelined versions of all schemes, namely FB, PB, and proposed, were implemented in this work. As a reminder, the term "non-pipelined" is used in the sense that the multipliers are reused for vertical and horizontal convolution. For all implementations, BRAM is used to store the input and output images. All images used were of size 128×128.

Method	External Memory Bandwidth (pixels/cc)	Embedded Multipliers	Processing Rate (output pixels/cc)
FB	0.5	K	0.5
PB	$\lceil K/2 \rceil$	K	0.5
Proposed	$(2K-1)/(2K)$	K	0.5

Table 2.1: Characteristics of the three schemes for a $K \times K$ filter

The characteristics of FB, PB and proposed schemes are summarized in Table 2.1. The EMB required for the FB scheme is 1 pixel per 2 clock cycles or 0.5 pixels/cc. The upper bound of the EMB for the proposed scheme is 1 pixel/cc. Although the EMB upper bound for the proposed scheme is almost twice as large as the EMB for the FB scheme, it is still considered quite small, and it is independent of the filter kernel size. On the other hand, the PB scheme has a significantly higher EMB, which increases proportionally to the kernel size.

The dynamic power analysis of FB, PB, and proposed schemes is summarized in Fig. 2.7 for a kernel of size 3×3 and clock frequencies varying from 100MHZ to 500MHZ. It is evident from Fig. 2.7 that the proposed architecture consumes less power compared to the other two architectures, especially as the clock frequency increases. The larger power consumption for FB and PB is partially attributed, respectively, to the operation of a large number of line buffers, and to the multiple pixel readings/cc.

Comparisons between the three schemes in terms of flip flop count are summarized in Table 2.2 for various filter sizes. It can be observed that for smaller size filters, the flip flop utilization for the proposed scheme is only moderately larger than that of the PB scheme, and still remains significantly smaller to that of the FB scheme, even as the filter size increases. As a result, the proposed technique provides an appropriate trade off between EMB and resource utilization.

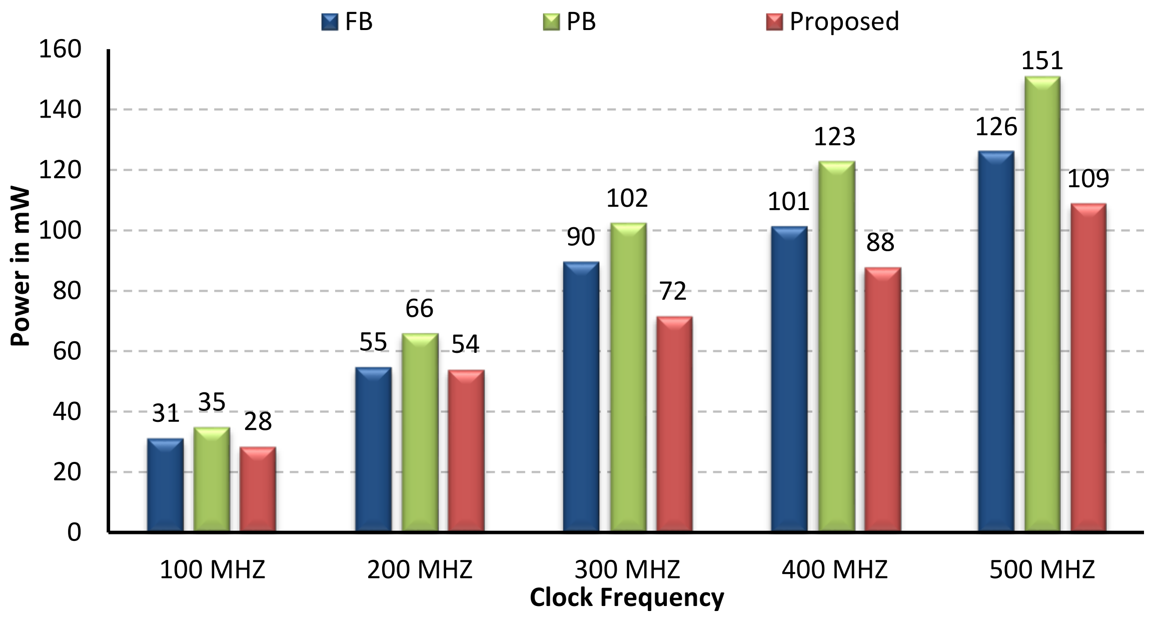

Figure 2.7: FB, PB and proposed schemes power consumption (mW) for a 3×3 filter

Filter Size	FB		PB		Proposed	
	FF	LUT	FF	LUT	FF	LUT
3×3	2236	194	214	115	234	133
5×5	4348	300	310	146	433	231
7×7	6460	407	406	176	632	328
9×9	8572	513	502	204	831	426
11×11	10684	619	598	234	1030	524

Table 2.2: Comparison between FB, PB and proposed schemes for various filter sizes in terms of flip flops and LUTs using xilinx virtex 5 FPGA device

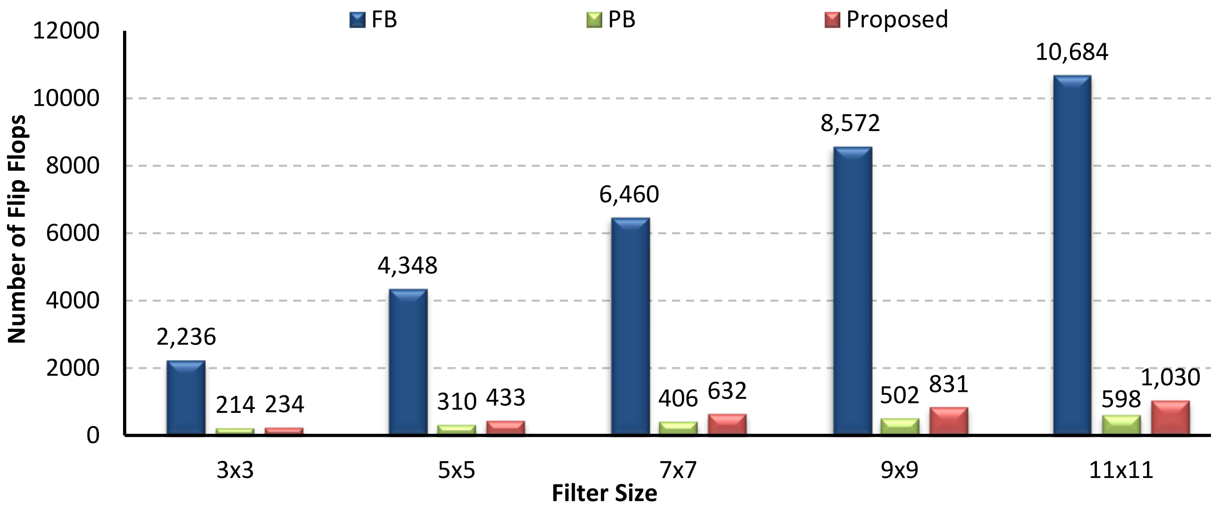

Figure 2.8: Comparison between FB, PB and proposed schemes for various filter sizes in terms of flip flops shown using barplots

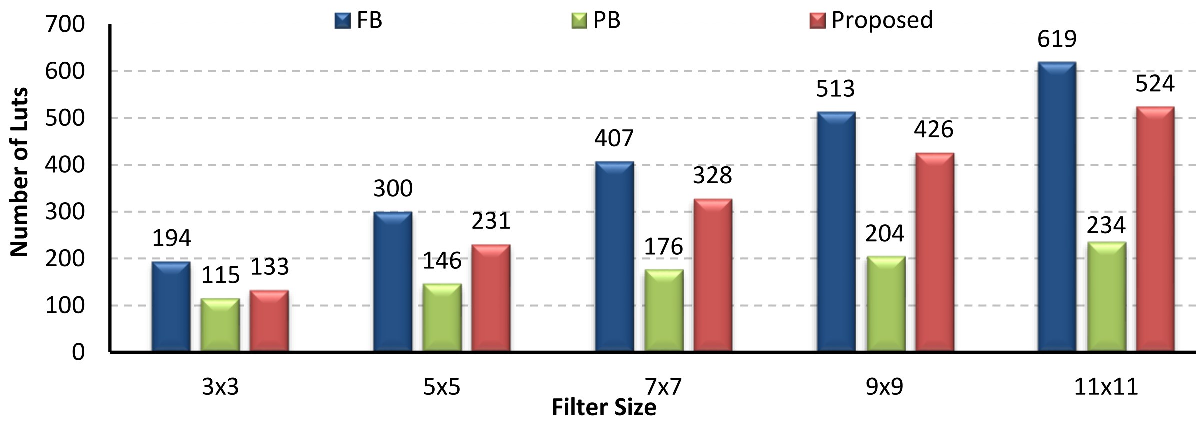

Figure 2.9: Comparison between FB, PB and Proposed schemes for various filter sizes in terms of LUTs shown using barplots

2.6 Conclusions

A new scheme for separable convolution on FPGA was proposed. Compared to the existing schemes, the proposed technique provides a more appropriate balance between on-chip resource utilization and EMB.

For instance, although PB requires moderately fewer resources compared to the proposed scheme, this advantage is overshadowed by its high EMB requirements. External memory elements such as SDRAM or DDRAM may not be capable of transfer rates comparable to that of the system clock. For example, the DDRAM2 available on Virtex 5 LX devices has a maximum transfer rate of 600 Mb/s [29], or 75 Mpixels/s for grayscale images, while the Virtex 5 LX50T [28] system clock runs at 500MHz. According to Table 2.1, PB requires that EMB is $K/2$ pixels/cc for a processing rate of 0.5 output pixels/cc. To achieve the required EMB using DDRAM2, the system clock should be slowed down, resulting in a processing rate which is $K/2$ times slower than that of the proposed scheme.

The EMB requirements for FB allow the processing rate to be twice than that of the proposed method. However, the resource requirements for FB significantly limit its usability with large images or large filter kernels. For example, using Virtex 5 LX50T with FB and a 11×11 filter, only images with up to 345 columns can be processed. The proposed scheme is independent of image size, therefore considerably larger filters can be implemented on Virtex 5 LX50T.

Although it is not the objective of this work, it is worth mentioning that employing LUT-based multiplier-equivalent circuits [18],[19],[20],[21],[22] may further reduce resource utilization.

Chapter 3

An Efficient Method of Error Reduction in Logarithmic Multiplication for Filtering Applications

3.1 Abstract

Many real-time digital signal and image processing applications demand high performance. This can be achieved at the expense of area, power dissipation and accuracy. Multiplication is one of the most critical and time consuming step in filtering applications. An alternative way is to convert multiplication to addition by converting the integer number system to the logarithmic number system. However, multiplication in logarithmic domain in not accurate. In image and signal processing applications, where filtering is most widely used, small errors introduced with multipliers do not affect the results significantly and can be used in practice. The proposed method alters the filter weights so that the error produced by multiplication in the logarithmic domain is reduced without increasing any additional circuitry or resources except the basic Mitchell-based logarithmic multiplier.

3.2 Introduction

Filtering is a computationally complex process which is most widely used in image and signal processing applications. Arithmetic operations such as multiplication and division are complex in terms of area, delay, speed and power. Converting the integer number system to the logarithmic number system (LNS) will convert arithmetic operations such as multiplication to addition, division to subtraction, power to multiplication, and roots to division. However, the results obtained in LNS are not accurate. In image and signal processing applications such as image enhancement [4], smoothing [12],[5] and edge detection [2], accurate result of the multiplication is not essential and an approximate result of the multiplication is practically acceptable in most cases. Hence, LNS is a simple alternative to computing multiplication.

Mitchell [7] first proposed a simplified method for computing multiplication in LNS domain by using a straight line approximation technique. Mitchell's method calculates the approximate logarithmic value of a number by encoding a binary number into a format from where the characteristic is determined exactly and the mantissa is calculated approximately. The logarithm and anti-logarithm functions in Mitchell's method use less hardware and no LUT is required. However, multiplication results obtained using Mitchell's method have a maximum possible relative error of around 11% and the average error is around 3.8%.

Numerous attempts have been made to improve Mitchell's approximation by proposing several error correction circuits. These methods include Mitchell-based methods [30],[31],[32],[33], LUT-based methods [34],[35],[36], and region-based approaches [37],[38],[39],[40],[41],[42].

In [7], Mitchell has shown that the relative error increases with the number of bits with the value of 1 in the mantissa. Using this idea, the operand decomposition (OD) method proposed in [30] reduces the number of 1 bits by decomposing the operands, which in turn decreases the chances of carryover from the mantissa part to the integer part during the logarithmic summation step. In [31], an implementation of 2D convolution based on the Mitchell logarithmic multiplier is presented with an aim of minimizing the power consumption. A partitioning and gating technique is proposed to reduce the switching activity based on the detection of insignificant data bits. In [32],[33], a single stage of the iterative algorithm follows the idea of Mitchell but uses a different error correction circuit which aids in increasing the accuracy and efficiency of iterative method. The final pipelined hardware of the itertaive method results in reduction of logic utilization and power consumption.

An LUT-based approach to design error correction circuits for the logarithmic multiplier is proposed in [34]. This approach is faster and has a simple error correction circuit. But, this method requires large number of logic gates to implement the LUT. A hardware efficient architecture based on [34] is proposed in [35]. It uses an LUT followed by a multiplier-less linear interpolation stage. In [36], the author proposed logarithmic approximation by combining the Mitchell method with two correction stages. The first stage is a piece-wise linear interpolation with power-of-two slopes and truncated mantissas', and the next stage is an LUT-based correction that corrects the piece-wise interpolation error. Hence, LUT-based methods are fast but inefficient in terms of hardware resource utilization which increases abruptly for inputs with large bit width.

Several region-based approaches are proposed in the literature for improving the accuracy of the results obtained using the Mitchell logarithmic multiplier. In [37] and [38], the mantissa range of [0,1) was partitioned into four parts, and a separate approximation equation was obtained by using trial and error method for each sub interval. The coefficients of each interval are chosen in such a way that numerator is integer and denominator is powers of two, where as the coefficients used in [37] are not powers of two. It has to be noted at this point that the sum operations are performed at full precision. Hence, the error while computing logarithmic process is reduced but with an increased hardware and longer computation time. In [37] and [38], both methods use all the bits of the mantissa and the coefficients are not all powers of two, which is addressed in the next methods by [39] and [40].

In [39], the author takes into consideration only the mantissas' four most significant bits (MSB) and divides the power-of-two intervals into two regions and further reduce the complexity by only using the mantissas' four MSBs. In [40] and [41], the author extends this approach by dividing the interval into two, three, and six regions and further reduces the complexity by only using the mantissas' three MSBs. The advantage of using this method is that the error correction does not need to be done on all mantissa bits. In [42], the exact logarithmic curve is divided into two symmetric regions obtaining a reduced error of 0.045. It has to be noted at this point that the coefficients derived in methods [39], [40] and [42] are restricted to the powers of two. Based on [40] and [41], a logarithmic multiplier architecture for complex numbers is proposed in [43]. Besides the region-based approach, linear programming techniques are used to obtain optimized coefficients for designing logarithmic converters [44] and anti-logarithmic converters [45].

This chapter is organised as follows: Section 3.3 discusses in more detail the Mitchell logarithmic multiplier and the iterative Mitchell (IM) multiplier. Section 3.4 discusses the proposed logarithmic multiplier. Section 3.5 presents results with comparisons between the proposed and existing logarithmic multipliers. Section 3.6 presents briefly the extension of the proposed logarithmic multiplier. Some concluding remarks are presented in section 3.7.

3.3 Review of Logarithmic Multipliers

Logarithmic multiplication introduces an operand conversion from the integer number system into the LNS. The multiplication of two operands N_1 and N_2 is performed in three phases: calculating the operand logarithms, the addition of the operand logarithms, and the calculation of the anti-logarithm. The main advantage of this method is the substitution of the multiplication with addition after the conversion of operands to logarithms.

3.3.1 Mitchell Logarithmic Multiplier

Mitchell presented a simple method to approximate the logarithm and anti-logarithm calculations using piece-wise straight line approximations of the log and anti-log curves. The approximated curve obtained from the Mitchell log approximation and the actual curve for log_2 of an integer N is shown in Fig. 3.1. Mitchell's method accurately calculates the result when the number N is a perfect power-of-two numbers such as 2, 8, 16, 32, 64, 128, 256 and so on. The maximum error occurs at the mid-point between the two consecutive power-of-two numbers such as 192. The error in Mitchell's log is due to the fractional part of the log. Hence, whenever the fractional part is zero (i.e. the operand is exact power of 2), the Mitchell log curve intersects the actual log curve. In this section, derivation for multiplication of operands N_1 and N_2 using the Mitchell method is explained in more detail.

The binary representation of the number N can be written as given below:

$$N = \sum_{i=j}^{k} 2^i Z_i \tag{3.1}$$

Since Z_k is the MSB, we may assume $Z_k = 1$ for any valid $k \geq j$. Factoring out the value of 2^k from N, we get

$$N = 2^k \left(1 + \sum_{i=j}^{k-1} 2^{(i-k)} Z_i\right) = 2^k\,(1 + x) \tag{3.2}$$

where, k is a characteristic or the position of MSB with the value of '1',

Z_i is the bit value at i^{th} position,

x is the fraction or mantissa.

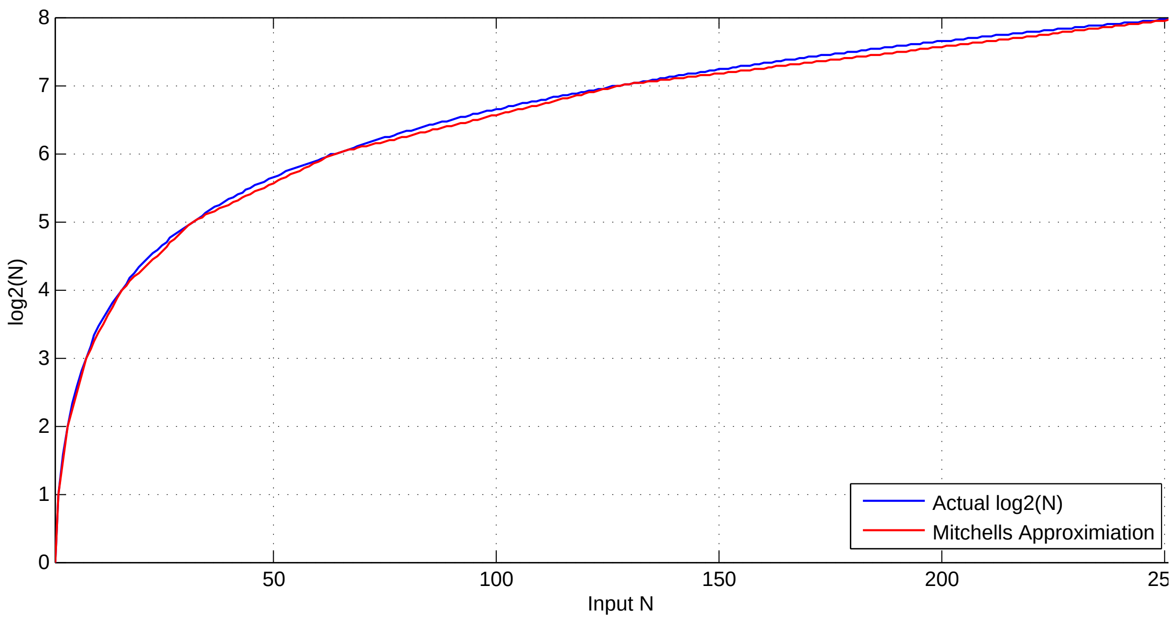

Figure 3.1: Actual values and Mitchell's approximated values of $log_2(N)$

The logarithm with the base 2 of N is derived as given below:

$$log_2\left(N\right) = log_2\left(2^k\left(1+\sum_{i=j}^{k-1}2^{(i-k)}Z_i\right)\right) = log_2\left(2^k\left(1+x\right)\right) = k + log_2(1+x) \tag{3.3}$$

Mitchell used straight line linear approximation for $log_2(1+x)$, which only uses the first linear term in the Taylor series: $log_2(1+x) \approx x$. This eliminates the need for an LUT. The approximate logarithm of the binary number using Mitchell is shown below:

$$log_2\left(N\right)_{MA} = log_2\left(2^k\left(1+\sum_{i=j}^{k-1}2^{i-k}Z_i\right)\right) = log_2\left(2^k\left(1+x\right)\right) = k + x \tag{3.4}$$

The true logarithm of this binary number is shown below:

$$log_2\left(N\right)_{true} = k + log_2(1+x) \tag{3.5}$$

Let N_1 and N_2 be the input operands. The actual logarithm of the product and approximate logarithm of the product are shown in the following two equations.

$$log_2\left(N_1 \times N_2\right)_{true} = k_1 + k_2 + log_2(1+x_1) + log_2(1+x_2) \tag{3.6}$$

$$log_2\left(N_1 \times N_2\right)_{MA} \approx k_1 + k_2 + x_1 + x_2 \tag{3.7}$$

The characteristic numbers k_1 and k_2 represent the MSBs of the operands' N_1 and N_2 with the value of '1'. x_1 and x_2 represent fractions of the operands' N_1 and N_2 respectively. For 16 bit numbers, the range of k_1 and k_2 characteristic numbers is from 0 to 15. The fractions are in the range of $[0, 1)$. The above equation 3.7 is rewritten as two separate expressions to account for the two possible cases of carry. One is the case of "no carry" from the mantissa x to the characteristic k, and the other is the case where the "carry" occurs.

$$log_2\left(N_1 \times N_2\right)_{MA} \approx \begin{cases} k_1 + k_2 + x_1 + x_2, & x_1 + x_2 < 1 \\ 1 + k_1 + k_2 + (x_1 + x_2 - 1), & x_1 + x_2 \geq 1 \end{cases} \tag{3.8}$$

Taking antilogarithm to the above equation 3.8 gives the final product P_{MA}.

$$P_{MA} \approx \begin{cases} 2^{k_1+k_2}\left(1 + x_1 + x_2\right), & x_1 + x_2 < 1 \\ 2^{k_1+k_2+1}\left(x_1 + x_2\right), & x_1 + x_2 \geq 1 \end{cases} \tag{3.9}$$

The error due to Mitchell's approximation is in the range $[0, 0.08639]$ and attains the maximum value when the fractional part of the log is equal to 0.44 (i.e. when the number is in the middle of power of 2). The maximum possible relative error for Mitchell's log multiplication is around 11% and the average error is around 3.8%. The error in Mitchell's method is always positive which can be reduced by successive multiplications, but the error correction can start only after the calculation of term $(x_1 + x_2)$. IM log multiplier which is based on successive multiplications is explained in more detail in the next section.

3.3.2 Derivation of IM log multiplier

The logarithm approximation presented in equation 3.8 can be simplified by using the IM algorithm [32],[33], which is explained in more detail in this section. According to equation 3.2, the true product of operands N_1 and N_2 can be expressed as

$$P_{true} = 2^{k_1+k_2}\left(1 + x_1 + x_2\right) + 2^{k_1+k_2}\left(x_1 \times x_2\right) \tag{3.10}$$

In order to simplify the approximation error, the following expression derived from equation 3.2 is given below:

$$N - 2^k = 2^k \times x \tag{3.11}$$

Using equation 3.11 in equation 3.10, the following expression is derived which is given below:

$$P_{true} = 2^{(k_1+k_2)} + \left(N_1 - 2^{k_1}\right) \times 2^{k_2} + \left(N_2 - 2^{k_2}\right) \times 2^{k_1} + \left(N_1 - 2^{k_1}\right) \times \left(N_2 - 2^{k_2}\right) \tag{3.12}$$

Let us assume

$$P^{(0)}_{approx} = 2^{(k_1+k_2)} + \left(N_1 - 2^{k_1}\right) \times 2^{k_2} + \left(N_2 - 2^{k_2}\right) \times 2^{k_1} \tag{3.13}$$

Substituting equation 3.13 in equation 3.12, we get

$$P_{true} = P^{(0)}_{approx} + \left(N_1 - 2^{k_1}\right) \times \left(N_2 - 2^{k_2}\right) \tag{3.14}$$

If we discard the term $\left(N_1 - 2^{k_1}\right) \times \left(N_2 - 2^{k_2}\right)$ from equation 3.14, we have approximate product of the operands N_1 and N_2. The error term $\left(N_1 - 2^{k_1}\right) \times \left(N_2 - 2^{k_2}\right)$ can be calculated in the similar manner as $P^{(0)}_{approx}$ until $P^{(0)}_{approx}$ is equal to P_{true}. Hence, the comparison of the addend $(x_1 + x_2)$ is completely eliminated, and the error correction can start immediately after removing leading ones from the operands N_1 and N_2. The expression for multiplication using i correction terms is given below:

$$P_{true} = P^{(0)}_{approx} + C^1 + C^2 + ... + C^i = P_{approx} + \sum_{j=1}^{i} C^j \tag{3.15}$$

The number of iterations required for obtaining results with zero error is equal to the number of bits with the value of '1' in the operand. Moreover, the operand which has smaller numbers of bits with the value of '1' is taken into consideration for calculating the number of iterations.

3.4 Proposed logarithmic multiplier

For filtering applications, the filter weights are already known and fixed. Taking this into account, the proposed method modifies the given weights so that the product error is reduced. The derivation for calculating optimized weights for the proposed method is described below:

Let N_1 be the image pixel and N_2 be the filter weight. Similar to the representation in equation 3.2, N_1 can be expressed as given below:

$$N_1 = 2^n + (N_1 2^{-n} - 1)2^{-n} \tag{3.16}$$

Applying $log2$ on both sides of equation 3.16 we get,

$$log_2(N_1) = n + (N_1 2^{-n} - 1) \tag{3.17}$$

Here it is assumed that $log_2\left((N_1 2^{-n} - 1)2^{-n}\right) \approx N_1 2^{-n} - 1$. Similarly applying $log2$ on the filter weight N_2 we get,

$$log_2(N_2) = W_I + W_F \tag{3.18}$$

where, W_I is the integer part of the weight N_2, and
W_F is the fractional part of the weight N_2.
The product of N_1 and N_2 in LNS domain can be derived as

$$log_2(N_1 \times N_2) = \big[n + (N_1 2^{-n} - 1)\big] + [W_I + W_F] \tag{3.19}$$

By rearranging the above equation and taking into consideration the cases of "no carry" and "carry", the following equations are derived respectively.

$$log_2(N_1 \times N_2) \approx \begin{cases} [n + W_I] + [(N_1 2^{-n} - 1) + W_F]\,, & (N_1 2^{-n} - 1) + W_F < 1 \\ [n + W_I + 1] + [(N_1 2^{-n} - 2) + W_F]\,, & (N_1 2^{-n} - 1) + W_F > 1 \end{cases} \tag{3.20}$$

Applying antilog on both sides of the equation 3.20, we get

$$(N_1 \times N_2) \approx \begin{cases} 2^{W_I} N_1 + 2^{(n+W_I)} W_F, & (N_1 2^{-n} - 1) + W_F < 1 \\ 2^{W_I+1} N_1 + 2^{(n+W_I+1)} (W_F - 1), & (N_1 2^{-n} - 1) + W_F > 1 \end{cases} \tag{3.21}$$

Applying mean square error (MSE) to equation 3.21 w.r.t product of original operands N_1 and N_2 and equating it to zero, the following expression is derived:

$$\sum_{N_1} \left[N_1 2^{W_I} + W_F 2^{(n_1+W_I)} - N_1 N_2\right]^2 + \sum_{N_2} \left[N_2 2^{(W_I+1)} + (W_F - 1) 2^{(n_2+W_I+1)} - N_1 N_2\right]^2 = 0 \tag{3.22}$$

Now differentiating the above equation 3.22 w.r.t fractional part of the filter weight W_F, we get

$$W_F = \frac{\sum\limits_{N_1} N_1 2^{n_1} \left(N_2 - 2^{W_I}\right) + \sum\limits_{N_2} N_2 2^{(n_2+1)} \left(N_2 - 2^{W_I+1}\right) + \sum\limits_{N_2} 2^{(2n_2+W_I+2)}}{\sum\limits_{N_1} 2^{(2n_1+W_I)} + \sum\limits_{N_2} 2^{(2n_2+W_I+2)}} \tag{3.23}$$

3.5 Results and Discussion

The fractional part of the original filter weight is modified and replaced by the value obtained using the equation 3.23. This value ensures the reduction of error in multiplying N_1 and N_2. The proposed method, Mitchell method, OD method, and IM method with 2 and 3 stages are implemented in Matlab, and the average relative error is calculated to evaluate accuracy of the proposed algorithm.

The average relative error is calculated as follows:

$$AverageErrorPercentage = \sum_{i=1}^{N} \frac{EP_i}{N} \tag{3.24}$$

where, $ErrorPercentage = \left[\frac{TV - LV}{TV}\right] \times 100;$
TV is the true value obtained using binary multiplication,
LV is the value obtained using the above proposed logarithmic multipliers,
N is the total number of multiplications performed.

For instance, if the input operands N_1 and N_2 are represented using 8 bit numbers, then all the combination of numbers ranging from 1 to 256 are multiplied, and the average error percentage is calculated. The average error percentage reported for the Mitchell algorithm in [7] is 3.87%, OD in [30] is between 2.07% and 2.15%, and the IM algorithm with 2 stages is between 0.83% and 0.99%. The average error percentage for the proposed logarithmic multiplier is around 0.1%.

MSE's of all the above methods are calculated and tabulated in Table 3.1. Fig. 3.2 depicts the MSE's of all the above methods using barplots. It is evident from Table 3.1 and Fig. 3.2 that, logarithmic multiplication of operands N_1 and N_2 using the proposed method produces less error when compared to the Mitchell, OD, and IM algorithm with 2 stages. The IM algorithm with 3 stages produces the least error of all the methods but at a cost of much increased resource utilization. The objective of the proposed logarithmic multiplier is to reduce the product error without any increase in the circuit of the basic Mitchell multiplier. In section 3.6, the proposed method for logarithmic multiplication is extended where the error can be further reduced with little increase in the circuit of the basic Mitchell multiplier. It is worth

Input Signal	Mitchell	OD	IM 2 stages	IM 3 stages	Proposed
Input1	11.84	4	1.27	0.1	1.97
Input2	30	9.38	10.54	0.36	7.36
Input3	22.28	6.62	1.98	0.13	3.07
Input4	7.79	2.14	0.68	0.04	1.1
Input5	3.44	1.47	0.51	0.05	0.61
Input6	17.32	4.59	1.96	0.12	2.23
Input7	32.22	10.22	7.82	0.3	5.2
Input8	25.87	8.24	13.85	0.41	9.42
Input9	26.16	7.63	2.54	0.16	3.28
Input10	10.52	3.32	4.99	0.22	4.67
Input11	14.07	4.52	9.48	0.35	9.18
Input12	15.33	4.88	6.97	0.27	6.8
Input13	15.52	4.77	7.43	0.31	7.04

Table 3.1: MSE between the Mitchell, OD, IM and proposed methods using input signals of size 1×256 and Gaussian filter of size 1×13

mentioning at this point that the proposed method doesn't require any extra circuitry for hardware implementation, since the original filter weights are replaced by the modified weights. Hence,

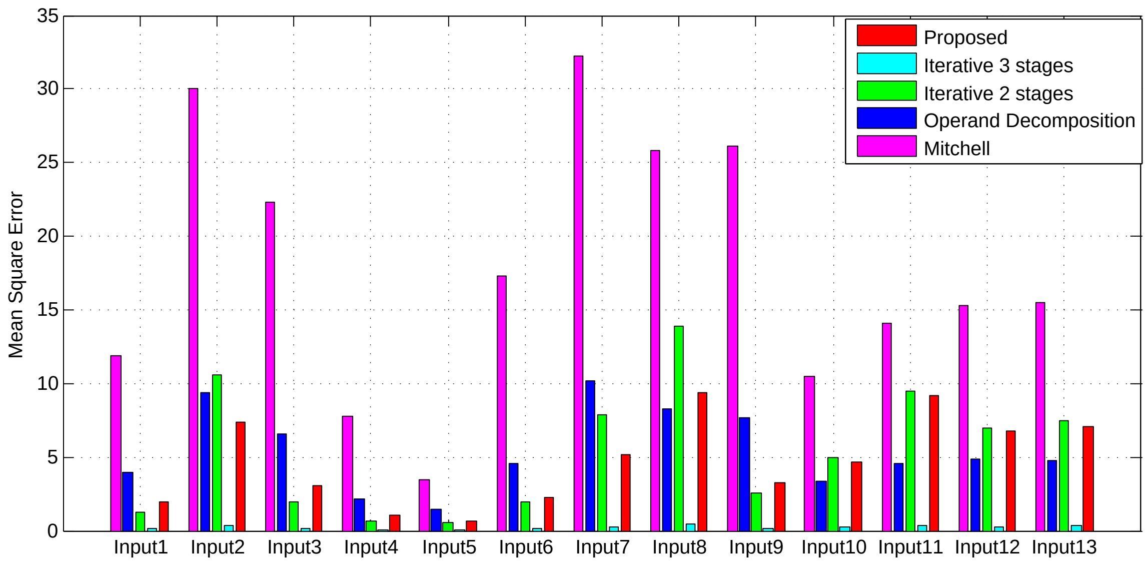

Figure 3.2: MSE between the Mitchell, OD, IM and proposed methods using input signals of size 1×256 and Gaussian filter of size 1×13, shown with barplots

the proposed method can be implemented on FPGA without any increase in the circuitry of the basic Mitchell multiplier, but it still provides accurate results. The authors in [32] implemented 16 bit logarithmic multipliers on the Xilinx xc3s1500-5fg676 FPGA [46] using the Mitchell method, OD method and the IM method with 1, 2 and 3 stages. The resource utilization for the non-pipelined implementation of 16 bit logarithmic multiplier as reported in [32] is compared with the proposed method. Table 3.2 presents the comparison of the above mentioned methods in terms of LUTs, slices, flip flops, and input/output blocks (IOBs).

Logarithmic Multiplier	LUTs	Slices	Slice FFs	IOBs
Mitchell	622	321	66	99
OD	1187	604	101	99
IM 1 stage	533	276	64	99
IM 2 stages	1099	564	80	99
IM 3 stages	1596	814	77	99
IM 4 stages	1937	993	78	99
Proposed	622	321	66	99

Table 3.2: Resource utilization of non-pipelined logarithmic multiplier using the Mitchell, OD, IM (1, 2, 3 and 4 stages), and proposed method

3.6 Extension of Proposed Logarithmic Multiplier

In this section, the proposed method is extended, based on the Mitchell algorithm where the error is further reduced, with little increase in circuitry of the basic Mitchell logarithmic multiplier. Instead of storing the filter integer values, storing the true $log2$ value of filter weights reduces the error of the product result. Based on this observation, all the above mentioned algorithms are implemented by storing true $log2$ value of the second operand, i.e. N_2.

A comparison of existing methods (Mitchell, OD and IM) with the proposed method extension is performed, and the MSE's are calculated for each method. Table 3.3, Fig. 3.3, Fig. 3.4 provides a comparison between the Mitchell, proposed method extension for Mitchell, OD, and proposed method extension for OD in terms of MSE for input signals of size 1×256 and Gaussian filter of size 1×13. From Fig. 3.3, it is evident that the proposed method extension for Mitchell reduces the product error significantly without any additional utilization of resources. This is achieved by storing true $log2$ value of the filter (i.e. N_2), instead of storing the original number. Similarly, from Fig. 3.4, the proposed method extension for OD is not feasible and does not provide an accurate product when compared to the original OD method. Moreover, the proposed method extension for OD requires a storage of true $log2$ values of all the decomposed operands which increases the memory and resource utilization. Hence, the extension of the proposed method is not applicable for the OD method.

In the IM method, the first stage is similar to the Mitchell algorithm and the second stage involves the error correction circuit. From our analysis and implementation, the product error can be reduced significantly when both the first and second stages of the IM method are implemented using the proposed method extension for Mitchell. Table 3.4 and Fig. 3.5 provides the comparison between IM and the proposed method extension for IM in terms of MSE using input signals of size 1×256 and Gaussian filter of size 1×13. In Table 3.4 and Fig. 3.5, the column headed "IM 2 stages" represents direct implementation of the IM method with 2 stages. The column headed "IM 2 stages-Modified" represents implementation of the IM method with 2 stages where true $log2$ value of the operands is taken into consideration. The column headed "IM-Mitchell" represents implementation of the IM 2 stages using the original Mitchell

Input Signal	Mitchell	Mitchell Proposed Extension	OD	OD Proposed Extension
Input1	11.84	1.95	4.00	1.91
Input2	30.00	8.96	9.38	19.07
Input3	22.28	2.82	6.62	3.74
Input4	7.79	1.07	2.14	1.26
Input5	3.44	0.56	1.47	0.80
Input6	17.32	2.35	4.59	3.78
Input7	32.22	6.42	10.22	14.15
Input8	25.87	12.59	8.24	25.38
Input9	26.16	3.27	7.63	4.63
Input10	10.52	5.71	3.32	9.15
Input11	14.07	11.51	4.52	17.05
Input12	15.33	8.55	4.88	12.89
Input13	15.52	8.94	4.77	14.23

Table 3.3: MSE between the Mitchell, proposed method extension for Mitchell, OD, and proposed method extension for OD using input signals of size 1×256 and Gaussian filter of size 1×13

method. The column headed "IM-Mitchell-Proposed-Extension" represents implementation of the proposed method extension for Mitchell. From Fig. 3.5, the proposed method extension for IM is more accurate when compared to the above mentioned methods. It has to be noted at this point that the proposed method extension for IM requires storing of true $log2$ values of all decomposed operands which account for an increase in the resource utilization. As a direction towards the future scope of work, the increase in memory and resource utilization for storing the true $log2$ values of the actual operand and the decomposed operand for the IM method can be optimized by using an 8 bit state diagram.

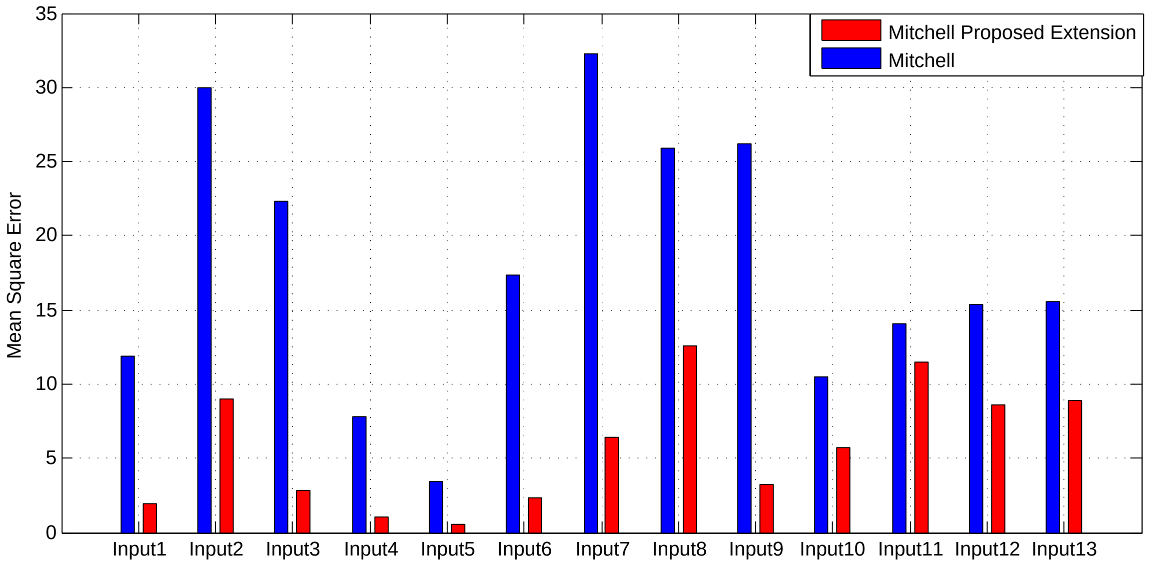

Figure 3.3: MSE between the Mitchell and the proposed method extension for Mitchell using input signals of size 1×256 and Gaussian filter of size 1×13, shown using barplots

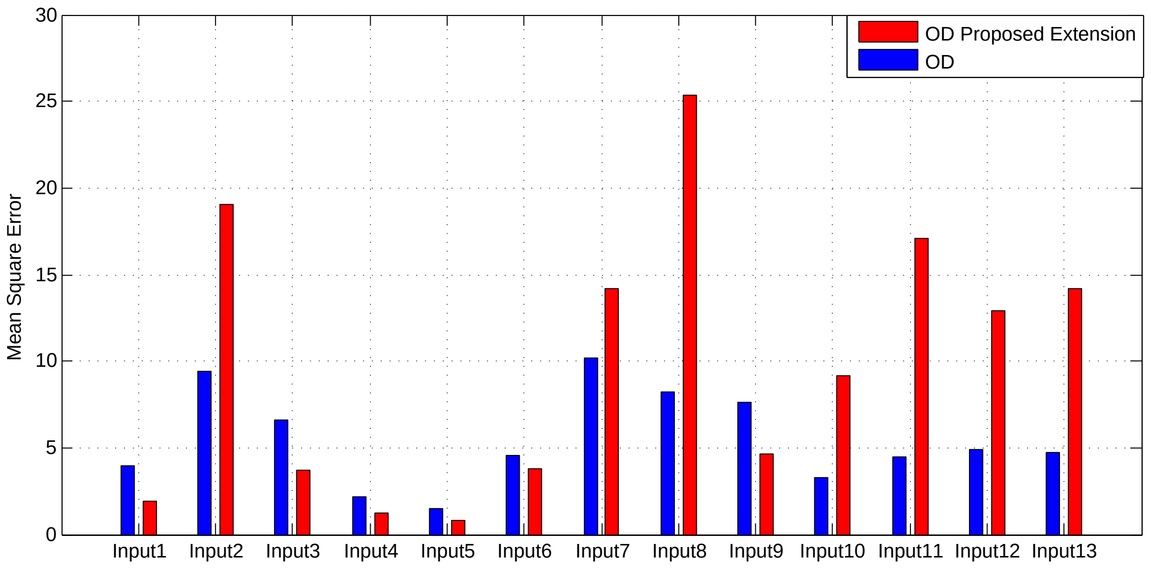

Figure 3.4: MSE between the OD and the proposed method extension for OD using input signals of size 1×256 and Gaussian filter of size 1×13, shown using barplots

Input Signal	IM 2 Stages	IM 2 Stages-Modified	IM-Mitchell	IM-Mitchell-Proposed-Extension
Input1	1.27	0.91	0.23	0.11
Input2	10.54	7.65	0.51	0.36
Input3	1.98	1.43	0.34	0.14
Input4	0.68	0.51	0.23	0.07
Input5	0.51	0.4	0.13	0.02
Input6	1.96	1.53	0.39	0.12
Input7	7.82	5.47	0.55	0.27
Input8	13.85	9.92	0.57	0.4
Input9	2.54	1.73	0.41	0.15
Input10	4.99	3.67	0.29	0.24
Input11	9.48	7.2	0.34	0.42
Input12	6.97	5.28	0.35	0.38
Input13	7.43	5.6	0.34	0.33

Table 3.4: MSE between IM and the proposed method extension for IM using input signals of size 1×256 and Gaussian filter of size 1×13

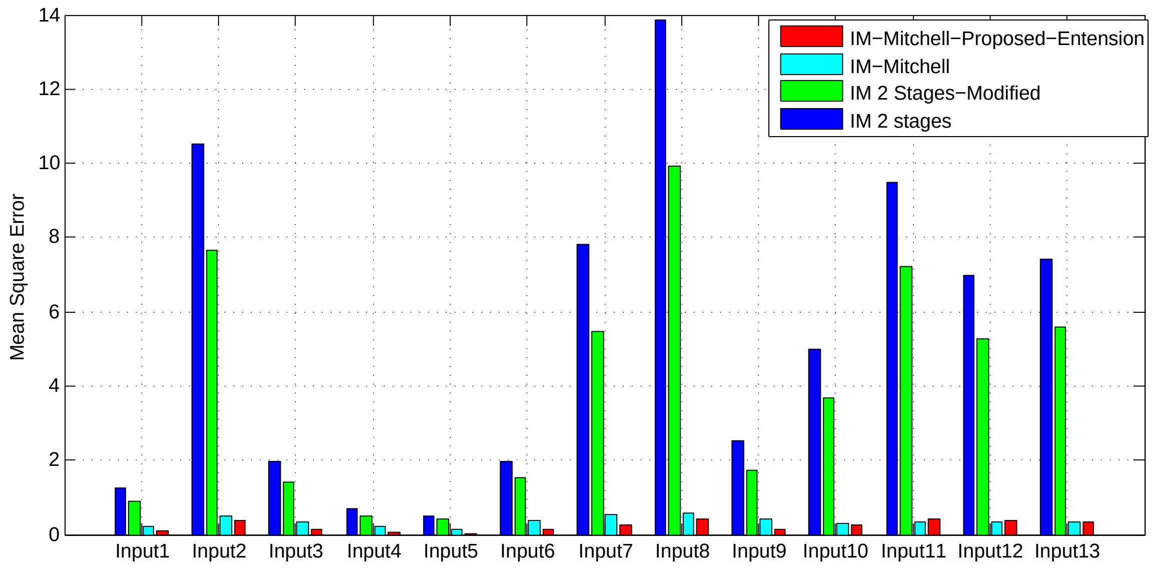

Figure 3.5: MSE between IM and the proposed method extension for IM using input signals of size 1×256 and Gaussian filter of size 1×13, shown using barplots

3.7 Conclusions

The proposed method for logarithmic multiplication which alters the filters weights provided significant reduction in product error without any increase in the resource utilization or circuit besides the basic Mitchell multiplier. The average error percentage for the Mitchell algorithm in [7] is 3.87%, OD in [30] is between 2.07% and 2.15%, and the IM algorithm with 2 stages is between 0.83% and 0.99%. The average error percentage for the proposed logarithmic multiplier is around 0.1%. Hence, the proposed method for logarithmic multiplication is best suited for filtering applications where the filter weights are known and fixed. The proposed method extension for Mitchell further reduces the error without any increase in the resource utilization. In the case of the proposed method extension for IM (i.e. "IM-Mitchell-Proposed-Extension"), error is further reduced at a cost of few extra resources which are essential to store the true $log2$ values of the decomposed operands. As a future work, optimization can be done in storing the true $log2$ values of the original operand and the decomposed operand for the IM method using an 8 bit state diagram.

Chapter 4

Species-Specific Fish Feature Extraction Using Gabor Filters

4.1 Abstract

Fish recognition and classification are challenging when performed on video data obtained in non-controlled environments (NCE's) such as in natural waters. Many NOAA Fisheries surveys use underwater cameras to gather video data for this purpose, which facilitate the analysis of fish populations. Since the amount of data is large, manual data analysis is insufficient. Automatic processing tools are necessary. Most techniques that extract features from fish are in two categories. In the first, features are specific to fish but not necessarily to a particular species. Yet, such measurements are often unreliable when extracted from video obtained in (NCE's), since they strongly depend on the aspect of fish with respect to the camera. In the second, features are generic and may include texture and shape descriptors. Such features do not target specific species of interest. In this chapter, we present an automatic technique using Gabor filters to extract characteristic features from two important species, namely, Epinephelus morio (which has a vertical band located at the tale) and Ocyurus chrysurus (which has a long horizontal line that runs across the body). The proposed algorithm is tested on 200 frames, each containing several fish and non-fish regions. The detection rate is 70.6% for Epinephelus morio and 80.3% for Ocyurus chrysurus, while 23.5% of the undetected Epinephelus morio cases do not have a visible tail band, and 16.7% of the undetected Ocyurus chrysurus cases do not have a visible straight body line. The false alarm rates are 3.8% and 2.1%, respectively. [1]

4.2 Background

Underwater video and still images are used by many programs within National Oceanic and Atmospheric Administration (NOAA) Fisheries with the objective of identifying and quantifying

[1]This work was supported by a grant from NOAA Fisheries Advanced Sampling Technology Working Group. The project was NASA (2011)-STENNIS-02 "Feature Analysis for Classification of Fish in Underwater Video" with University of New Orleans award number CON000000001307.

living marine resources. The NOAA Southeast Fisheries Science Center (SEFSC) – laboratories at Pascagoula, MS, Panama City, FL, and Beaufort, NC, all conduct annual fishery independent reef fish surveys using video, trap, and hook gear. The surveys target reef habitat and yield demographic data and abundance indices used in assessments for many federally managed species. Video techniques overcome the fish sampling limitations imposed by depth, fish behaviour, seafloor rugosity, and the selectivity inherent in hook, trap and trawl methods [47].

In a given year, cameras are deployed at a large number of locations that are not amenable to sampling with nets or other means. Some of the systems use stereo pairs of cameras [48] that allow fish lengths to be estimated while others are used simply to identify and count the fish present. Analyses of the images from these surveys are used to produce indices of abundance and size distributions for the fish species observed. These data, in turn, are used in stock assessment models that ultimately influence regulations for the harvest of reef fish. Human analysts are required to view each image sequence to identify and enumerate fish species present at each location and measure their lengths. The process of manual analysis is both labour intensive and time consuming and is a significant limitation on how much this type of population sampling can be utilized. Automated image analysis capabilities are desperately needed in order to take full advantage of current image data collection technology.

Recent efforts to automate analysis of underwater images [49],[50],[51] of fish has focussed on detection and tracking of fish in sequences of images. Motion of the fish against a relatively static background has been exploited for detection and proven methods of object tracking [52] have been used to track fish from frame to frame. Accurate detection and tracking allows the number of fish present during a given time to be determined, but the ultimate goal is to count the number of fish of each species. Thus the next step in the process of automation is classification. A human analyst uses a multitude of cues to visually identify fish species. However, the primary features used are morphological properties such as shape of the body, head, fins, and tail and patterns in coloration. Two species having distinctive patterns in coloration that are frequently observed in survey images in the Gulf of Mexico are red grouper – Epinephelus morio (Valenciennes, 1828) and yellowtail snapper – Ocyurus chrysurus (Bloch, 1791).

In underwater grayscale images, Epinephelus morio (EM) frequently display a light colored band near the margin of the caudal fin that contrasts strongly with a black band on the extreme margin. On the other hand, Ocyurus chrysurus (OC) have, as their common name indicates, a yellow caudal fin. The same color extends in a tapering band along the side of the body through the eye to the anterior end of the head. In grayscale images, this band appears dark in contrast to the body's background color. The two species were selected in this work for their great importance in the Gulf of Mexico. More specifically, EM is the most abundant grouper species in the Gulf of Mexico. It accounts for the bulk of the commercial grouper landings, and is the second most commonly caught grouper species recreationally. OC are fished along the US south Atlantic coast and south-eastern Gulf of Mexico. They are managed as a single stock with allowable catches distributed between the south Atlantic and Gulf of Mexico regions. Currently, the stock allowable biological catch (ABC) is set at 2.9 million pounds, with 0.725 million pounds (25% of ABC) going to the Gulf of Mexico.

4.3 Gabor Filters Used For Fish Feature Extraction

In this chapter, Gabor filters (GF) are used to extract species-specific features from EM and OC. Gabor filters are widely used for texture segmentation [53],[54] and feature extraction [55]. In its most general form, the GF is a complex sinusoid (shown in Fig. 4.1-A) modulated by a Gaussian (shown in Fig. 4.1-B). An example of a GF filter is depicted in Fig. 4.1-C. The mathematical representation of a horizontally oriented spatial GF is as follows:

$$h(x,y) = A \exp\left\{\frac{-x^2}{2\sigma_x^2} - \frac{y^2}{2\sigma_y^2}\right\} \exp\left\{2\Pi i F x_r\right\} \tag{4.1}$$

where, A is a constant,

F is the spatial frequency, and

σ_x and σ_x are the standard deviations of the GF in the x and y directions respectively.

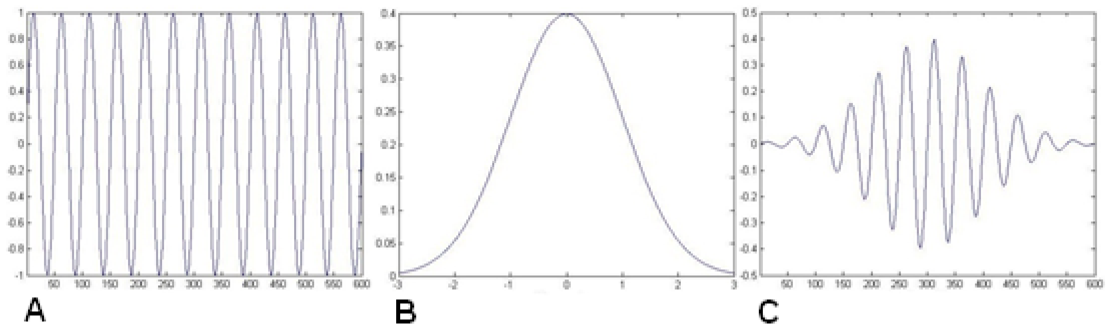
Figure 4.1: 1D Gabor filter

A. Sinusoid. B. Gaussian. C. Gabor filter.

As will be described later in this chapter, the feature extraction techniques employed in this work require that the filter is scaled according to the fish size. The GF filter was employed in this work owing to the ease with which its parameters can be tuned. The literature review of previous works on fish classification is found in [49],[50],[51],[55],[56].

A GF oriented vertically exhibits a strong response for horizontal details as shown in Fig. 4.2-A. Similarly, a GF oriented horizontally emphasizes vertical details as shown in Fig. 4.2-B. The size of the filter is adjusted according to the size of the fish being tested by assigning the filter standard deviation to be proportional to the square root of the fish area. In other words, $\sigma_x^2 = \sigma_y^2 = \left(\alpha \times \sqrt[2]{Area}\right)$, where α is a user defined constant empirically chosen to be 0.035.

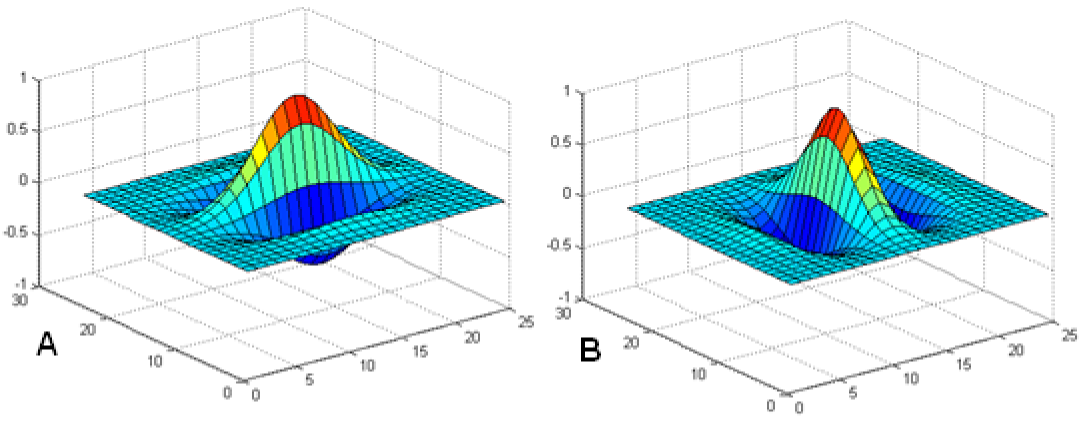
Figure 4.2: 2D Gabor filters with $\sigma_x^2 = \sigma_y^2 = 4$.

A. GF oriented vertically. B. GF oriented horizontally.

4.3.1 Epinephelus morio Feature Extraction Using Gabor Filters

EM has a bright band on its tail, a feature that is specific to its species. An example is depicted in Fig. 4.3-A. In order to detect this feature, the original fish image is first multiplied by its region mask, which is shown in Fig. 4.3-B, to isolate the fish from its surroundings, as shown in Fig. 4.3-C. The region mask is automatically obtained by subtracting the original image from the estimated background image [57], and by setting equal to 1 (white) or 0 (black) all pixels that exceed or fall below user defined thresholds, respectively. The background image is obtained as the pixel-wise median of several frames [58]. The resultant image is filtered separately with a horizontally and a vertically oriented GF to obtain images $I_{hor}(x, y)$ and $I_{ver}(x, y)$ as shown in Fig. 4.4-A and Fig. 4.4-B, respectively. The image ratio, $I_r(x, y) = \frac{I_{hor}(x,y)}{I_{ver}(x,y)}$, is used to emphasize the stripe as shown in Fig. 4.4-C. In order to eliminate the effect of vertical stripe-like edges in $I_{hor}(x, y)$ associated with the fish outline, a zone of pixels around the fish outline is set to zero. This is achieved by employing an erosion operation using a square structural element of size $[3\sigma_y \times 3\sigma_y]$. In most cases, the edges are caused by the intensity difference between the fish region and the background.

In order to quantify the presence of the stripe, the following approach is used. First, a vertical moving average (MA) filter, $f_{MA}(y)$ of size $W_{EM} \times 1$ is applied on $m_{I_r}^{EM}(x, y)$ to emphasize cases of consecutive vertical high intensity pixels, such as vertical stripes. The value associated with W_{EM} is selected by calculating the square root of the area of the fish in each frame which is empirically chosen to be 21. The maximum value per column $m_{I_r}^{EM}(x)$ is computed for EM as follows:

$$m_{I_r}^{EM}(x) = max_y \left\{ f_{MA}(y) * I_r(x, y) \right\} \tag{4.2}$$

where, $*$ represents the convolution operation.

The maximum value of $m_{I_r}^{EM}(x)$, namely $m_{I_r}^{EM}(max) = max_x \left\{ m_{I_r}^{EM}(x) \right\}$, quantifies the presence of a stripe in the fish region as shown in Fig. 4.5. A median filter of size 3×1 is applied on $m_{I_r}^{EM}(x)$ to eliminate narrow spikes which are unlikely to correspond to the tail band.

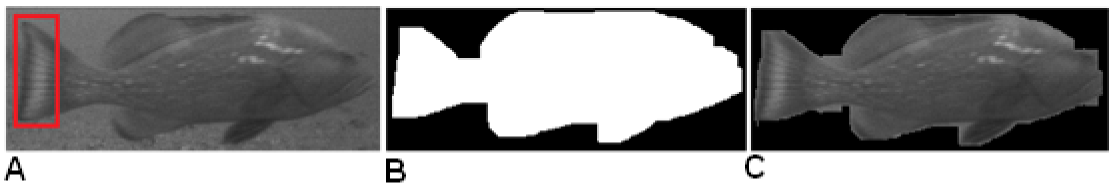

Figure 4.3: EM pre-processing steps.

A. Original image of EM with indicator for vertical stripe at the end of tale. B. Region mask of EM. C. Pre-processed image of EM.

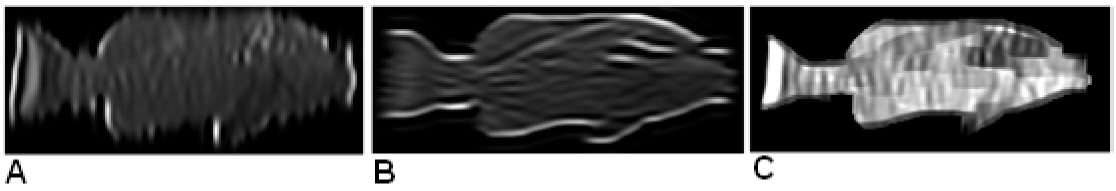

Figure 4.4: Filtering results of horizontal and vertical Gabor filters on EM.

A. Filtering result of horizontal Gabor filter. B. Filtering result of vertical Gabor filter. C. Ratio image $I_r(x, y)$.

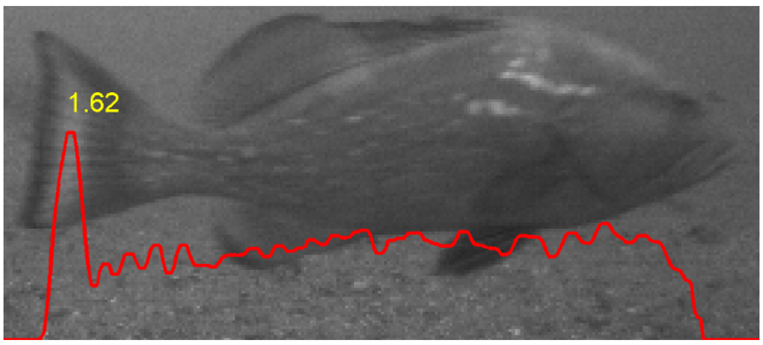

Figure 4.5: EM tail band detection – maximum value per column $m_{I_r}^{EM}(x)$.

4.3.2 Ocyurus chrysurus Feature Extraction Using Gabor Filters

The OC fish has a different feature specific to its species – a straight line across the fish body as depicted in Fig. 4.6-A. A similar approach explained for EM is followed for OC to extract the horizontal line along the length of its body. An OC example is shown in Fig. 4.6-A. Fig. 4.6-B depicts the corresponding region mask, and Fig. 4.6-C shows the isolated fish region. The resultant image is filtered separately with a vertically and a horizontally oriented GF to obtain images $I_{hor}(x, y)$ and $I_{ver}(x, y)$ as shown in Fig. 4.7-A and Fig. 4.7-B, respectively. However, the ratio of the images in the OC case is computed as $I_r(x, y) = \frac{I_{ver}(x,y)}{I_{hor}(x,y)}$ which highlights the horizontal line as shown in Fig. 4.7-C. In order to eliminate the effect of horizontal stripe-like edges in $I_{ver}(x, y)$ associated with the fish outline, a zone of pixels around the fish outline is set to zero. This is achieved by employing an erosion operation using a square structural element of size $[4.8\sigma_y \times 4.5\sigma_y]$.

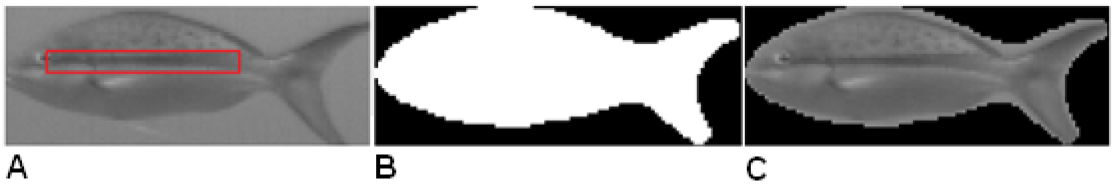

Figure 4.6: OC pre-processing steps.

A. Original image of OC with indicator for horizontal line. B. Region mask of OC. C. Pre-processed image of OC.

It can be observed in Fig. 4.7-A that the straight line is darker than the rest of the OC body. On the other hand, detecting the EM tail band is a maximization problem. For consistency with the EM technique, detecting the OC straight body line is converted to a maximization problem by subtracting the value of each pixel in the ratio image $I_r(x, y)$ from the maximum intensity of $I_r(x, y)$.

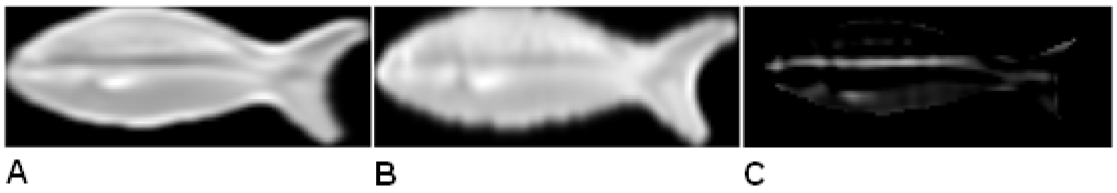

Figure 4.7: Filtering results of horizontal and vertical Gabor filters on OC.

A. Filtering result of horizontal Gabor filter. B. Filtering result of vertical Gabor filter. C. Ratio image $I_r(x, y)$.

A horizontal MA filter of size $1 \times W_{OC}$ is applied along the rows of $I_r(x, y)$. The OC straight line runs across the fish body and thus strongly depends on the fish size. For this reason, the value associated with W_{OC} is selected automatically, and is set to be proportional to the square root of the fish area. The maximum value for each row is computed as:

$$m_{I_r}^{OC}(x) = max_x \{f_{MA}(x) * I_r(x, y)\} \tag{4.3}$$

The maximum value of $m_{I_r}^{OC}(x)$, namely $m_{I_r}^{OC}(max) = max_x \left\{m_{I_r}^{OC}(y)\right\}$, quantifies the presence of the straight line in the fish region as shown in Fig. 4.8. When a fish is oriented at an angle with respect to x-axis as shown in Fig. 4.9-E, the straight line may not be exactly horizontal and

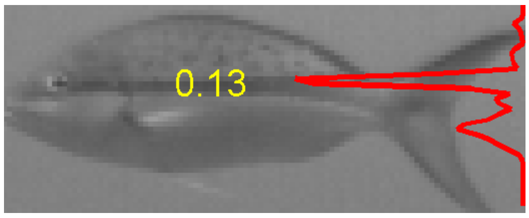

Figure 4.8: OC straight line detection – maximum value per row $m_{I_r}^{OC}(y)$.

may not be detected by the algorithm. For this reason, the original fish image, $OF_r(x, y)$, as shown in Fig. 4.9-E, is rotated by different angles, and the largest m_{Irmax}^{OC} is considered. From Fig. 4.9 it is evident that the plot corresponding to the maximum m_{Irmax}^{OC} is the one for which the OC line is horizontal. In this particular example, the image is rotated in steps of 10^0 from -40^0 to 40^0 and the largest m_{Irmax}^{OC} is obtained when the image is rotated at 10^0 as shown in Fig. 4.9-F. The fish orientation is not a significant issue for EM, since the tail band is relatively short and wide, as opposed to the OC straight line which is relative long and narrow.

The algorithm for detection and classification of OC fish species is shown in Fig. 4.10. The algorithm described earlier for the classification of EM species is similar to the algorithm presented in Fig. 4.10, if the iterative process involving the rotation of image and region mask is removed.

4.4 Results and Discussion

In this section, the performance of the algorithm is discussed in terms of detection and false alarm rates. The algorithm was tested on sequence of 200 images from an annual reef fish survey conducted by the SEFSC Pascagoula laboratory. The total number of EM, OC, and non-EM/non-OC fishes are provided in Table 4.1. The results for EM are summarized in Table 4.2. An EM fish is detected when the $m_{Ir}^{EM}(max)$ value is found to be between two thresholds. More specifically, $T_1^{EM} < m_{I_r}^{EM}(max) < T_2^{EM}$. The threshold values were empirically chosen

Total number of frames used for experimentation	200
Total number of EM available in total frames	68
Total number of OC available in total frames	66
Total number of non-EM and non-OC fish available in total frames	159
Total number of non-fish objects available in total frames	74

Table 4.1: Information about reef fish images obtained from SEFSC Pascagoula laboratory

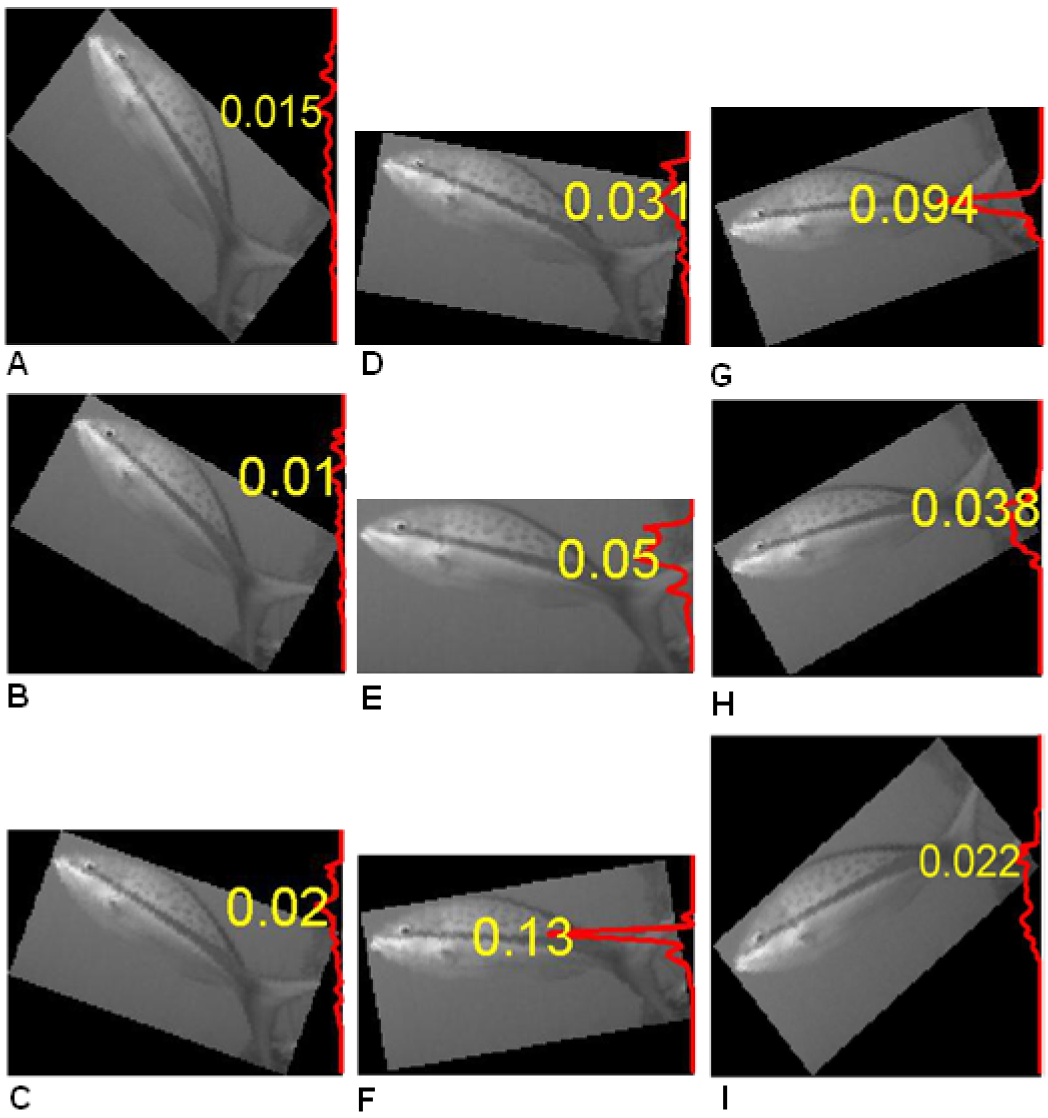

Figure 4.9: OC straight line detection when the fish is oriented at a different angle with respect to x axis.

(a) A. -40^0. B. -30^0. C. -20^0. D. -10^0 . E. 0^0 (which is also the original view of the OC fish). F. 10^0 . G. 20^0. H. 30^0. I. 40^0.

as $T_1^{EM} = 0.95$ and $T_2^{EM} = 2$. Any region with corresponding $m_{I_r}^{EM}(max)$ peak outside the range specified by the two thresholds is categorized as non-EM. Such a region may correspond to a non-EM fish or to a non-fish object. The purpose of the lower threshold, T_1^{EM}, is to separate regions which contain a significant vertical band from regions that do not include such a band. It was observed that in some cases non-fish regions included some apparent vertical stripes, which however result in exceptionally high $m_{I_r}^{EM}(max)$ peaks. The higher threshold, T_2^{EM}, is used to identify such non-fish regions.

As can be observed from Table 4.2, the total number of EM not detected by the proposed

EM	Correctly detected		48 (70.6%)
	Not Detected	Technique not successful (missed detection)	4 (5.9%)
		Due to non-visible tail band	16 (23.5%)
Non-EM	Other fish detected as EM (false alarm)		5 (2.1%)
	Other fish correctly not detected as EM		154 (66.1%)
	Non-fish objects detected as EM (false alarm)		4 (1.7%)
	Non-fish objects correctly not detected as EM		70 (30.0%)

Table 4.2: EM results

algorithm is 20, which corresponds to 29.4% of the EM cases. However, in the vast majority of missed cases (16 or 23.5%) the video frame itself has no or little information about the tail band. Two examples of such missed cases are shown in Fig. 4.11. Fig. 4.11-A depicts a case where the tail is positioned in a way that the band is only slightly visible, and Fig. 4.11-D shows an example where the fish is facing the camera. Function $m_{I_r}^{EM}(x)$ is depicted as a red curve, and the value shown indicates the maximum peak, $m_{I_r}^{EM}(max)$. Even human analysts, who may be capable of recognizing the fish as EM, may not be able to distinguish the tail band in these two examples.

On the other hand, the algorithm is capable of detecting EM even when fishes merge together. An example is illustrated in Fig. 4.11-C, where it is clearly shown that the $m_{I_r}^{EM}(max)$ peak is found at the tail band. Fig. 4.11-B shows a non-EM fish example, which is correctly detected as non-EM, since the fish does not have a tail stripe. According to the results presented in Table 4.2, the vast majority of the non-EM cases are correctly detected as non-EM. Only 3.8% of the non-EM cases are categorized as EM. In particular, 2.1% and 1.7% of the false alarm cases

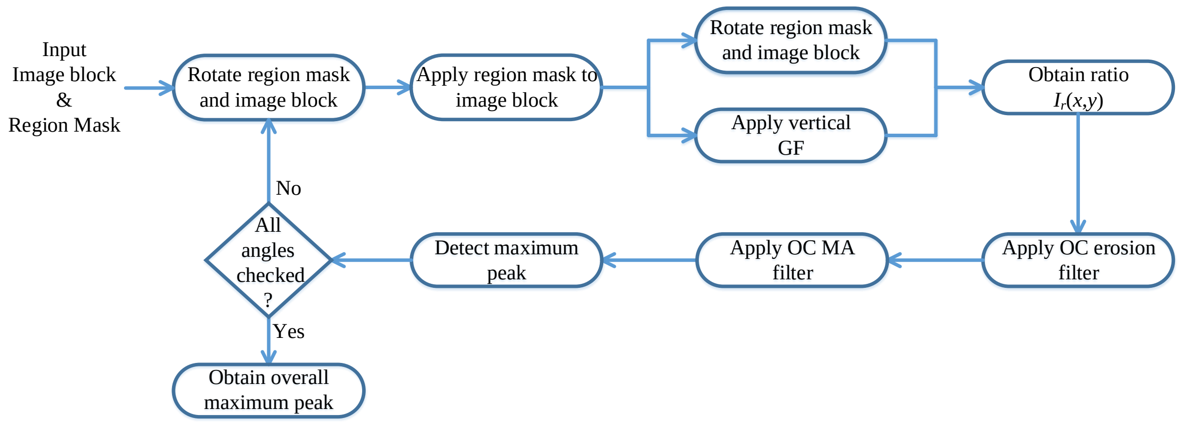

Figure 4.10: Proposed algorithm flowchart.

OC	Correctly detected		53 (80.3%)
	Not detected	Technique not successful (missed detection)	2 (3.0%)
		Due to non-visible horizontal line	11 (16.7%)
Non-OC	Other fish detected as OC (false alarm)		4 (1.7%)
	Other fish correctly not detected as OC		155 (66.5%)
	Non-fish objects detected as OC (false alarm)		1 (0.4%)
	Non-fish objects correctly not detected as OC		73 (31.3%)

Table 4.3: OC results

correspond to other fish and non-fish regions, respectively.

The results for OC are summarized in Table 4.3. Similar to the EM case, an OC fish is detected when the $m_{I_r}^{OC}(max)$ value is found to be between two thresholds. In other words, $T_1^{OC} < m_{I_r}^{OC}(max) < T_2^{OC}$. The threshold values were empirically chosen as $T_1^{OC} = 0.06$ and $T_2^{OC} = 0.2$. Any region with corresponding $m_{I_r}^{OC}(max)$ peak outside the range specified by the two thresholds is categorized as non-OC. Regions with associated $m_{I_r}^{OC}(max)$ peak below the lower threshold, T_1^{OC}, do not include a significant horizontal line. The higher threshold, T_2^{OC}, is used for the same reason as in the case of EM.

As can be observed from Table 4.3, the total number of OC not detected by the proposed algorithm is 13, which corresponds to 19.7% of the OC cases. As in the case of EM, for the majority of missed cases (11 or 16.7%) the video frame itself has no or little information about the straight line. Two examples where the straight line is not clearly visible are presented in Fig. 4.11-E and Fig. 4.11-G. Fig. 4.11-F shows a non-OC fish example, which is correctly detected as non-OC, since the fish does not have a horizontal line. Function $m_{I_r}^{OC}(y)$ is depicted as a red curve, and the value shown indicates the maximum peak, $m_{I_r}^{OC}(max)$. The proposed algorithm detected a total of 5 false alarms (i.e., non-OC regions detected as OC), which is only 2.1% of the non-OC cases. In particular, 1.7% and 0.4% of the false alarm cases correspond to other fish and non-fish regions, respectively. Fig. 4.12 illustrates a few examples where non-EM/non-OC species are detected as EM or OC. Fig. 4.13 presents a few examples where non-fish objects are detected as EM or OC. Some examples where the proposed algorithm missed detection of EM and OC are shown in Fig. 4.14.

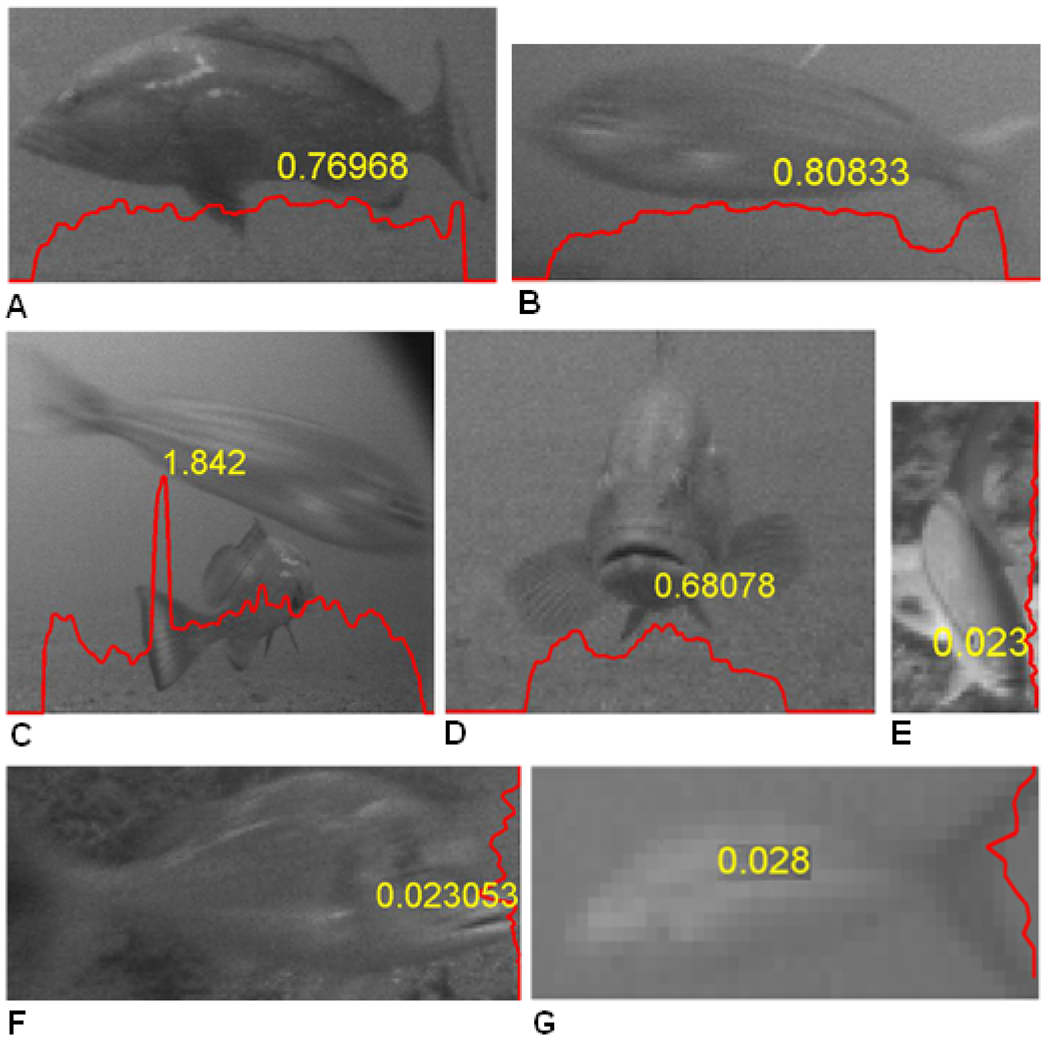

Figure 4.11: Illustration of some EM and OC results

A. EM tail is positioned in a way that the tail band is only slightly visible. B. As expected, the EM fish is detected as non-EM since the tail band is not visible. C. EM tail band is detected even though the two fishes merged. D. As anticipated, the EM fish is detected as non-EM since the fish is facing the camera and the tail band is not visible. E. The OC fish is not detected since the straight line is not clearly visible. F. As anticipated, the OC fish is detected as non-OC since the fish does not appear to have a straight line. G. As expected, the OC fish is not detected since the straight line is not visible, mainly due to the low image resolution.

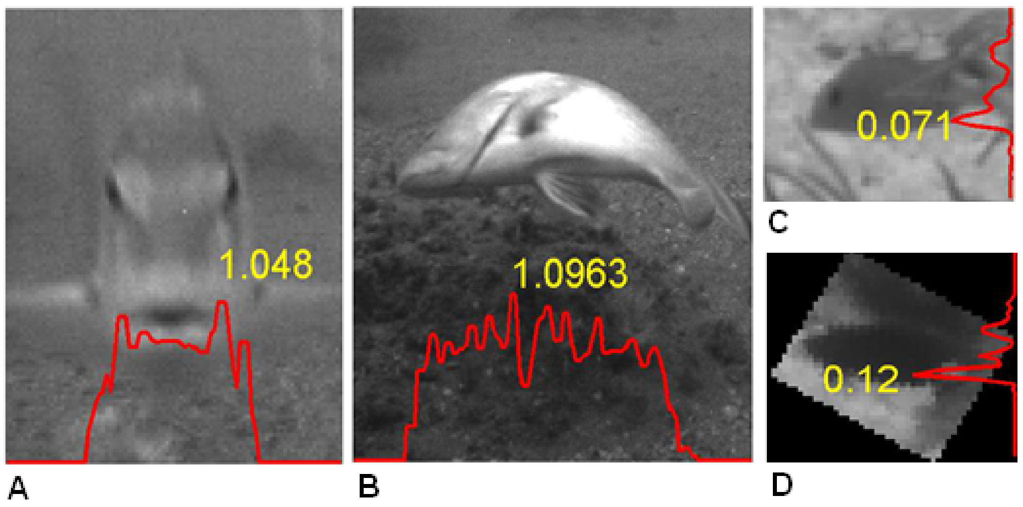

Figure 4.12: Illustration of few cases of false alarms (non-EM detected as EM, or non-OC detected as OC)

A. Fish detected as EM although there is no visible tail band information. B. Fish detected as EM although there is no visible tail band. C. Fish detected as OC although there is no clear visibility of straight body line. D. Fish detected as OC although there is no clear visibility of straight body line.

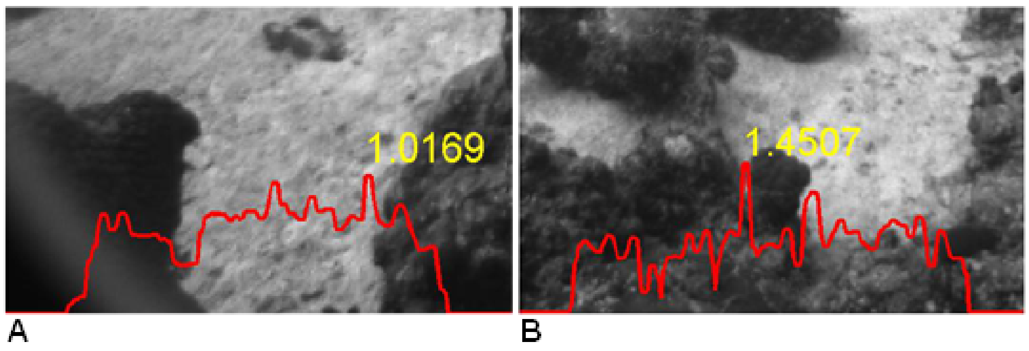

Figure 4.13: Illustration of few cases of false alarms (non-fish detected as EM or OC)

A. The algorithm detected non-fish as EM. B. The algorithm detected non-fish as EM.

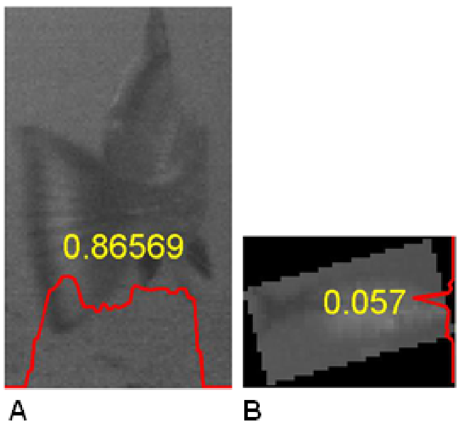

Figure 4.14: Illustration of few cases where EM and OC are not detected by the algorithm.

A. The technique is unsuccessful in detecting EM although the tail band is clearly visible. B. The technique failed in detecting OC although the straight line is clearly visible.

4.5 Conclusions

A technique based on GFs to effectively extract species-specific features from EM and OC fish species is presented. These features are used for detecting EM and OC fish species. The detection rate is 70.6% for EM and 80.3% for OC, while 23.5% of the undetected EM cases do not have a visible tail band, and 16.7% of the undetected OC cases do not have a visible straight body line. Therefore, missed detection for these cases is expected. The false alarm rates are only 3.8% and 2.1%, respectively. These results are promising and indicate that these features may complement other feature extraction techniques for the purpose of fish classification. Additionally, target tracking may be used to associate the same fish region in different frames, so that the whole fish sequence is classified as a single species, even if detection is successful in only one of the frames. Even if false alarms occur, automatic detection significantly reduces the amount of data to be examined by human analysts.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

An attempt to optimize the convolution operation is made through the proposed method based on FPGA in combination with separable kernel filters. The results obtained prove that the proposed method achieves the objective of optimizing the convolution operation by providing the necessary balance between on-chip resource utilization and external memory bandwidth. Thus, the proposed method improves the performance of filtering process while lowering the cost, operation time, and utilization of resources.

The proposed method for error reduction in logarithmic multiplication is based on Mitchell's method. In the proposed method the filters weights are altered which resulted in significant reduction of product error without any increase in the resource utilization other than the Mitchell multiplier. The average error percentage for the Mitchell algorithm in [7] is 3.87%, OD in [30] is between 2.07% and 2.15%, and the IM algorithm with 2 stages is between 0.83% and 0.99%. The average error percentage for the proposed logarithmic multiplier is around 0.1%. Hence, the proposed method for logarithmic multiplication is best suited for filtering applications where the filter weights are known and fixed. The proposed method extension for Mitchell further reduces the error without any increase in the resource utilization other than the Mitchell multiplier. In the case of the proposed method extension for IM (i.e. "IM-Mitchell-Proposed-Extension"), error is further reduced at a cost of few extra resources which are essential to store the true $log2$ values of the decomposed operands.

A technique based on GFs is proposed to effectively extract species-specific fish features from EM and OC fish species with a detection rate of 70.6% for EM and 80.3% for OC. The false alarm rates are as low as 3.8% and 2.1% respectively. The proposed method on species-specific fish feature extraction using Gabor filters also automates the species detection process, thus eliminating the need for manual analysis which is laborious as well as error prone. Hence, the proposed method is one of the best alternatives for species-specific fish feature extraction which greatly reduces the probability of error occurrence in the detection process.

5.2 Future Work

The dissertation research work point to several interesting directions for future work which are as follows:

1. It can be observed from Table 2.2 and Fig. 2.4 that the resource utilization for storing the intermediate results increase exponentially depending on the size of the mask. For instance, if the mask size is $K \times K$, then K^2 registers are required to store the intermediate results. A solution to reduce the exponential increase of resources for storing the intermediate results is to store them in an LUT. An LUT has n bits and provides 2^n output bits. Hence, intermediate results can be stored in an LUT which aids in reducing the resource utilization increase from exponential to linear.

2. The increase in memory and resource utilization for storing the true $log2$ values of the original operand and the decomposed operand in the proposed method extension for IM can be optimized by using an 8 bit state diagram. The scope of future work includes average error analysis and FPGA implementation of the proposed method extension for IM.

3. The proposed Gabor filter for fish feature extraction can be further improved for real time implementation using FPGA. The proposed FPGA architecture for separable convolution and logarithmic multiplication can be used as basic building blocks for implementation of the Gabor filter using FPGA for fish feature extraction.

Bibliography

[1] J. Tripp, H. Mortveit, A. Hansson, and M. Gokhale, "Metropolitan road traffic simulation on fpgas," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005)*, April 2005, pp. 117–126.

[2] E. Shang, J. Li, X. An, and H. He, "Lane detection using steerable filters and fpga-based implementation," in *Sixth International Conference on Image and Graphics (ICIG)*, Aug 2011, pp. 908–913.

[3] A. E. Nelson, "Implementation of image processing algorithms on fpga hardware," Ph.D. dissertation, Vanderbilt University, 2000.

[4] S. Sowmya and R. Paily, "Fpga implementation of image enhancement algorithms," in *International Conference on Communications and Signal Processing (ICCSP)*, Feb 2011, pp. 584–588.

[5] A. Joginipelly, A. Varela, D. Charalampidis, R. Schott, and Z. Fitzsimmons, "Efficient fpga implementation of steerable gaussian smoothers," in *44th Southeastern Symposium on System Theory (SSST)*, March 2012, pp. 78–82.

[6] T. R. Reed and J. M. H. du Buf, "A review of recent texture segmentation and feature extraction techniques," *CVGIP: Image Underst.*, vol. 57, no. 3, pp. 359–372, May 1993.

[7] J. N. Mitchell, "Computer multiplication and division using binary logarithms," *IRE Transactions on Electronic Computers*, vol. 11, no. 4, pp. 512–517, Aug 1962.

[8] B. Bosi, G. Bois, and Y. Savaria, "Reconfigurable pipelined 2-d convolvers for fast digital signal processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 3, pp. 299–308, Sept 1999.

[9] F. Cardells-Tormo and P.-L. Molinet, "Area-efficient 2-d shift-variant convolvers for fpga-based digital image processing," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 53, no. 2, pp. 105–109, Feb 2006.

[10] H. Zhang, M. Xia, and G. Hu, "A multiwindow partial buffering scheme for fpga-based 2-d convolvers," *IEEE Transactions on Circuits and Systems II: Express Brief*, vol. 54, no. 2, pp. 200–204, Feb 2007.

[11] S. Carlo, G. Gambardella, M. Indaco, D. Rolfo, G. Tiotto, and P. Prinetto, "An area-efficient 2-d convolution implementation on fpga for space applications," in *IEEE 6th International Design and Test Workshop (IDT)*, Dec 2011, pp. 88–92.

[12] D. Charalampidis, "Efficient directional gaussian smoothers," *IEEE Geoscience and Remote Sensing Letters*, vol. 6, no. 3, pp. 383–387, July 2009.

[13] F. Cardells-Tormo and J. Arnabat-Benedicto, "Flexible hardware-friendly digital architecture for 2-d separable convolution-based scaling," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 53, no. 7, pp. 522–526, July 2006.

[14] M. Al-Mistarihi, "Separable implementation of the second order volterra filter (sovf) in xilinx virtex-e fpga," in *International Conference on Field Programmable Logic and Applications*, Sept 2008, pp. 531–534.

[15] C.-S. Bouganis, S.-B. Park, G. A. Constantinides, and P. Y. K. Cheung, "Synthesis and optimization of 2d filter designs for heterogeneous fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 1, no. 4, pp. 24:1–24:28, jan 2009.

[16] J. Mori, C. Llanos, and P. Berger, "Kernel analysis for architecture design trade off in convolution-based image filtering," in *25th Symposium on Integrated Circuits and Systems Design (SBCCI)*, Aug 2012, pp. 1–6.

[17] G. C. Jung, S. M. Park, and J. H. Kim, "Efficient vlsi architectures for convolution and lifting based 2-d discrete wavelet transform," in *Proceedings of the 10th Asia-Pacific Conference on Advances in Computer Systems Architecture*, 2005, pp. 795–804.

[18] M. Z. Zhang and V. K. Asari, "An efficient multiplier-less architecture for 2-d convolution with quadrant symmetric kernels," *Integr. VLSI J.*, vol. 40, no. 4, pp. 490–502, Jul. 2007.

[19] S. Y. Eun and M. Sunwoo, "An efficient 2-d convolver chip for real-time image processing," in *Proceedings of Asia and South Pacific Design Automation Conference*, Feb 1998, pp. 329–330.

[20] P. Meher, "New approach to look-up-table design and memory-based realization of fir digital filter," *IEEE Transactions on Circuits and Systems I: Regular Paper*, vol. 57, no. 3, pp. 592–603, March 2010.

[21] G. Deepak, R. Mahesh, and A. Sluzek, "Design of an area-efficient multiplierless processing element for fast two dimensional image convolution," in *13th IEEE International Conference on Electronics, Circuits and Systems*, Dec 2006, pp. 467–470.

[22] A. Yurdakul, "Multiplierless implementation of 2-d fir filters," *Integration, the VLSI Journal*, vol. 38, no. 4, pp. 597 – 613, 2005.

[23] P. Meher, "Parallel and pipelined architectures for cyclic convolution by block circulant formulation using low-complexity short-length algorithms," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 18, no. 10, pp. 1422–1431, Oct 2008.

[24] B. Krill and A. Amira, "Efficient reconfigurable architectures of generic cyclic convolution," in *15th IEEE International Symposium on Consumer Electronics (ISCE)*, June 2011, pp. 560–564.

[25] R. Turney, "Two-dimensional linear filtering," 2007. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp933.pdf

[26] B. Mohanty and P. Meher, "New scan method and pipeline architecture for vlsi implementation of separable 2-d fir filters without transposition," in *TENCON IEEE Region 10 Conference*, Nov 2008, pp. 1–5.

[27] [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/mult_gen_ds255.pdf

[28] [Online]. Available: http://www.digilentinc.com/Data/Products/GENESYS/Genesys_rm.pdf

[29] A. Cosoroaba, "Memory interfaces made easy with xilinx fpgas and the memory interface generator," 2007. [Online]. Available: http://ece545.com/F12/resources/Virtex5/wp260.pdf

[30] V. Mahalingam and N. Ranganathan, “Improving accuracy in mitchell’s logarithmic multiplication using operand decomposition,” *IEEE Transactions on Computers*, vol. 55, no. 12, pp. 1523–1535, Dec 2006.

[31] H. Ngo and V. Asari, “Design of a logarithmic domain 2-d convolver for low power video processing applications,” in *Sixth International Conference on Information Technology: New Generations*, April 2009, pp. 1280–1285.

[32] Z. Babic, A. Avramovic, and P. Bulic, “An iterative logarithmic multiplier,” *Microprocessors and Microsystems*, vol. 35, no. 1, pp. 23 – 33, Feb 2011.

[33] P. Bulic, Z. Babic, and A. Avramovic, “Digital signal processing applications with iterative logarithmic multipliers,” *Journal of Information Technology and Applications*, vol. 1, pp. 77 – 148, Dec 2011.

[34] D. Mclaren, “Improved mitchell-based logarithmic multiplier for low-power dsp applications,” in *IEEE International Conference on Systems-on-Chip*, Sept 2003, pp. 53–56.

[35] S. Paul, N. Jayakumar, and S. Khatri, “A fast hardware approach for approximate, efficient logarithm and antilogarithm computations,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 2, pp. 269–277, Feb 2009.

[36] R. Gutierrez and J. Valls, “Low cost hardware implementation of logarithm approximation,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 12, pp. 2326–2330, Dec 2011.

[37] E. L. Hall, D. Lynch, and I. Dwyer, S.J., “Generation of products and quotients using approximate binary logarithms for digital filtering applications,” *IEEE Transactions on Computers*, vol. 19, no. 2, pp. 97–105, Feb 1970.

[38] M. Combet, H. Van Zonneveld, and L. Verbeek, “Computation of the base two logarithm of binary numbers,” *IEEE Transactions on Electronic Computers*, vol. 14, no. 6, pp. 863–867, Dec 1965.

[39] S. SanGregory, C. Brothers, D. Gallagher, and R. Siferd, “A fast, low-power logarithm approximation with cmos vlsi implementation,” in *42nd Midwest Symposium on Circuits and Systems*, vol. 1, 1999, pp. 388–391.

[40] K. Abed and R. Siferd, “Cmos vlsi implementation of a low-power logarithmic converter,” *IEEE Transactions on Computers*, vol. 52, no. 11, pp. 1421–1433, Nov 2003.

[41] R. Siferd and K. Abed, “Vlsi implementation of a low-power antilogarithmic converter,” *IEEE Transactions on Computers*, vol. 52, no. 9, pp. 1221–1228, Sept 2003.

[42] T.-B. Juang, S.-H. Chen, and H.-J. Cheng, “A lower error and rom-free logarithmic converter for digital signal processing applications,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 56, no. 12, pp. 931–935, Dec 2009.

[43] M. Y. Kong, J. M. P. Langlois, and D. Al-Khalili, “Efficient fpga implementation of complex multipliers using the logarithmic number system,” in *IEEE International Symposium on Circuits and Systems*, May 2008, pp. 3154–3157.

[44] D. De Caro, N. Petra, and A. Strollo, “Efficient logarithmic converters for digital signal processing applications,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 58, no. 10, pp. 667–671, Oct 2011.

[45] R. Selina, "Vlsi implementation of piecewise approximated antilogarithmic converter," in *International Conference on Communications and Signal Processing (ICCSP)*, April 2013, pp. 763–766.

[46] [Online]. Available: http://www.xilinx.com/support/documentation/boards_and_kits/ug130.pdf,

[47] M. Cappo, E. Harvey, and M. Shortis, "Counting and measuring fish with baited video techniques-an overview," *Proceedings of Australian Society for Fish Biology Workshop*, pp. 101–114, 2006.

[48] G. Boynton and K. Voss, "An underwater digital stereo video camera for fish population assessment," Physics Department, University of Miami, Coral Gables, Florida, Tech. Rep., 2006. [Online]. Available: http://data.nodc.noaa.gov/coris/library/NOAA/CRCP/project/1339/UnderwaterDigitalStereoVideoSystem_FishPopAssess.pdf

[49] C. Spampinato, D. Giordano, R. Di Salvo, Y.-H. J. Chen-Burger, R. B. Fisher, and G. Nadarajan, "Automatic fish classification for underwater species behavior understanding," in *Proceedings of the First ACM International Workshop on Analysis and Retrieval of Tracked Events and Motion in Imagery Streams*, 2010, pp. 45 – 50.

[50] J. D. Wilder, "System integration and image pre-processing for an automated real-time identification and monitoring system for coral reef fish," Master's thesis, The State University of New Jersey, Oct 2010.

[51] H. J. Williams K, Rooper C, "Report of the national marine fisheries service workshop on underwater video analysis," U.S. Dep. Commerce, NOAA Tech. Memo.NMFS-F/SPO-121, Tech. Rep., 2012. [Online]. Available: http://www.pifsc.noaa.gov/library/pubs/tech/NOAA_TM_NMFS_F-SPO_121.pdf

[52] X. Li, K. Wang, W. Wang, and Y. Li, "A multiple object tracking method using kalman filter," in *IEEE International Conference on Information and Automation (ICIA)*, June 2010, pp. 1862–1866.

[53] I. Fogel and D. Sagi, "Gabor filters as texture discriminator," *Biological Cybernetics*, vol. 61, no. 2, pp. 103–113, 1989.

[54] V. S. Vyas and P. Rege, "Automated texture analysis with gabor filter," *International Journal on Graphics, Vision and Image Processing*, vol. 6, July 2006.

[55] G. T. Shrivakshan and D. C. Ch, "Detecting the age of the fish through image processing using its morphological features," *International Journal of Computer Science and Information Technologies*, vol. 2, pp. 2562–2567, 2011.

[56] S. Cadieux, F. Michaud, and F. Lalonde, "Intelligent system for automated fish sorting and counting," in *IEEE International Conference on Intelligent Robots and Systems*, vol. 2, Nov 2000, pp. 1279–1284.

[57] M. Piccardi, "Background subtraction techniques: a review," in *IEEE International Conference on Systems, Man and Cybernetics*, vol. 4, Oct 2004, pp. 3099–3104.

[58] B. Han and L. Davis, "Density-based multifeature background subtraction with support vector machine," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 5, pp. 1017–1023, May 2012.

List of Codes

Appendices

VHDL Source Codes

Code 1: Input Image Read Controller

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use ieee.numeric_std.all;
entity spbram_inputimage_rd_controller is
    Port ( clk_system : in  STD_LOGIC;
        reset : in std_logic;
        enable : in std_logic;
            addra : out  STD_LOGIC_VECTOR (14 downto 0);
            dina : out STD_LOGIC_VECTOR (7 downto 0);
            wea : out  STD_LOGIC;
            ena : out  STD_LOGIC;
            clk_out : out  STD_LOGIC);
end spbram_inputimage_rd_controller;
architecture Behavioral of spbram_inputimage_rd_controller
      is
type state_reg_type is (initialstate , rdstate ,halt);
signal sreg:state_reg_type;
signal addra_sig:std_logic_vector(14 downto 0);
signal acount:std_logic_vector(14 downto 0);
begin
clk_out<=clk_system;
  process(clk_system,reset)
  begin
    if(clk_system'event and clk_system='0') then
      if(reset='1') then
        ena <='0';
      elsif (enable = '1') then
        ena <= '0';
      else
        case sreg is
          when initialstate=>  ena<='1';
                              wea<='0';
                              addra_sig<=(others=>'0');
                              acount<="000000000000001";
                              sreg<=rdstate;
          when rdstate=> ena<='1';
                        addra_sig<=acount;
                        acount<=acount+1;
                        if(acount=28379) then
                          sreg<=halt;
                        else
                          sreg<=rdstate;
                        end if;
          when halt=> wea<='0';
                    ena<='0';
        end case;
      end if;
    end if;
  end process;
addra<=addra_sig;
end Behavioral;
```

Code 2: Shift Register Module 1

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;
entity regmod1 is
    Port ( clk_system : in  STD_LOGIC;
            reset : in  STD_LOGIC;
            douta : in  STD_LOGIC_VECTOR (7 downto 0);
            r1 : out  STD_LOGIC_VECTOR (7 downto 0);
            r2 : out  STD_LOGIC_VECTOR (7 downto 0);
            r3 : out  STD_LOGIC_VECTOR (7 downto 0));
end regmod1;
architecture Behavioral of regmod1 is
signal r1_sig:std_logic_vector(7 downto 0);
signal r2_sig:std_logic_vector(7 downto 0);
signal r3_sig:std_logic_vector(7 downto 0);
begin
  process(clk_system)
  begin
  if(clk_system'event and clk_system='1') then
    if(reset='0') then
      r3_sig<=douta;
      r2_sig<=r3_sig;
      r1_sig<=r2_sig;
    else
      r1_sig<=(others=>'0');
      r2_sig<=(others=>'0');
      r3_sig<=(others=>'0');
    end if;
  end if;
end process;
r1<=r1_sig;
r2<=r2_sig;
r3<=r3_sig;
end Behavioral;
```

Code 3: Central Selection Controller

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use ieee.numeric_std.all;
entity central_selection_controller is
    Port ( clk_system : in STD_LOGIC;
           reset : in STD_LOGIC;
           bram_en : out STD_LOGIC;
           mux1_sel : out STD_LOGIC_VECTOR (4 downto 1);
           demux2_sel : out STD_LOGIC;
           reg3_sel : out STD_LOGIC);
end central_selection_controller;
architecture Behavioral of central_selection_controller is
signal count : std_logic_vector(2 downto 0);
signal bram_en_sig:std_logic;
signal mux1_sel_sig:std_logic_vector(4 downto 1);
signal demux2_sel_sig:std_logic;
signal reg3_sel_sig:std_logic;
begin
  process(clk_system)
  begin
    if(clk_system'event and clk_system='1') then
      if(reset='0') then
        count<=count+1;
        if(count=6) then
          count<="001";
        end if;
        if(count=5) then
          bram_en_sig<='1';
        elsif(count=6) then
          bram_en_sig<='0';
        end if;
        if(count=3) then
          mux1_sel_sig(1)<='0';
        elsif(count=6) then
          mux1_sel_sig(1)<='1';
        end if;

        if(count=3) then
          mux1_sel_sig(2)<='1';
        elsif(count=4) then
          mux1_sel_sig(2)<='0';
        end if;
        mux1_sel_sig(3)<=mux1_sel_sig(2);
        mux1_sel_sig(4)<=mux1_sel_sig(3);
        if(count=3) then
          demux2_sel_sig<='0';
        elsif(count=6) then
          demux2_sel_sig<='1';
        end if;
        if(count=3) then
          reg3_sel_sig<='0';
        elsif(count=6) then
          reg3_sel_sig<='1';
        end if;
      else
         count<=(others=>'0');
         bram_en_sig<='0';
         mux1_sel_sig(1)<='1';
         mux1_sel_sig(2)<='0';
         mux1_sel_sig(3)<='0';
         mux1_sel_sig(4)<='0';
         demux2_sel_sig<='1';
         reg3_sel_sig<='1';
      end if;
    end if;
  end process;
bram_en<=bram_en_sig;
mux1_sel<=mux1_sel_sig;
demux2_sel<=demux2_sel_sig;
reg3_sel<=reg3_sel_sig;
end Behavioral;
```

Code 4: Multiplexer 1

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use ieee.numeric_std.all;
entity multipliexer1 is
    Port ( clk_system : in STD_LOGIC;
           reset : in STD_LOGIC;
           r1 : in STD_LOGIC_VECTOR (7 downto 0);
           r2 : in STD_LOGIC_VECTOR (7 downto 0);
           r3 : in STD_LOGIC_VECTOR (7 downto 0);
           vr1 : in STD_LOGIC_VECTOR (7 downto 0);
           vr2 : in STD_LOGIC_VECTOR (7 downto 0);
           vr3 : in STD_LOGIC_VECTOR (7 downto 0);
           vr4 : in STD_LOGIC_VECTOR (7 downto 0);
           vr5 : in STD_LOGIC_VECTOR (7 downto 0);
           vr6 : in STD_LOGIC_VECTOR (7 downto 0);
           vr7 : in STD_LOGIC_VECTOR (7 downto 0);
           vr8 : in STD_LOGIC_VECTOR (7 downto 0);
           vr9 : in STD_LOGIC_VECTOR (7 downto 0);
           mux1_sel : in STD_LOGIC_VECTOR (4 downto 1);
           o1 : out STD_LOGIC_VECTOR (7 downto 0);
           o2 : out STD_LOGIC_VECTOR (7 downto 0);
           o3 : out STD_LOGIC_VECTOR (7 downto 0));
end multipliexer1;

```

```vhdl
architecture Behavioral of multipliexer1 is
begin
  process(clk_system)
  begin
    if(clk_system'event and clk_system='1') then
      if(reset='0') then
        case mux1_sel is
          when "0001" =>  o1<=r1; --1--
                          o2<=r2;
                          o3<=r3;
          when "0010" =>  o1<=vr1;--2--
                          o2<=vr4;
                          o3<=vr7;
          when "0100" =>  o1<=vr2;--4--
                          o2<=vr5;
                          o3<=vr8;
          when "1000" =>  o1<=vr3;--8--
                          o2<=vr6;
                          o3<=vr9;
          when others=> o1<=(others=>'0');
                        o2<=(others=>'0');
        end case;
      else
        o1<=(others=>'0');
        o2<=(others=>'0');
        o3<=(others=>'0');
      end if;
    end if;
  end process;
end Behavioral;
```

Code 5: Multiplier

```vhdl
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;
library UNISIM;
use UNISIM.Vcomponents.ALL;

entity multiplier_adder is
   port ( ce          : in    std_logic;
          clk_system  : in    std_logic;
          m1a         : in    std_logic_vector (7 downto 0);
          m1b         : in    std_logic_vector (7 downto 0);
          m2a         : in    std_logic_vector (7 downto 0);
          m2b         : in    std_logic_vector (7 downto 0);
          m3a         : in    std_logic_vector (7 downto 0);
          m3b         : in    std_logic_vector (7 downto 0);
          reset       : in    std_logic;
          result      : out   std_logic_vector (7 downto 0));
end multiplier_adder;

architecture BEHAVIORAL of multiplier_adder is
   attribute BOX_TYPE   : string ;
   signal XLXN_18    : std_logic_vector (15 downto 0);
   signal XLXN_19    : std_logic_vector (15 downto 0);
   signal XLXN_20    : std_logic_vector (15 downto 0);
   signal XLXN_21    : std_logic;
   component multiplier1
      port ( a   : in    std_logic_vector (7 downto 0);
             b   : in    std_logic_vector (7 downto 0);
             clk : in    std_logic;
             ce  : in    std_logic;
             p   : out   std_logic_vector (15 downto 0));
   end component;

   component finaladder
      port ( clk_system  : in    std_logic;
             reset       : in    std_logic;
             p1          : in    std_logic_vector (15 downto 0);
             p2          : in    std_logic_vector (15 downto 0);
             p3          : in    std_logic_vector (15 downto 0);
             result      : out   std_logic_vector (7 downto 0));
   end component;

   component INV
      port ( I : in    std_logic;
             O : out   std_logic);
   end component;
   attribute BOX_TYPE of INV : component is "BLACK_BOX";

begin
   XLXI_1 : multiplier1
      port map (a(7 downto 0)=>m1a(7 downto 0),
                b(7 downto 0)=>m1b(7 downto 0),
                ce=>ce,
                clk=>clk_system,
                p(15 downto 0)=>XLXN_18(15 downto 0));

   XLXI_2 : multiplier1
      port map (a(7 downto 0)=>m2a(7 downto 0),
                b(7 downto 0)=>m2b(7 downto 0),
                ce=>ce,
                clk=>clk_system,
                p(15 downto 0)=>XLXN_19(15 downto 0));

   XLXI_3 : multiplier1
      port map (a(7 downto 0)=>m3a(7 downto 0),
                b(7 downto 0)=>m3b(7 downto 0),
                ce=>ce,
                 clk=>clk_system,
                 p(15 downto 0)=>XLXN_20(15 downto 0));

   XLXI_4 : finaladder
      port map (clk_system=>XLXN_21,
                 p1(15 downto 0)=>XLXN_18(15 downto 0),
                 p2(15 downto 0)=>XLXN_19(15 downto 0),
                 p3(15 downto 0)=>XLXN_20(15 downto 0),
                 reset=>reset,
                 result(7 downto 0)=>result(7 downto 0));

   XLXI_5 : INV
      port map (I=>clk_system,
                 O=>XLXN_21);

end BEHAVIORAL;
```

Code 6: Filter Weight Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use ieee.numeric_std.all;
entity weightmodule is
    Port ( clk_system : in STD_LOGIC;
           reset : in STD_LOGIC;
        w1 : out STD_LOGIC_VECTOR (7 downto 0);
           w2 : out STD_LOGIC_VECTOR (7 downto 0);
        w3 : out STD_LOGIC_VECTOR (7 downto 0));
end weightmodule;
architecture Behavioral of weightmodule is
begin
  process(clk_system)
  begin
    if(clk_system'event and clk_system='1') then
      if(reset='0') then
        w1<="00000001";--1--
        w2<="00000001";--1--
        w3<="00000001";--1--
      else
        w1<=(others=>'0');
        w2<=(others=>'0');
        w3<=(others=>'0');
      end if;
    end if;
  end process;
end Behavioral;
```

Code 7: Shift Register Module 2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use ieee.numeric_std.all;
entity regmod3 is
    Port ( clk_system : in STD_LOGIC;
           reset : in STD_LOGIC;
           reg3_sel : in STD_LOGIC;
           vert_result : in STD_LOGIC_VECTOR (7 downto 0)
     ;
           vr1 : out STD_LOGIC_VECTOR (7 downto 0);
           vr2 : out STD_LOGIC_VECTOR (7 downto 0);
           vr3 : out STD_LOGIC_VECTOR (7 downto 0);
           vr4 : out STD_LOGIC_VECTOR (7 downto 0);
           vr5 : out STD_LOGIC_VECTOR (7 downto 0);
           vr6 : out STD_LOGIC_VECTOR (7 downto 0);
           vr7 : out STD_LOGIC_VECTOR (7 downto 0);
           vr8 : out STD_LOGIC_VECTOR (7 downto 0);
           vr9 : out STD_LOGIC_VECTOR (7 downto 0));
end regmod3;
architecture Behavioral of regmod3 is
signal vr1_sig:std_logic_vector(7 downto 0);
signal vr2_sig:std_logic_vector(7 downto 0);
signal vr3_sig:std_logic_vector(7 downto 0);
signal vr4_sig:std_logic_vector(7 downto 0);
signal vr5_sig:std_logic_vector(7 downto 0);
signal vr6_sig:std_logic_vector(7 downto 0);
signal vr7_sig:std_logic_vector(7 downto 0);
signal vr8_sig:std_logic_vector(7 downto 0);
signal vr9_sig:std_logic_vector(7 downto 0);
begin
  process(clk_system)
  begin
    if(clk_system'event and clk_system='1') then
      if(reset='0') then
        if(reg3_sel='1') then
          vr9_sig<=vert_result;
          vr8_sig<=vr9_sig;
          vr7_sig<=vr8_sig;
          vr6_sig<=vr7_sig;
          vr5_sig<=vr6_sig;
          vr4_sig<=vr5_sig;
          vr3_sig<=vr4_sig;
          vr2_sig<=vr3_sig;
          vr1_sig<=vr2_sig;
        end if;
      else
        vr1_sig<=(others=>'0');
        vr2_sig<=(others=>'0');
        vr3_sig<=(others=>'0');
        vr4_sig<=(others=>'0');
        vr5_sig <=(others=>'0');
        vr6_sig <=(others=>'0');
        vr7_sig <=(others=>'0');
        vr8_sig <=(others=>'0');
        vr9_sig <=(others=>'0');
      end if;
    end if;
  end process;
vr1<=vr1_sig;
vr2<=vr2_sig;
vr3<=vr3_sig;
vr4<=vr4_sig;
vr5<=vr5_sig;
vr6<=vr6_sig;
vr7<=vr7_sig;
vr8<=vr8_sig;
vr9<=vr9_sig;
end Behavioral;
```

Code 8: De-Multiplexer Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use ieee.numeric_std.all;
entity demultiplexer2 is
    Port ( clk_system : in STD_LOGIC;
           reset : in STD_LOGIC;
           demux2_sel : in STD_LOGIC;
           result : in STD_LOGIC_VECTOR (7 downto 0);
           vert_result : out STD_LOGIC_VECTOR (7 downto 0);
           hort_result : out STD_LOGIC_VECTOR (7 downto 0));
end demultiplexer2;
architecture Behavioral of demultiplexer2 is
begin
  process(clk_system)
  begin
    if(clk_system'event and clk_system='1') then
      if(reset='0') then
        if(demux2_sel='1') then
          vert_result <=result;
          hort_result <=(others=>'0');
        else
          hort_result <=result;
          vert_result <=(others=>'0');
        end if;
      else
        vert_result <=(others=>'0');
        hort_result <=(others=>'0');
      end if;
    end if;
  end process;
end Behavioral;
```

Code 9: Adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use ieee.numeric_std.all;
entity finaladder is
    Port ( clk_system : in STD_LOGIC;
           reset : in STD_LOGIC;
           p1 : in STD_LOGIC_VECTOR (15 downto 0);
           p2 : in STD_LOGIC_VECTOR (15 downto 0);
           p3 : in STD_LOGIC_VECTOR (15 downto 0);
           result : out STD_LOGIC_VECTOR (7 downto 0));
end finaladder;
architecture Behavioral of finaladder is
signal p1_sig:std_logic_vector(17 downto 0);
signal p2_sig:std_logic_vector(17 downto 0);
signal p3_sig:std_logic_vector(17 downto 0);
signal result_sig:std_logic_vector(17 downto 0);
begin
  process(clk_system)
  begin
    if(clk_system'event and clk_system='1') then
      if(reset='0') then
        p1_sig<="00" & p1;
        p2_sig<="00" & p2;
        p3_sig<="00" & p3;
        result_sig <=p1_sig+p2_sig+p3_sig;
      else
        p1_sig <= (others=>'0');
        p2_sig <= (others=>'0');
        p3_sig <= (others=>'0');
        result_sig <= (others=>'0');
      end if;
    end if;
  end process;
result(7 downto 0)<=result_sig(7 downto 0);
end Behavioral;
```

Matlab Source codes

Code 10: Leading One Bit Detector

```matlab
function [ k1,k2 ] = MSB1bitdetection_function( n,N1,N2 )
    N1bin=dec2bin(N1,8);
    N2bin=dec2bin(N2,8);
    N1bin_str=num2str(N1bin);
    N2bin_str=num2str(N2bin);
    if(N1~=0)
        k1_all=find(N1bin_str=='1');
        k1_min=min(k1_all);
        k1=(n-k1_min);
    else
        k1=0;
    end
    if(N2~=0)
        k2_all=find(N2bin_str=='1');
        k2_min=min(k2_all);
        k2=(n-k2_min);
    else
        k2=0;
    end
end
```

Code 11: Gaussian 1D Filter

```matlab
function [ c1d,N,Nh,gh1d,quantize_gh1d,hshift,hscale] =
        quantized1dgaussianfilter(sigma )
  c1d=1/sqrt(2*pi*sigma.^2);
  N=2*round(3*sigma)+1;
  Nh=round(3*sigma);
  gh1d=zeros(1,N);
  for x=-Nh:Nh
    gh1d(x+Nh+1)=c1d*exp(-(x.^2)/(2*sigma.^2));
  end
  hshift=floor(log2(1/(max(gh1d))));
  hscale=gh1d*2^(hshift);
  quantize_gh1d=round(hscale*2^8);
end
```

Code 12: Mitchell Original Log Multiplier Function

```matlab
function [ result_ma,k1,k2,k12,x1,x2,x12 ] =
        mitchell_multiplier_function_original( n,N1,N2 )
if(N1~=0 && N2~=0)
  [ k1,k2 ] = MSB1bitdetection_function( n,N1,N2 );
    x1=uint16(bitshift(N1,n-k1,n));
    x2=uint16(bitshift(N2,n-k2,n));
    k12=k1+k2;
    x12=uint16(x1+x2);
    if(x12>=2^8)
        k12=k12+1;
        result_ma=bitshift(x12,int16(k12)-n, 'uint16');
    else
        temp=uint16(bitshift(1,k12,'uint16'));
        x12_shift=bitshift(x12,int16(k12)-n,'uint16');
    result_ma=bitor(temp,x12_shift,'uint16');
     end
else
    result_ma=0;
end
```

Code 13: Mitchell Modified Log Multiplier Function

```matlab
function [ result_ma,k1,k2,k12,x1,x2,x12 ] =
        mitchell_multiplier_function( n,N1,N2 )
if(N1~=0 && N2~=0)
  [ k1,k2 ] = MSB1bitdetection_function( n,N1,N2 );
  x1=uint16(bitshift(N1,n-k1,n));
    x2=uint16(bitshift(N2,n-k2,n));
    N2_actual_log=log2(N2);
    x2=round((N2_actual_log-k2)*2^8);
    k12=k1+k2;
    x12=uint16(x1+x2);
    if(x12>=2^8)
        k12=k12+1;
        result_ma=bitshift(x12,int16(k12)-n, 'uint16');
    else
        temp=uint16(bitshift(1,k12,'uint16'));
        x12_shift=bitshift(x12,int16(k12)-n,'uint16');
        result_ma=bitor(temp,x12_shift,'uint16');
     end
else
    result_ma=0;
end
```

Code 14: Operand Decomposition Original Log Multiplier Function

```
function [ result_od ,result_ma_ab ,result_ma_cd ] = od_mitchell_original_multiplier_function( n,N1,N2 )
if(N1~=0 && N2~=0)
    N1bin=dec2bin(N1,n);
    N2bin=dec2bin(N2,n);
    N1_cmp=bitcmp(N1,'uint8');
    N2_cmp=bitcmp(N2,'uint8');
    a=bitor(N1,N2,'uint8');
    b=bitand(N1,N2,'uint8');
    c=bitand(N1_cmp,N2,'uint8');
    d=bitand(N1,N2_cmp,'uint8');
    if(a~=0 && b~=0)
        [ result_ma_ab ,ka,kb,kab,xa,xb,xab ] = mitchell_multiplier_function_original( n,a,b );
    else
        result_ma_ab=0;
    end
    if(c~=0 && d~=0)
        [ result_ma_cd ,kc,kd,kcd,xc,xd,xcd ] = mitchell_multiplier_function_original( n,c,d );
    else
        result_ma_cd=0;
    end
    result_od=result_ma_ab+result_ma_cd;
    else
    result_od=0;
end
```

Code 15: Operand Decomposition Modified Log Multiplier Function

```
function [ result_od ,result_ma_ab ,result_ma_cd ] = od_mitchell_multiplier_function( n,N1,N2 )
if(N1~=0 && N2~=0)
    N1bin=dec2bin(N1,n);
    N2bin=dec2bin(N2,n);
    N1_cmp=bitcmp(N1,'uint8');
    N2_cmp=bitcmp(N2,'uint8');
    a=bitor(N1,N2,'uint8');
    b=bitand(N1,N2,'uint8');
    c=bitand(N1_cmp,N2,'uint8');
    d=bitand(N1,N2_cmp,'uint8');
    if(a~=0 && b~=0)
        [ result_ma_ab ,ka,kb,kab,xa,xb,xab ] = mitchell_multiplier_function( n,a,b );
    else
        result_ma_ab=0;
    end
    if(c~=0 && d~=0)
        [ result_ma_cd ,kc,kd,kcd,xc,xd,xcd ] = mitchell_multiplier_function( n,c,d );
    else
        result_ma_cd=0;
    end
    result_od=result_ma_ab+result_ma_cd;
  else
    result_od=0;
end
```

Code 16: Iterative Mitchell Log Multiplier with T Stages

```
function [ result_approx ] = iterative_mitchell_multiplier_function( n,N1,N2 )
if(N1~=0 && N2~=0)

    N1bin=dec2bin(N1,n);
    N2bin=dec2bin(N2,n);
    result_actual=uint16(N1*N2);
    result_approx=uint16(0);

    x1=N1;
    x2=N2;
    stage=1;
 t=5 % t=number of stages;

    while(stage<=t)
        if(x1~=0 && x2 ~= 0)
            [ k1,k2 ] = MSB1bitdetection_function( n,x1,x2 );
            x1=uint16(x1-2^k1);
            x1shift=bitshift(x1,k2,'uint16');
            x2=uint16(x2-2^k2);
            x2shift=bitshift(x2,k1,'uint16');
            k12=uint16(k1+k2);
            r=uint16(2^k12);
            c1=r+x1shift+x2shift;
            stage=stage+1;
        else
            c1=0;
            stage=stage+1;
        end
        result_approx=result_approx+c1;
    end
else
    result_approx=0;

end
```

Code 17: Calculation of Optimized Weights for Proposed Log Multiplier

```matlab
function [ N2_modified,N2_modified_round,cf,cf_int ] = derivecf_independentoninputsignal( n,N2 )
if(N2~=0)
    N_temp=1:255;
    x1=zeros(size(N_temp));
    x12=zeros(size(N_temp));
    nocarry=zeros(size(N_temp));
    carry=zeros(size(N_temp));
    term1=double(zeros);
    term2=double(zeros);
    term3=double(zeros);
    term4=double(zeros);
    for i=1:size(N_temp,2)
        [ k1,k2 ] = MSB1bitdetection_function( n,N_temp(i),N2 );
        x1(i)=uint16(bitshift(N_temp(i),n-k1,n));
        x2=uint16(bitshift(N2,n-k2,n));
        x12(i)=uint16(x1(i)+x2);
        if(x12(i)>=2^8)
            carry(i)=N_temp(i);
        else
            nocarry(i)=N_temp(i);
        end
    end
  for i=1:size(N_temp,2)
    [ k1,k2 ] = MSB1bitdetection_function( n,N_temp(i),N2 );
        if (N_temp(i)==nocarry(i))
            term1= term1+N_temp(i)*2^double(k1)*(N2-2^double(k2));
            term4= term4+2^double(2*k1+k2);
        end
        if (N_temp(i)==carry(i))
            term2= term2+N_temp(i)*2^double(k1+1)*(N2-2^double(k2+1));
            term3= term3+2^double(2*k1+k2+2);
        end
    end
    cf=(term1+term2+term3)/(term4+term3);
    cf_int=round(cf*2^8);
    N2_modified=(2^(double(k2)+cf));
    N2_modified_round=round(2^(double(k2)+cf));
else
    N2_modified=0;
    N2_modified_round=0;
    cf=0;
    cf_int=0;
end
end
```

Code 18: Proposed Log Multiplier

```matlab
function [ result_dc,cf,cf_int,k1,k2,k12,x1,x2,x12 ] = proposed_multiplier_function_cf_independentoninputsignal( n,N1,N2,cf_int )
if(N1~=0 && N2~=0)
    [ k1,k2 ] = MSB1bitdetection_function( n,N1,N2 );
    x1=uint16(bitshift(N1,n-k1,n));
    x2=cf_int;
    k12=k1+k2;
    x12=uint16(x1+x2);
    if(x12>=2^8)
        k12=k12+1;
        result_dc=bitshift(x12,int16(k12)-n, 'uint16');
    else
        temp=uint16(bitshift(1,k12,'uint16'));
        x12_shift=bitshift(x12,int16(k12)-n,'uint16');
        result_dc=bitor(temp,x12_shift,'uint16');
    end
else
    result_dc =0;
end
end
```

Code 19: Iterative Modified Algorithm 1

```matlab
function [ result_approx_method2 ] = iterative_first_iterative_exact_second_multiplier_function( n,N1,N2 )
if(N1~=0 && N2~=0)
    N1bin=dec2bin(N1,n);
    N2bin=dec2bin(N2,n);
    result_actual=uint16(N1*N2);
    result_approx_stage1=uint16(0);
    result_approx_method2=uint16(0);
    x1=N1;
    x2=N2;
    stage=1;
    while(stage<=1)
        % function will calculate the position of 1 in the MSB of binary number
        [ k1,k2 ] = MSB1bitdetection_function( n,x1,x2 );
        x1=uint16(x1-2^k1);
        x1shift=bitshift(x1,k2,'uint16');
        x2=uint16(x2-2^k2);
        x2shift=bitshift(x2,k1,'uint16');
        k12=uint16(k1+k2);
        r1=uint16(2^k12);
        c1=r1+x1shift+x2shift;
        result_approx_stage1=result_approx_stage1+c1;
        stage=stage+1;
        if(x1~=0 && x2~=0)
            [ k11,k22 ] = MSB1bitdetection_function( n,x1,x2 );
            x11=uint16(x1-2^k11);
            x11shift=bitshift(x11,k22,'uint16');
            x22=log2(double(x2))-k22;
            x22temp=round(x22*2^k22);
            x22shift=uint16(bitshift(uint16(x22temp),k11,'uint16'));
            k1122=uint16(k11+k22);
            r2=uint16(2^k1122);
            c2=r2+x11shift+x22shift;
        else
            c2=0;
        end
        result_approx_method2=result_approx_stage1+c2;
    end
else
    result_approx_method2=0;
end
```

Code 20: Iterative Modified Algorithm 2

```
function [ result_approx_method1 ] = iterative_first_mitchell_original_second_multiplier_function( n,N1,N2 )
if(N1~=0 && N2~=0)
    N1bin=dec2bin(N1,n);
    N2bin=dec2bin(N2,n);
    result_actual=uint16(N1*N2);
    result_approx_stage1=uint16(0);
    x1=N1;
    x2=N2;
    stage=1;
    while(stage <=1)
    [ k1,k2 ] = MSB1bitdetection_function( n,x1,x2 );
    x1=uint16(x1-2^k1);
        x1shift=bitshift(x1,k2,'uint16');
        x2=uint16(x2-2^k2);
        x2shift=bitshift(x2,k1,'uint16');
        k12=uint16(k1+k2);
        r1=uint16(2^k12);
        c1=r1+x1shift+x2shift;
        result_approx_stage1=result_approx_stage1+c1;
        stage=stage+1;
        [ result_approx_stage2 ] = mitchell_multiplier_function_original( n,double(x1),double(x2) );
        result_approx_method1=result_approx_stage1+result_approx_stage2;
    end
else
    result_approx_method1=0;
end
```

Code 21: Extension of Proposed Log Multiplier

```
function [ result_approx_method1 ] = iterative_first_mitchell_modified_second_multiplier_function( n,N1,N2 )
if(N1~=0 && N2~=0)
    N1bin=dec2bin(N1,n);
    N2bin=dec2bin(N2,n);
    result_actual=uint16(N1*N2);
    result_approx_stage1=uint16(0);
    x1=N1;
    x2=N2;
    stage=1;
    while(stage <=1)
        [ k1,k2 ] = MSB1bitdetection_function( n,x1,x2 );
        x1=uint16(x1-2^k1);
        x1shift=bitshift(x1,k2,'uint16');
        x2=uint16(x2-2^k2);
        x2shift=bitshift(x2,k1,'uint16');
        k12=uint16(k1+k2);
        r1=uint16(2^k12);
        c1=r1+x1shift+x2shift;
        result_approx_stage1=result_approx_stage1+c1;
        stage=stage+1;
        [ result_approx_stage2 ] = mitchell_multiplier_function( n,double(x1),double(x2) );
        result_approx_method1=result_approx_stage1+result_approx_stage2;
    end
else
    result_approx_method1=0;

end
```

Code 22: Gabor Filter Function

```
function [ E,im_out2,gb ] = dcgaborfinal( mf,fo,sigma,
        theta,M )
theta=pi*(theta/180);
Nt=2*(3*sigma)+1;
Nh=ceil(3*sigma);
gb=zeros(Nt);
  for x=-Nh:Nh
    for y=-Nh:Nh
        xr=x*cos(theta)+y*sin(theta);
        gb(x+Nh+1,y+Nh+1)=exp(-(x.^2+y.^2)/(2*sigma^2)).*
      cos(2*pi*fo*xr);
    end
  end
im_out2=filter2(gb,mf);
E=filter2(ones(M,M)/(M*M),abs(im_out2));
end
```

Code 23: Emorio Region Measurements Function

```
function[numreg,rg_data1,S] = current_measurements(im_out1
        )
CC = bwconncomp(im_out1);
L=labelmatrix(CC);
S = regionprops(L,'basic');
numreg = length(S);
rg_data1 = zeros(numreg,7);
for a = 1:1:numreg
    center = S(a).Centroid;
    BBox = S(a).BoundingBox;
    length1 = BBox(1,3);
    breadth1 = BBox(1,4);
    l2 = BBox(1,1)+length1;
    b2 = BBox(1,2)+breadth1;
    rg_data1(a,:) = [center(1,2) center(1,1) BBox(1,1) l2
      BBox(1,2) b2 a];
end
```

Code 24: Snapper Region Measurements Function

```
function[numreg,rg_data1,S0,S] =
        snapper_current_measurements(im_out1)
CC = bwconncomp(im_out1);
L=labelmatrix(CC);
S0 = regionprops(L,'basic');
idx=find([S0.Area]>400);
L1=ismember(L,idx);
S=regionprops(L1,'basic');
numreg = length(S);
rg_data1 = zeros(numreg,7);
for a = 1:1:numreg
    center = S(a).Centroid;
    BBox = S(a).BoundingBox;
    length1 = BBox(1,3);
    breadth1 = BBox(1,4);
    l2 = BBox(1,1)+length1;
    b2 = BBox(1,2)+breadth1;
    rg_data1(a,:) = [center(1,2) center(1,1) BBox(1,1) l2
      BBox(1,2) b2 a];
end
```

Code 25: Emorio Feature Extraction

```
clear all;
close all;
clc;
warning off;

N=100;
io = 1611
 im_past=double(imread(['data',num2str(io),'-L' '.bmp']))
        /255;
im_past=im_past(1:2:end,1:2:end);
imsize=size(im_past);
im_buffer1=zeros(imsize(1)*imsize(2),10);
im_buffer2=zeros(imsize(1)*imsize(2),10);
im_background=importdata('im_background.mat');
im_std=importdata('im_std.mat');
for i=io+1:io+2
    filename = ['data',num2str(i),'-L' '.bmp']
    im=double(imread(filename))/255;
    im=im(1:2:end,1:2:end);
    figure(1),subplot(2,1,1),imshow(im),title('original
      image frame');
    im_diff=im./im_past;
    adjust=median(im_diff(:));
    im_adjusted=im./adjust;
```

```
  im_thres=abs((im_adjusted-im_background)./im_std);
  im_out=im_thres*0+1;
  im_out(im_thres<12)=1;
  im_out(im_thres<6)=1;
  im_out(im_thres<3)=0;
  im_out=medfilt2(im_out,[5 5]);
  im_out=imclose(im_out,ones(15,15));
  im_out=imopen(im_out,ones(15,15));
  figure(1),subplot(2,1,2),imshow(im_out),title('im out');
im1=im;
im_out1=im_out;
[numreg,rg_data1,S]=current_measurements(im_out1);
a1=zeros(numreg);
a2=zeros(numreg);
for a=1:1:numreg
      a1(a,1)=rg_data1(a,3) ;
  a1(a,2)=rg_data1(a,5);
  a1(a,3)=rg_data1(a,4);
  if(a1(a,3)>imsize(2))
    a1(a,3)=imsize(2);
  end
  a1(a,4)=rg_data1(a,6);
  if(a1(a,4)>imsize(1) )
    a1(a,4)=imsize(1);
  end
  a2(a,1)=a1(a,1)-20;
  if(a2(a,1)<0)
    a2(a,1)=a1(a,1);
  end
  a2(a,2)=a1(a,2)-20;
  if(a2(a,2)<0)
    a2(a,2)=a1(a,2);
  end
  a2(a,3)=a1(a,3)+20;
      if(a2(a,3)>imsize(2))
    a2(a,3)=a1(a,3);
  end
  a2(a,4)=a1(a,4)+20;
      if(a2(a,4)>imsize(1))
    a2(a,4)=a1(a,4);
  end
  of=im1(a2(a,2):a2(a,4),a2(a,1):a2(a,3));
  tf=im_out1(a2(a,2):a2(a,4),a2(a,1):a2(a,3));
  tf_sum=sum(tf,1);figure(9),plot(tf_sum);
      H=38;h=22;
  tf_dilate=tf;
  for k =1:1:size(tf_sum,2)
    if((tf_sum(1,k)>=h) && (tf_sum(1,k)<=H))
           w=tf_sum(1,k);
           tf_dilate(:,k)=imdilate(tf(:,k),ones(H-w+1,1));
    else
          tf_dilate(:,k)=tf(:,k);
    end
  end
  figure(10),imshow(tf_dilate);title('tf_dilate');
  mf=of.*tf_dilate;
  figure(4),subplot(3,1,1),imshow(of),title('Original Fish');
  subplot(3,1,2),imshow(tf),title('Threshold Fish');
  subplot(3,1,3),imshow(mf),title('Modified Fish');
  sigma_ref=4;
  fo_ref=0.1;
  lo=132;
  beta=1;
  l=round(sqrt(S(a,1).Area));
  sigma=sigma_ref*((l/lo)^beta)
  fo=fo_ref*((lo/l)^beta);
  theta=0;
 M=10;
  [E,im_out2,gb]=dcgaborfinal(mf,fo,sigma,theta,M );
  figure(5),subplot(2,2,1),imshow(mf);title('modified fish');
  subplot(2,2,2),imshow(im_out2/max(max(im_out2)));title('fish output when gabor horizontal is applied');
  subplot(2,2,3),imshow(E./max(max(E)));title('energy of fish output when gabor horizontal is applied');
  subplot(2,2,4),surf(gb);title('gabor filter in horizontal direction');
  sigma2=sigma_ref*((l/lo)^beta);
  fo2=fo_ref*((lo/l)^beta);
  theta2=90;
 M=10;
    [E2,im_out3,gb2]=dcgaborfinal(mf,fo2,sigma2,theta2,M );
  figure(6),subplot(2,2,1),imshow(mf);title('modified fish');
  subplot(2,2,2),imshow(im_out3/max(max(im_out3)));title('fish output when gabor vertical is applied');
  subplot(2,2,3),imshow(E2./max(max(E2))); title('energy of fish output when gabor vertical is applied');
  subplot(2,2,4),surf(gb2);title('gabor filter in vertical direction');
  im_out2=imdilate(abs(im_out2),ones(11,11));
  final=((im_out3)./abs(im_out2).*sign(imerode(mf,ones(3*sigma,3*sigma))));
  ind1=find(isnan(final));
  final(ind1)=0;
  finalmean=filter2(ones(21,1)/21,final);% here we are running a mean filter of size equal to white strip size=21
  figure(8),subplot(3,1,1),imshow(mf);title('Modified Fish')
  figure(8),subplot(3,1,2),imshow(finalmean);title('mean filter of size=21 applied on final fish image')
  finalmax=max(finalmean,[],1);
  finalmaxmedian=medfilt2(finalmax,[1 5]);
  finalmax(finalmaxmedian<finalmax+0.2)=finalmaxmedian(finalmaxmedian<finalmax+0.2);
  fishmaxpeakvalue=(max(finalmax))
  figure(8),subplot(3,1,3),plot(finalmax);title('plot of length of the fish vs maximum value of mean'),xlabel('length of the fish'),ylabel('maximum intensity');
  sof=size(of);
  figure(9),imshow(of);
  hold on
  figure(9),plot(sof(1)-0.5*sof(1)*(finalmax)/1.5,'r');
  [m w]=max(finalmax);
```

```
        t=text(w,-10+sof(1)-0.5*sof(1)*(finalmax(w))/1.5,num2str(finalmax(w)));set(t,'Color',[1 1 0])
        pause;
      end
%pause;
end
[num,txt,raw]=xlsread('fish feature extraction database.xlsx');
fr=num(:,1);
non_em=num(:,2);
em=num(:,3);
figure(12),scatter(fr,non_em,'S','red');title('graph of frame vs non E.Morio fish ');
figure(13),scatter(fr,em,'O','blue');title('graph of frame vs E.Morio fish');
hold on,figure(14),scatter(fr,em,'O','blue');title('graph of frame vs  E.Morio & Non E.Morio ');
```

Code 26: Snapper Feature Extraction

```
clear all;
close all;
clc;
warning off;
N=100;
io = 11600
%% Reading the first frame
im_past=double(imread([num2str(io) '.tif']))/255;
% Downsampling the first frame
im_past=im_past(1:2:end,1:2:end);
% The size of frames, i.e. imsize(1) x imsize(2)
imsize=size(im_past);
im_buffer1=zeros(imsize(1)*imsize(2),10);
im_buffer2=zeros(imsize(1)*imsize(2),10);
im_background=importdata('1600_im_background.mat');
figure(2),imshow(im_background),title('background image');
im_std=importdata('1600_im_std.mat');
figure(3),imshow(im_std),title('standard deviation of the image');
aa=44;
bb=44;
nf=abs(bb-aa);
frame_max=cell(2,nf+1);
count=1;
for ii=io+aa:io+bb
    filename=[num2str(ii) '.tif']
    im=double(imread(filename))/255;
    im=im(1:2:end,1:2:end);
    figure(1),subplot(2,1,1),imshow(im),title('original image frame');
    im_diff=im./im_past;
    adjust=median(im_diff(:));
    im_adjusted=im./adjust;
    im_thres=abs((im_adjusted-im_background)./im_std);
    im_out=im_thres*0+1;
    im_out(im_thres<12)=1;
    im_out(im_thres<6)=1;
    im_out(im_thres<3)=0;
    im_out=medfilt2(im_out,[11 11]);
    figure(1),subplot(2,1,2),imshow(im_out),title('im out');
    im1=im;
    im_out1=im_out;
    [numreg,rg_data1,S0,S]=snapper_current_measurements(im_out1);
    a1=zeros(numreg);
    objectmax(1,1:numreg)=double(zeros(1,numreg));
    rotationmax(1,1:numreg)=double(zeros(1,numreg));
    for a=1:1:numreg
        sprintf('object a=%d ',a)

        a1(a,1)=rg_data1(a,3) ;
        a1(a,2)=rg_data1(a,5);
        a1(a,3)=rg_data1(a,4);
        if(a1(a,3)>imsize(2))
            a1(a,3)=imsize(2);
        end
        a1(a,4)=rg_data1(a,6);
        if(a1(a,4)>imsize(1) )
            a1(a,4)=imsize(1);
        end
        deg=30;
        stepinterval=5;
        step=(deg+deg)/stepinterval+1;
    result=cell(step,4);
        for rotation_degree=-deg:stepinterval:deg
            of=im1(a1(a,2):a1(a,4),a1(a,1):a1(a,3));
            of=imrotate(of,rotation_degree);
            tf=im_out1(a1(a,2):a1(a,4),a1(a,1):a1(a,3));
            tf=imrotate(tf,rotation_degree);
            mf=of.*tf;
            figure(4),subplot(3,1,1),imshow(of),title('Original Fish');
            subplot(3,1,2),imshow(tf),title('Threshold Fish');
            subplot(3,1,3),imshow(mf),title('Modified Fish');
            sigma_ref=2;
            fo_ref=0.1;
            lo=58;
            beta=1;
            l=round(sqrt(S(a,1).Area));
            sigma=sigma_ref*((l/lo)^beta);
            fo=fo_ref*((lo/l)^beta);
            theta=0;
            M=10;
            [E,im_out2,gb]=dcgaborfinal(mf,fo,sigma,theta,M );
            figure(5),subplot(2,2,1),imshow(mf);title('modified fish');
            subplot(2,2,2),imshow(im_out2/max(max(im_out2)));title('fish output when gabor horizontal is
applied');
        subplot(2,2,3),imshow(E./max(max(E)));title('energy of fish output when gabor horizontal applied');
        subplot(2,2,4),surf(gb);title('gabor in horizontal direction');
        sigma2=sigma_ref*((l/lo)^beta);
        fo2=fo_ref*((lo/l)^beta);
        theta2=90;
        M=10;
        [E2,im_out3,gb2]=dcgaborfinal(mf,fo2,sigma2,theta2,M );
        figure(6),subplot(2,2,1),imshow(mf);title('modified fish');
        subplot(2,2,2),imshow(im_out3/max(max(im_out3)));title('fish output when gabor vertical is applied');
        subplot(2,2,3),imshow(E2./max(max(E2))); title('energy of fish output when gabor vertical applied');
        subplot(2,2,4),surf(gb2);title('gabor in vertical direction');
        final=((im_out2)./abs(im_out3).*sign(imerode(mf,ones(4.8*sigma,4.5*sigma))));
        ind1=find(isnan(final));
        final(ind1)=0;
        figure(7),subplot(3,1,1),imshow(final);title('final fish image with horizontal line detected in black');
        final_imsize=size(final); % final_imsize gives the size of final where final is the fish image with line detected in black
        final_mod=max(max(final))-final;
        figure(7),subplot(3,1,2),imshow(final_mod);title('final fish image with horizontal line detected in white');
        mm=mean(final_mod(final_mod<max(max(final_mod))));
        final_mod(final_mod==max(max(final_mod)))=0;
        final_mod=final_mod-mm;
        final_mod(final_mod<0)=0;
        figure(7),subplot(3,1,3),imshow(final_mod);title('final fish image with horizontal line detected in white and outside fish area keeping it black i.e not changing anything outside fish area ');
        result{(rotation_degree/stepinterval)+(deg/stepinterval)+1,1}=rotation_degree;
        result{(rotation_degree/stepinterval)+(deg/stepinterval)+1,2}=final_mod;
        result{(rotation_degree/stepinterval)+(deg/stepinterval)+1,3}=filter2(ones(1,l)/(l), result{(rotation_degree/stepinterval)+(deg/stepinterval)+1,2});
        result{(rotation_degree/stepinterval)+(deg/stepinterval)+1,4}=max( result{(rotation_degree/stepinterval)+(deg/stepinterval)+1,3},[],2);
        pause;
      end
      [maxvalue rotation]=max(cc_result);
      rotation1 = (rotation -1)*stepinterval;
      objectmax(1,a)=maxvalue
      rotationmax(1,a)=rotation1
      pause;
    end
    frame_max{1,count}=objectmax;
    frame_max{2,count}=rotationmax;
    if count==1
       histvariable=frame_max{1,count};
    end
    histvariable=[histvariable frame_max{1,count}];
    count=count+1;
end

figure(10),hist(histvariable,nf*10);
[num,txt,raw]=xlsread('fish feature extraction database.xlsx');
fr=num(:,1);
non_em=num(:,2);
em=num(:,3);
figure(12),scatter(fr,non_em,'S','red');title('graph of frame vs non emoria fish ');
figure(13),scatter(fr,em,'O','blue');title('graph of frame vs emoria fish');
hold on,figure(14),scatter(fr,em,'O','blue');title('graph of frame vs  emoria fish ');
```

Vita

Arjun Kumar Joginipelly was born in 1985, India. He received his Bachelor of Engineering in Electronics and Communications from Jawaharlal Nehru Technology University, A.P, India in May, 2007. He received his Masters in Electrical Engineering from University of New Orleans (UNO) in Dec, 2010. He then continued on with his PhD in Electrical Engineering, graduating in Aug, 2014. While at UNO, he worked as graduate research assistant under Dr. Dimitrios Charalampidis. His areas of interest include Image processing using FPGAs, Separable and Bilateral filters, Feature Extraction and Classification, and Logarithmic Multipliers. He spends his free time by playing racquetball, volleyball and video games.