Integrated Sorting, Noise Estimation, Object Detection and Contour Analysis on one FPGA for Video Object Segmentation

Kumara Ratnayake

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science (Electrical and Computer Engineering) at

Concordia University

Montréal, Québec, Canada

June 2007

Your file Votre référence
ISBN: 978-0-494-34453-8
Our file Notre référence
ISBN: 978-0-494-34453-8

ABSTRACT

Integrated Sorting, Noise Estimation, Object Detection and Contour Analysis on one FPGA for Video Object Segmentation

Kumara Ratnayake

Although solutions for robust video processing methods, such as compression or segmentation, have been considerably investigated using general-purpose processors (GPPs), these software implementations are too slow to achieve real-time performance due to the computational complexity and memory bandwidth involved in present complex video processing methods. As such, efficient hardware accelerations are inevitable for fast video systems. The state-of-the-art field programmable gate arrays (FPGAs) fill the gap between very inflexible, but high performance ASICs and flexible, yet performance-constrained GPPs. Thus, FPGAs are increasingly employed on hardware platforms in many signal and video processing applications.

This thesis proposes an FPGA-based architecture that integrates four video processing methods (sorting, noise estimation, object detection, and contour analysis) on one FPGA, which takes a video signal and outputs a contour filled video sequence along with the corresponding contour chain codes. The proposed architecture aims at segmenting moving objects in video signals. A video object segmentation consists of several steps: pre-processing (e.g., noise estimation), object detection (i.e., separation of objects and background), and contour analysis. The proposed architecture is simulated, synthesized and verified for its functionality, accuracy and performance on an actual hardware platform consisting of a Xilinx Virtex-4 SX35 FPGA.

Compared to related work, our architecture obtains orders of magnitude performance improvements utilizing minimal hardware resources and power, and possesses key algorithmic features, which are inherently required in many video processing applications.

Acknowledgments

This thesis is based on three years of research on my part, but would not have been possible without inspiration, education and support from a number of people.

First, I would like to express my deepest appreciations and gratitude to my supervisor Dr. Aishy Amer. Her guidance and support have been instrumental in my success. With your enthusiasm, inspiration, and great efforts to explain things clearly and simply, you helped me to succeed in this wonderful research world. Thank you for guiding me through the writing of all the reports, papers and, of course, this thesis, and for all the corrections and revisions made to each of them.

I would like to thank all my colleagues at VidPro for their support: Chang Su, M. Ghazal, Firas, Francois, El Helali, Hanif, Julius, Bin and Ken.

Last, but not least, I am forever indebted to my entire family for their love, understanding, endless patience and encouragement when it was most required.

To my daughter Melina Seneli Ratnayake.

Contents

List of Figures

List of Tables

List of Notations

General Acronyms

1D	One Dimensional
2D	Two Dimensional
ASIC	Application Specific Integrated Circuits
CCL	Connected Component Labeling
CIF	Common Intermediate Format
CLB	Configurable Logic Blocks
DCM	Digital Clock Manager
DMA	Direct Memory access
DSP	Digital Signal Processing
EDIF	Electronic Design Interchange Format
FIR	Finite Impulse Response
FHV	Formal Hardware Verification
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
HDTV	High Definition Television
IO	Input Output
LUT	Look Up Tables
NTSC	National Television Standards Committee
PAL	Phase Alternate Line
RAM	Random Access Memory
SNR	Signal to Noise Ratio
SRAM	Static RAM
SBSRAM	Synchronous Burst SRAM
DRAM	Dynamic RAM
DDR	Double Data Rate
BOM	Bill Of Materials
RTL	Resistor Transistor Level
VHDL	Very high speed integrated circuit Hardware Description Language
VLSI	Very Large Scale Integration

Sorting and Video Noise Estimation

a_i	Unsorted Keys
d_i	Stably Sorted Keys
$I_{i,j}$	Pixel Intensity at Position (i,j)
W	Block Size
ξ_{Bh}	Block Homogeneity Measure
σ^2_{Bh}	Block Variance
μ_{Bh}	Block Sample Mean
σ^2_{REF}	Reference Variance
t_σ	User Defined Homogeneity Threshold
σ^2_n	Image Global Noise Variance

Object Segmentation

n	Current Discrete Time Instant
$AD(n)$	Absolute Frame Difference
$I(n)$	Current Frame
$R(n)$	Reference Frame
$BK(n)$	Background Frame
$I(n-1)$	Previous Frame
$D(n)$	Spatially-Filtered Frame
T_g	Global Spatial Threshold
W_k	kth Consecutive Non-Overlapping Block
K	Number of Consecutive Non-Overlapping Blocks
L	Number of Sections in W_k Histogram
μ_k	Mean Value of W_k
a	Noise Scaling Factor
$E_{hw}(n)$	Hardware Result (Frame)
$E_{sw}(n)$	Software Result (Frame)
Δ_{hw}	Absolute Different between the $E_{hw}(n)$ and $E_{sw}(n)$
PCP	Product of Correctly Classified Proportions

Contour Tracing and Filling

$E(n)$	Gaps Free Edge Image
p_i	A Point in an 8-Neighborhood
p_w	White Point
p_b	Background (Black) Point
p_s	Starting Point
p_c	Current Point
p_p	Previous Point
d_s	Searching Direction
C_c	Current Contour
C_p	Previous Contour
$C(n)$	Current Contour List
$C(n-1)$	Previous Contour List
cc_i	Current Contour Point
cc_{i+1}	Next Contour Point

Chapter 1

Introduction

1.1 Motivation

The computational complexity involved in today's complex video processing algorithms [6–10] make virtually impossible to achieve real-time performance on general purpose sequential processor-based systems. Thus, an efficient hardware acceleration is inevitable. There exists three possible hardware-based solutions: 1) parallel computing, 2) Application Specific Integrated Circuit (ASIC), and 3) FPGAs. The parallel computing gives the highest flexibility among the three solutions, but heat dissipation and area are all increased. Traditional full custom ASIC provides the highest processing speed with minimal power consumption, but suffers from longer development time and expensive engineering cost. In contrast, emerging FPGAs with embedded multipliers, memory blocks and high pin counts, are increasingly employed on hardware platforms in many signal/video processing applications [11]. Moreover, FPGA reconfigurability is an attractive feature which allows an FPGA-based systems to be adopted for an another purpose.

Video object segmentation, which classifies pixels of a frame into multiple moving regions, plays a key role in many video processing applications such as video compression, video surveillance, machine vision, and enabling techniques such as object-based motion estimation. Video object segmentation methods [12] can be classified based on their automation, spatial accuracy, temporal stability, and computation load. Computationally expensive methods give, in general, accurate results while low-computation methods may fail. However, few of the methods are tested on a large number of videos, throughout long videos, on noisy videos, images with artifacts, and without parameter tuning.

The non-parametric object segmentation method, described in [1], features low computation and noise and temporal stability. These features forgo spatial accuracy, e.g., at object boundaries. Such a method is most appropriate to applications, e.g., video surveillance, where stability under varying conditions is of more concern than accurate object boundaries. Consequently, the video object segmentation algorithm [1] first estimates the noise [13] to *effectively* detect moving video objects from background and then performs contour tracing and filling of these detected objects.

In this thesis, we propose a real-time, robust, scalable, and compact FPGA-based architecture and its implementation of video object segmentation algorithm [1]. Our finding to the existing related work confirms that the proposed architecture is much more feasible and cost effective. Furthermore, performance analysis shows that our hardware approach achieves an order of magnitude performance improvement over the existing pure software-based implementations. Fig.1.1 illustrates a high-level block diagram of the proposed FPGA-based implementation of the video object segmentation. Note that the output of the proposed system is an image of filled objects and the corresponding contour chain codes. Therefore, final results of the FPGA are presented in chapter 5.

Noise is inevitable in video signals due to video acquisition, processing, and transmission. Moreover, noise can significantly affect the performance and effectiveness

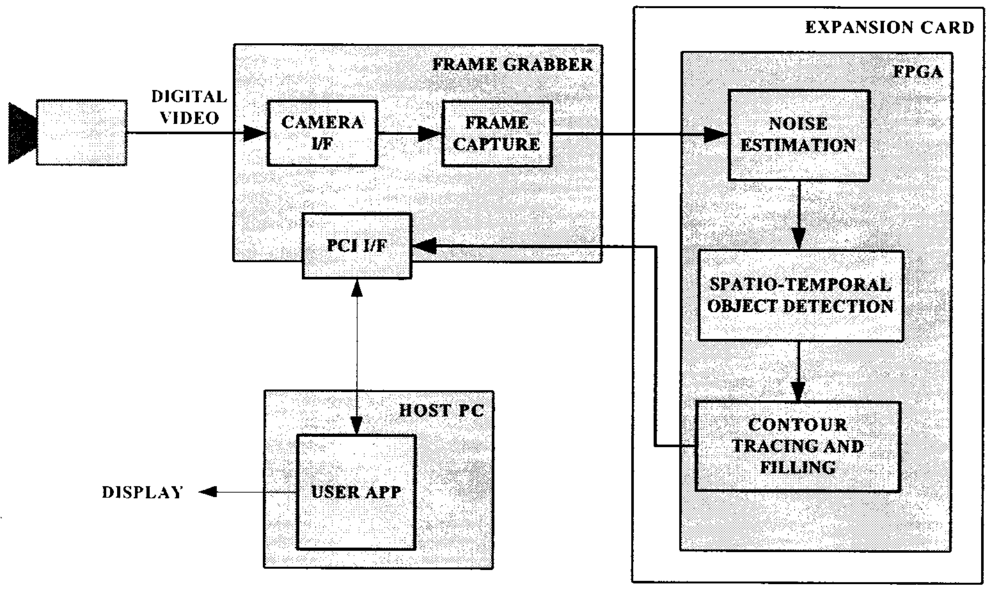

Figure 1.1: Proposed FPGA-based Implementation of Video Object Segmentation.

of many video processing algorithms, such as edge detection, object segmentation, motion estimation, and video filtering [13]. Thus, measuring noise, through noise estimation algorithms, in a corrupted video signal and adapting the video processing algorithms accordingly is fundamental and important feature for stable video processing systems.

In general, the noise estimation algorithms use spatial or temporal image characteristics to measure noise. Temporal noise estimation methods require one or more previous frames, and are more computationally intensive, while most of the spatial methods rely on intra-frame block-based variance or smoothing. The intra-field block-based noise estimation method [13] gives reliable estimates both in highly noisy and good quality images. This homogeneity-based technique utilizes a novel homogeneity measure, by taking image structure into account, rather than variance alone to determine if a block is homogeneous. Moreover, the methods requires to sort 10% of the block variances with respect to the homogeneous measure. The main computational bottleneck of the noise estimation algorithm [13] resides in this sorting process.

Sorting (in ascending order) is a process to rearrange a given set of keys a_0, a_1,..., a_{N-1} into a set d_0=a_{i_0}, d_1=a_{i_1}, ... , d_{N-1}=$a_{i_{N-1}}$ such that d_0 $\leq$, d_1 $\leq$, ... , $\leq$ d_{N-1}. Furthermore, sorting is stable if d_j=d_{j+1}, then $i_j\leq$ i_{j+1}, i.e., for each pair of consecutive equal keys in the sorted list, the order that they were presented in the unsorted set is preserved. Sorting a large volume of integer data is computationally intensive and is often required in diverse applications such as video processing [13] and ATM switching [14]. Software-based implementation for sorting large volume of data is still slow for real-time applications.

Contour tracing is a method that links connected neighborhood pixels in a binary edge frame, whereas contour filling fills the region inside a contour with a specific gray-level value, uniquely labeling each objects in an image. Contour tracing and filling are a fundamental element in many video and image processing applications such as object segmentation [1], medical image processing [15], computer vision [16] and pattern recognition [17].

1.2 Summary of Related Work

This section summarizes the architectures and implementations of video noise estimation, sorting, video object detection, video object contour trace and filling on FPGA and VLSI devices. Detailed related work review is given in chapters 2, 3, 4 and 5.

1.2.1 Noise Estimation and Sorting

To the best knowledge of the author, only Lapalme, in [5], has presented an FPAG-based architecture for estimating noise in digital video, but [5] requires a significant amount of FPGA resources, runs at a slower speed, and employs two SRAM devices. In contrast, the proposed architecture outperforms [5] significantly and avoids the utilization of external SRAM modules, which increase the area, cost, power and verification time of the overall video processing system.

Much research has been carried out implementing parallel sorting algorithms on VLSI recently, but very little emphasis has been put on FPGA-based implementations. Most of the FPGA-based architectures, which are based on pure comparison sorting algorithms, require an enormous of silicon area when sorting has to be performed on a very larger volume of data.

A hybrid implementation of a hardware sorter is studied in [18]. Here, the architecture is split into a sequential merge sorting algorithm running on a sequential processor, and a systolic insertion sorting algorithm running on a Xilinx XCV1000 FPGA co-processor. Huang et al. propose another architecture in [4] for merge-sorting networks with a fixed size Batcher's sorting network. However, performance of this architecture is considerably low for applications that require real-time sorting performance such as in [13]. Another VLSI architecture is presented in [14], which sorts small continuous data sequences up to 256 keys and partially sorts longer sequences. The architectures in [14,18] are based on comparative based sorting methods, therefore implementing them on FPGAs require more resources (area) than the proposed architecture based on a modified counting sort.

In signal processing, sorting is widely used in rank-order filters which are preferred over linear filters [19]. Here, the complexity of sorting is relatively small as the resolution and the number of data becomes small and no stable sorting is required. Several FPGA implementations for median filters have been presented [19–22] each of them exploiting different approaches for sorting architectures. Maheshwari et al. [19] propose a simple approach to a 3×3 median filter implementation on an FPGA, by sorting the elements in the 3×3 kernel vertically, horizontally and diagonally. In [20], a bit serial sorting algorithm and implementation are introduced for a general purpose median filter on an FPGA. Hamid et al. [21] present a genetic algorithm on a Xilinx Spartan-II FPGA device to optimize the tasks of the soft morphological filter (SMF), whereas Fahmy et al. [22] propose an efficient implementation of one dimensional (1D) median filters based on cumulative histogram approach. Although implementation in [22] can process 72 MPixels/s for 8 bit pixel depth, it is not extended to 2D median filters. Moreover, achieving timing constraints on the implementation [20] can be very difficult for higher pixel depths, due to the nature of the priority encoder. Although the comparative methods proposed in [19–22] can theoretically be extended to sort a large number of keys, implementation of such methods on a resources constrained hardware such as FPGAs would not be feasible due to the large silicon area required.

1.2.2 Object Detection

The method described in [23] demonstrates how a number of object detection algorithms can be implemented on FPGAs with dynamic reconfigurability feature. Another FPGA implementation for segmenting text in images is in [24]. Experimental results show that this algorithm implemented in FPGA achieved a speedup of close to 250 compared to a general purpose CPU implementation. However, [24] runs at 5 MHz which is well below the real-time performance. The study in [25] partially involves FPGA-based implementation of image segmentation based on the resistive-fuse network model. An extensive comparison between FPGA and DSP implementations of image classifier for object detection is in [26]. Although the performance of the FPGA implementation significantly overpasses that of the DSP implementation, its performance and scalability is heavily limited and embedded by the hardware platform chosen.

1.2.3 Object Contour Tracing and Filling

Agi et al. in [27] propose a full custom VLSI CMOS design for extracting contours, by attempting to minimize the memory usage by partitioning the input frame into smaller regions and distributing these regions to an array of processing elements (PEs). Each PE in [27] consists of its own memory and a contour tracing unit, and uses a 2×2 window for extracting partially completed contour lists. However, the technique in [27] fails to produce completed contour tracing, unless a full object can be completely stored in the relatively small processing memory. A parallel VLSI architecture consisting of $N + 1$ processing elements for generating the chain codes of object contours in a binary frame with N raws is presented in [28]. The algorithm proposed in [28] completes contour extraction in $3N$ cycles, but heavy parallel memory access required in [28] makes its architecture virtually infeasible to implement with presently available memory devices.

Although studies have been carried out to implement Connected Component Labeling, CCL, algorithms on hardware [29–33], no previous work has been dedicated on, to the authors' best knowledge, FPGA-based contour filling methods. The CCL methods assign a label to a pixel such that its adjacent and identical pixels have the same label. As such, the CCL algorithms can only label *filled* objects imposing a significant constraint in many video processing applications. A CCL architecture proposed for by Rasquinha et al. in [29] using N processing elements for $M \times N$ image. In [30], Crooks et al. present an FPGA architecture for CCL, which requires scanning iteratively the input and intermediate images until no change in the resulting image occurs. However, [30] achieves real-time performance *only* for images with simple objects, and therefore, fails to completely label real video objects in applications such as video surveillance. Another VLSI architecture, consisting of four processors, for CCL is presented by [31], and Jablonski et al. [32] present an implementation of Classical

CCL in Handel-C language. A fast and parallel VLSI architecture for object labeling in binary images, using a 3×4 window, is presented by Shyue et al. in [33]. The CCL can be exploited for parallelism, but contour filling methods are inherently sequential, therefore implementing sequential contour filling algorithms are more challenging than CCL on parallel hardware devices such as FPGAs.

1.3 Background on FPGAs

Historically, FPGAs were known as prototyping and integration devices for lower volume digital systems due to their higher cost and lower performance. However, deep sub-micron (e.g., 65-nm copper CMOS) silicon process technology has dramatically changed in recent years allowing new levels of integration onto reprogrammable chips with more features and capabilities. Consequently, state-of-the-art FPGAs have already exceeded densities of tens of million system gates operating at speeds surpassing 400 MHz, and therefore these are increasingly employed in digital embedded system as a compromise to inflexible full-custom ASIC devices and performance-constrained sequential processors.

It is, however, noteworthy to mention disadvantages of FPGAs when compared to the ASICs. The fact that an ASIC is designed from scratch for a specific task allows optimizing the area, speed and power up-to the gate-level. This level of optimization is not available for FPGAs, since the FPGA devices are already manufactured by the FPGA vendors. Thus, for a given algorithm implemented on an FPGA lags behind the same implementation on an ASIC in silicon area, performance and power dissipations. Moreover, a unit cost of an FPGA is relatively higher than an ASIC having the same functionality, hence it is difficult to justify to design embedded systems with FPGAs for high-volume applications such as those in consumer markets. However, in this thesis we have chosen FPGAs over ASICs for the implementation because FPGAs provide the ability to change the hardware entity with their inherent reconfigurability feature allowing the FPGA-based platform to be upgraded or enhanced throughout the product life time after the deployment. Furthermore, ASICs are intrinsically based on high-risk methodology requiring massive volume to recoup the Non-Recurring Engineering (NRE) costs required to create an ASIC at the beginning. Hence, FPGAs' flexibility with low risk and shorter development cycles make them as the optimal choice for a hardware implementation.

Widely known and available SRAM-based FPGAs can be configured, reconfigured or partially reconfigured by writing a bit-stream to the FPGAs, depending on the application requirements. In general, FPGAs are fabricated with matrices of few fundamental building blocks: configurable logic blocks (CLBs), embedded multipliers (or digital signal processing slices - DSP48E), blocks of random access memory (BRAMs), clock buffers (BUFGs), clock synthesizers (DCMs), and programmable interconnection network among these elementary components. Additionally, some FPGA families include hard-processors such as PowerPC in Xilinx Virtex4FX family. Throughout this thesis, we have used FPGAs from Xilinx and not Altera or any other vendors, primarily because Xilinx FPGAs possess some architectural features such as the ability to configure four input Look-up-tables (LUTs) into sixteen shift registers (SRL16) and availability of hardcore PowerPC processors, which are not present in FPGAs, for e.g., from Altera.

Synchronous or combinatorial logic circuits are implemented by configuring CLBs, which, in Xilinx Virtex-5 FPGAs [34], are comprised of two slices each containing four LUTs and four flip-flops. Each DSP48E slice contains a twos complement 25x18 multiplier and a 48-bit signed adder/accumulator. Excessively available DSP48Es are ideal for parallel implementation of many signal and video processing functions such as FIR filters, 2D convolution and correlation. BRAMs are 36 Kbit true dual-port memory blocks which are programmable from 32Kx1 to 512×72, in various depth and width configurations or can be configured to operate as two, independent 18 Kbit dual-port memory blocks. These BRAMs are extensively utilized from simple first in first out (FIFO) memories to line buffers and histogram computations.

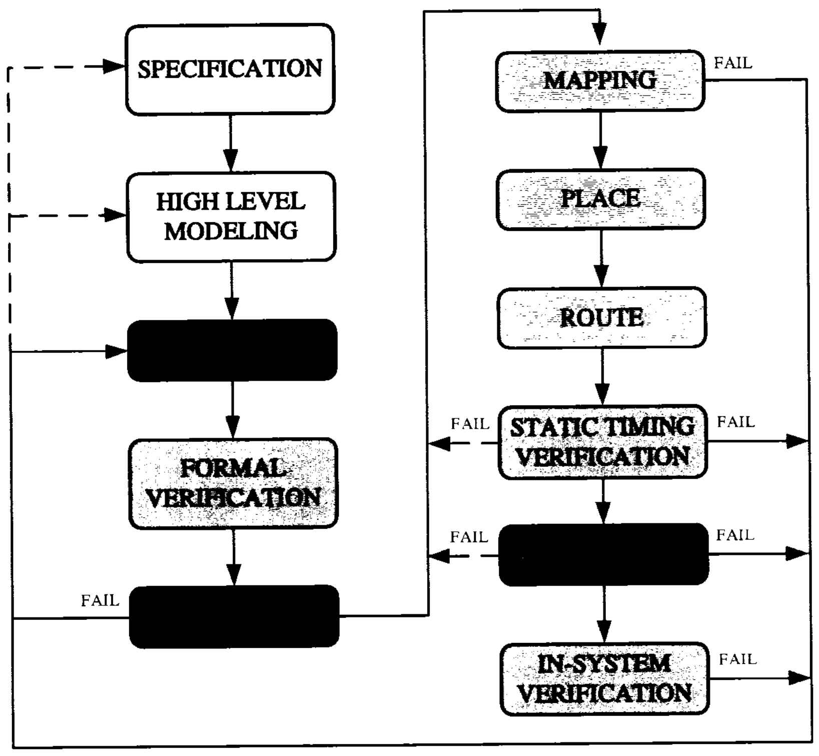

Figure 1.2: Typical FPGA Design Flow.

Fig. 1.2 illustrates a typical FPGA design flow, which, throughout this thesis work, has been used. Starting with specification defined in floating point C/C++ for all algorithms - noise estimation, sorting, object detection and contour tracing and filling - we model a virtual hardware with fixed-point C/C++ and MATLAB. These fixed point high-level models are extensively analyzed to formalize the final hardware design correlate *maximally* with the original specification. In addition, parallelism of the video processing algorithms are explored.

The high-level models are then coded and formally verified in a hardware description language - VHDL, which is extensively used as one of the modern approaches of designing digital circuits. Due to the complexity and scale of the hardware designs targeted to video processing systems, it is vital to perform vigorous verification through formal hardware verification (FHV) methods to ensure that the final design is relatively bug-free. We have modeled a verification platform that automates most of the formal verification tasks by combining high level languages, such as C/C++ and MATLAB, with VHDL test-benches. The state-of-the-art VHDL simulation tools allow co-simulations with MATLAB, which provides a higher level of verification environment, facilitating simulation of the design modules such as sorting quickly. Moreover, our verification platform accepts, not just a mere set of VHDL stimuli, but a *video sequence* which is passed through the synthesizable hardware design. The resulting data, composed of a video sequence and/or contour chain codes, are then automatically compared with the high level specification, originally written in C/C++. In addition, the environment generates various objective and subjective measurements of the hardware result, which facilitate comparing the intended hardware design with the original specification.

The VHDL codes are then synthesized with Synplicity Synplify, which translates the design into a technology specific netlist file format - Electronic Design Interchange Format (EDIF). Synthesis reports provide approximate timing and resources utilization of the target FPGA. If the synthesis results are not satisfactory, the design may require more optimization for area and timing, or, in some instances, changes to specification and overall architecture.

The place and route tools (Xilinx ISE) take the EDIF netlist and map into appropriate fundamental building blocks, for e.g., CLBs. Once the design has been placed, the tools perform an *iterative* routing procedure to interconnect the mapped blocks producing a routed design with minimal interconnection delays. Here, static timing verifications are performed and critical paths of the circuits are analyzed to improve the overall design performance. The routed FPGA design is then programmed into the FPGA to verify its functionality. In system verification tools, such as Xilinx Chipscope Pro^{TM} [35], are thoroughly used throughout the course of prototyping and circuit debugging. In the case of the routed design does not meet the timing, area and power, or if it is not functional correctly in real hardware in accordance with the specification or design goals, the complete design flow may require running a number of iteration, even from the specification level.

1.4 Overview of Contributions

The following list states which parts of this thesis are original to the knowledge of the author at the time the proposed methods of this thesis were developed [1]:

- A hardware-friendly modified counting sort algorithm.
- A single-chip scalable and compact FPGA-based architecture of the proposed counting sort algorithm which specifically addresses the issue of sorting large volume of integer or fractional data.
- An improved FPGA-based noise estimation where the following improvements have been made:
  - Excludes highly undesirable external SRAM.
  - Removes 53% BRAM utilization in logorithmic computation.

[1]At the time of this thesis publication, four papers based on this thesis are published in the Proceedings of:

- the 2006 IEEE International Conference on Image Processing (object detection) [36],
- the 41st IEEE Conference on Information Sciences and Systems (sorting) [37],
- the 2007 IEEE International Conference on Image Processing (contour tracing) [38], and
- the 2007 IEEE International Conference on Systems, Man, and Cybernetics (contour filling) [39].

  - Integrates into one FPGA device where video object detection and contour tracing and filling are implemented.

- A scalable and compact architecture for FPGA-based object detection.

- An implementation of chaotic contour tracing and filling of video objects on FPGA.

Various referenced architectures and implementation for sorting, video noise estimation, object detection and contour tracing and filling are studied, and their characteristics and performance are analyzed and compared with the proposed methods.

1.5 Thesis Outline

The remainder of the thesis is organized as follows.

Chapter 2 presents the proposed FPGA-based implementation of modified counting sort algorithm which is utilized in the implementation of spatial video noise estimation. Section 2.2 describes the related work to both sorting large volume. The modified counting sort algorithm and its adaptation to the FPGA implementation, verification and results are described in section 2.3 through 2.6.

In chapter 3, we outline an efficient hardware architecture in section 3.4 of the reference homogeneity-based noise estimation algorithm, which is described in section 3.3. The improvements of the proposed architecture and results are discussed in section 3.5.

Chapter 4.1 proposes a robust real-time, scalable and modular FPGA-based implementation of a spatio-temporal video objects detection. Section 4.2 outlines the reference algorithm and section 4.3 depicts the proposed architecture. Section 4 contains Verification and synthesis results are given in section 4.4.

In chapter 5, we propose novel FPGA-based architecture and its implementation of contour tracing and filling of video objects. Section 5.2 describes the related work and section 5.3 gives an overview of the contour tracing and filling algorithms [8]. Our proposed architecture is outlined in section 5.4, while section 5.5 contains experimental results. Note that the output of the proposed system is an image with filled moving objects, and their corresponding contour chain codes. Therefore, final results of the FPGA are presented in chapter 5.

Chapter 6 concludes the thesis and suggests future work.

Chapter 2

A Modified Counting Sort Algorithm and its FPGA Implementation

Many video processing algorithms, such as video noise estimation [13], often require sorting large volume of data. This chapter proposes a single-chip scalable and compact FPGA-based architecture of a modified counting sort algorithm, which specifically addresses the issue of sorting large volume of integer or fractional data.

2.1 Introduction

Sorting (in ascending order) is a process to rearrange a given set of keys a_0, a_1,..., a_{N-1} into a set d_0=a_{i_0}, d_1=a_{i_1}, ... , d_{N-1}=$a_{i_{N-1}}$ such that d_0 $\leq$, d_1 $\leq$, ... , $\leq$ d_{N-1}. Furthermore, sorting is stable if d_j=d_{j+1}, then $i_j \leq$ i_{j+1}, i.e., for each pair of consecutive equal keys in the sorted list, the order that they were presented in the unsorted set is preserved.

Sorting a large volume of integer data is computationally intensive and is often required in diverse applications such as video processing [13] and ATM switching [14]. Software-based implementation for sorting large volume of data is still slow for real-time applications, hence hardware acceleration is inevitable. Although parallel computing based solutions are inherently very flexible, heat dissipation and area are all increased. Designing a full-custom ASIC chip provides the highest processing speed with minimal power consumption, but requires a longer design cycle, expensive engineering cost, and is extremely inflexible. FPGAs, which fill the gap between general purpose sequential processors (GPPs) or parallel computers and ASICs are, therefore, utilized in many embedded systems. Evolving high-density FPGA architectures, such as those with embedded multipliers, high number/amount of memory blocks and high pin count, make FPGAs as an ideal solution for accelerating heavy number crunching operations such as those required in video processing.

The choice of an optimal sorting algorithm and adapting it to FPGAs are critical to have an optimized compact implementation. In applications that require sorting, it is most likely that sorting is a very small part of the whole application [13, 40]. The use of dedicated sorting chips is not a choice in cost-sensitive system designs as this would increase bill of material (BOM) costs. Implementing comparative based sorting methods [14, 18] would consume an enormous amount of FPGA resources.

In this chapter, we propose a modified counting sort algorithm [3] and an architecture such that it is well suited for an FPGA-based implementation. The primary motivation for designing a compact and efficient implementation for sorting large data volumes comes from the need to gain real-time performance in video processing algorithms [13,40]. The noise estimation algorithm in [13] requires to sort approximately 8000 fractional keys (variances) in real time, which is 33 ms for a standard video source running at 30 frames/s. The number of keys to be sorted could be well over 8000 for higher video frame sizes such as HDTV video resolution, and sorting is a very small part of [13] and many other video processing applications [40]. Hence a scalable, real-time and compact implementation for sorting variable and large volume of data is needed.

2.2 Related Work Review

In recent years, much research has been carried out implementing parallel sorting algorithms on ASICs, but very little emphasis has been put on FPGA-based implementations. Most of the FPGA-based architectures are based on pure comparison sorting algorithms. The area and time complexity becomes an enormous issue when sorting has to be performed on a very larger volume of data.

Studies carried out in [18] investigate trade-off of cost versus performance for a hybrid sorting algorithm of large data. This hybrid sorting algorithm is split into a sequential merge sorting algorithm running on a sequential processor, and a systolic insertion sorting algorithm running on a Xilinx XCV1000 FPGA co-processor at 66 MHz. Results in [18] show that having an FPGA as a co-processor improves the overall performance of the hybrid sorting algorithm up to a speed-up factor of 4. Furthermore, this speed-up factor is increased up to 100, using as many as 16 FPGA co-processors to sort 4096 keys.

Huang et al. propose a hardware design architecture in [4] for merge-sorting networks with a fixed size Batcher's sorting network. The number of sorting data N can be increased by modifying an address controller which controlls the memory accesses and increasing the amount of memory. However, performance of this architecture is considerably low for applications that require real time sorting performance such as in [13]. Our proposed architecture achieves a significant speed-up factor of 165 when compared to the architecture presented in [4].

A VLSI architecture presented in [14] sorts small continuous data sequences up to 256 keys and partially sorts longer sequences. Here, a regular architecture based on a chain of basic sorting units (BSU) is introduced which can handle a maximum operating frequency of 50 MHz.

The architectures in [14, 18] are based on comparative based sorting methods, therefore implementing [14, 18] on FPGAs would consume more FPGA internal resources (slices) than the proposed architecture.

In video processing, sorting is widely used in rank-order filters which are preferred over linear filters [19]. Here, the complexity of sorting is relatively small as the resolution and the number of data becomes small and no stable sorting is required. Several FPGA implementations for median filters have been presented [19–22] each of them exploiting different approaches for sorting architectures.

Maheshwari et al. [19] have taken a simple approach to a 3x3 median filter implementation on a Xilinx XC series FPGA. The elements in the 3x3 kernel are first sorted vertically, followed by horizontally, thereby producing the median in the center of the 3x3 kernel.

A bit serial sorting algorithm, similar to Quicksort, is introduced in [20], and a general purpose median filter is implemented on a Xilinx XC4010E FPGA. The implementation is very compact, occupying only 15 CLBs, but achieves real-time performance of 25 fps for 512x512 8 bits/pixel.

Hamid et al. [21] present a genetic algorithm on a Xilinx Spartan-II FPGA to optimize the tasks of the soft morphological filter (SMF). The genetic algorithm implementation essentially consists of a quick parallel sort algorithm, Biotonic Megasort. The maximum speed that the design can run is 56 MHz and it consumes about 87% slices in the FPGA.

Fahmy et al. [22] propose an efficient implementation of one dimensional (1D) median filters based on the cumulative histogram of the input samples. The implementation can process 72 MPixels/s for 8 bit pixel depth. The main drawback of [22] is that it is not extended to 2D median filters. Achieving timing constraints on the implementation may be very difficult for higher pixel depths, due to the nature of the priority encoder.

For sorting a large number of keys, although the comparative methods proposed in [19–22] can theoretically be extended, implementation of such methods on a resources constraint hardware such as FPGAs would not be feasible due to the large silicon area required.

2.3 A Modified Counting Sort Algorithm

In this section, we describe the proposed modified hardware-oriented sorting based on the counting sort algorithm [3] invented in 1954.

Table 2.1 lists the steps of a pseudo code of the original counting sort algorithm [3]. It starts sorting N integer elements by first counting the number of occurrences for every element in the input unsorted array, $A[0..N-1]$. The 'rank' of each element is then determined by counting the number of elements less than or equal to the element being considered, and is stored in the array $C[0..2^k-1]$. Each element in the input array is read from the 'right' and put in the array $D[0..N-1]$ (which contains the sorted elements) at the location stored in the array $C[0..2^k-1]$. At first, the array $C[0..2^k-1]$ provides the position of sorted array $D[0..N-1]$ for the first distinct element read from the array $A[0..N-1]$. If there are multiple occurrences for any element in $A[0..N-1]$, the next position is obtained by simply subtracting one from the corresponding location in $C[0..2^k-1]$, and the new result is stored back into the array $C[0..2^k-1]$. Once all the elements in the input array are read and scanned for the location of the array $D[0..N-1]$ from the array $C[0..2^k-1]$, the array $D[0..N-1]$ contains the stably sorted elements.

Table 2.1: Original Counting Sort Algorithm [3].

```
Input  : $A[0..N-1]$, $N$ integer keys in the
         range of $0, .., 2^k - 1$ to be sorted.
Output : $D[0..N-1]$, Sorted keys.
1:  // Initialize array $B[0..2^k - 1]$ to 0.
2:    $B[0..2^k - 1] \leftarrow 0$
3:  // Count same integers in $A[0..N-1]$
4:    for $i \leftarrow 0$ to $N-1$ do
5:       $B[A[i]] \leftarrow B[A[i]] + 1$
6:  // Initialize $C[0..2^k - 1]$ and set $C[0] = 0$
7:  // Compute the rank of each elements in $A[0..N-1]$.
8:    for $i \leftarrow 1$ to $2^k - 1$ do
9:       $C[i] \leftarrow B[i] + C[i-1]$
10: // Read each element from array $A[0..N-1]$
    // starting from right and place them in $D[0..N-1]$
    // at the position given by the content of $C[0..2^k - 1]$.
11:   for $i \leftarrow N-1$ to 0 do
12:      $D[C[A[i]]] \leftarrow A[i]$
13:      $C[A[i]] \leftarrow C[A[i]] - 1$
14:   return($D[0..N-1]$)
```

We propose two modifications to this algorithm so that it is well suited for a hardware implementation. First, the array $C[0..2^k - 1]$ is computed as

$$C(i) = \sum B(j) \quad \text{for} \quad i \in \{0, .., N-1\} \quad , \quad j \in \{0, .., i\}\,. \tag{2.1}$$

With this modification $C[0] = B(0)$, whereas in the original algorithm $C[0] = 0$.

Second, instead of using a decrementor in the step 13 of Table 2.1 ($C[A[i]] \leftarrow C[A[i]] - 1$), the modification requires an incrementor ($C[A[i]] \leftarrow C[A[i]] + 1$), which adds no additional cost in terms of hardware resources, timing or design complexities. The proposed modified algorithm is listed in Table 2.2.

The direct hardware implementation of the sorting algorithm [3] requires reading the array $A[0..N-1]$ in the reverse direction (right to left) as it was written to. For applications which require real-time performance, ability to stream data continuously

Table 2.2: Proposed Modified Counting Sort Algorithm.

```
Input  : $A[0..N-1]$, $N$ integer keys in the
         range of $0, ..., 2^k - 1$ to be sorted.
Output : $D[0..N-1]$, Sorted keys.
 1: // Initialize array $B[0..2^k - 1]$ to 0.
 2:    $B[0..2^k - 1] \leftarrow 0$
 3: // Count same integers in $A[0..N-1]$
 4:      for $i \leftarrow 0$ to $N-1$ do
 5:          $B[A[i]] \leftarrow B[A[i]] + 1$
 6: // Initialize $C[0..2^k - 1]$ and set $C[0] = B[0]$
 7:      for $i \leftarrow 1$ to $2^k - 1$ do
 8:          $C[i] \leftarrow B[i] + C[i-1]$
 9: // Read each element from array $A[0..N-1]$
    // starting from left and place them in $D[0..N-1]$
    // at the position given by the content of $C[0..2^k - 1]$.
10:      for $i \leftarrow 0$ to $N-1$ do
11:          $D[C[A[i]]] \leftarrow A[i]$
12:          $C[A[i]] \leftarrow C[A[i]] + 1$
13:      return($D[0..N-1]$ )
```

is an important factor. If a memory is read in the reverse order, then all the data in that memory must be read before the next data can be written, unless additional logic is implemented to handle reverse reading efficiently. This means that the overall throughput of the system is reduced by 50%. However, if the data can be read in the same order as it is written to (in a First In First Out (FIFO) fashion), then this problem is eliminated and design complexities and resources are both reduced. The proposed modifications to [3] allow us to use the array A as a FIFO, hence the proposed architecture is able to continuously stream unsorted keys without any timing or additional area overheads.

2.4 Proposed Sorting Architecture

This section describes the proposed architecture of a *single chip* FPGA implementation for the counting sort algorithm described in the previous section. Fig. 2.1 depicts the overall system level architecture, which takes the unsorted keys $x_i, i = 1, .., N$ as input and outputs stably sorted keys z_i. A more detailed block diagram is shown in Fig. 2.2. It can be seen that the implementation performs a number of tasks to stably sort a set of unsorted input keys. The input keys are buffered in the TEMPORAL KEY BUFFERS since all the data in a sequence is required to complete counting the same integers in $A[0..N-1]$ (building a histogram), which is performed in the HISTOGRAM UNIT. Once the histogram is computed, INITIAL ADDRESS CONTROL module reads the histogram and generates the initial starting addresses ($C[i]$, step 8 in Table 2.2). In the next step, the Direct Memory Access (DMA) reads the data from the TEMPORAL KEY BUFFERS and indexes into to the INDEXING RAM based on the address stored in the INIT ADDR RAM. The detailed description of each module is given in the next sections.

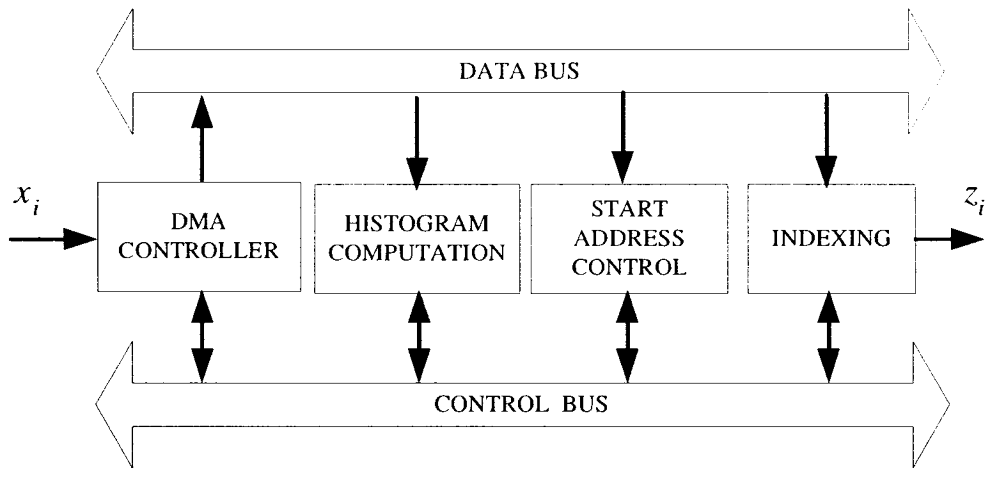

Figure 2.1: Overall System Level Sorting Architecture.

2.4.1 TEMPORAL KEY MANAGEMENT Module

The core of the TEMPORAL KEY MANAGEMENT module is the DMA controller that is responsible of moving input data between the memory and the appropriate processing nodes at high speed to sustain real-time performance. Our implementation performs

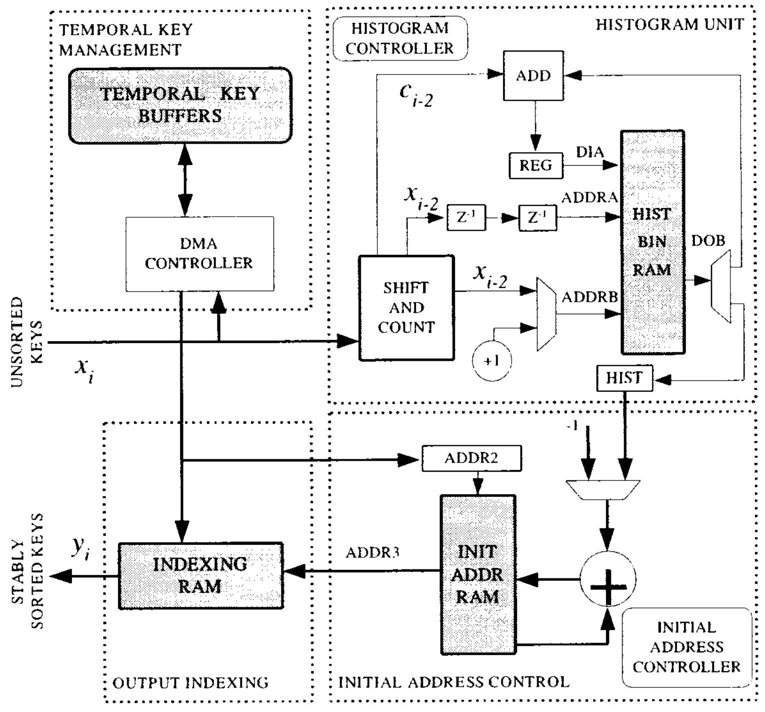

Figure 2.2: Detailed Circuit Diagram of the Sorting Architecture.

the histogram computation as new data is being arrived and are written into the TEMPORAL KEY BUFFERS by the DMA. While the previous set of data is being sorted at the back-end of processing, it is necessary to read those data from the TEMPORAL KEY BUFFERS while the new data is being written. Our DMA architecture manages these data transfers seamless to the other processing nodes.

2.4.2 Architecture of the HISTOGRAM UNIT

We implemented histogram computation using a number of concatenated dual-ported Block of RAMs (BRAMs) in order to support sorting higher resolution integers. The architecture is scalable and flexible since the implementation automatically computes the minimum required BRAMs and concatenates them at design compile time. The port A of the BRAMs is dedicated for histogram computation while port B is used for reading it out and clearing the content of the memory at the end of a sequence. The

SHIFT AND COUNT module counts co-occurrence of 3 consecutive keys so that reading data from an address of the RAMs is avoided while writing to the same location. The HISTOGRAM CONTROLLER controls all controlling logic required in the HISTOGRAM UNIT.

2.4.3 Architecture of the INITIAL ADDRESS CONTROL Module

The INITIAL ADDRESS CONTROL module is responsible for computing the starting addresses for each unsorted keys which are stored in the TEMPORAL KEY BUFFERS. These starting addresses, stored in the INIT ADDR RAM are used to address the INDEXING RAM of the OUTPUT INDEXING unit. For each histogram bin, the output of the HISTOGRAM UNIT is added to the content of the INIT ADDR RAM. Once the addresses for all the bins are computed, the INITIAL ADDRESS CONTROLLER triggers the DMA to output unsorted keys. For each unsorted keys, corresponding starting addresses are retrieved from the INIT ADDR RAM and used to address the INDEXING RAM. Furthermore, the content at the corresponding location in the INIT ADDR RAM is subtracted by one, which becomes the next starting address.

2.4.4 OUTPUT INDEXING Module

At this final sorting stage, the unsorted keys are written in the INDEXING RAM. The write addresses to the INDEXING RAM are obtained from the INITIAL ADDRESS CONTROL module. Once the DMA has transferred a complete set of keys, the content of the INDEXING RAM is stably sorted keys, which are read by the OUTPUT CONTROLLER sequentially. Moreover, OUTPUT CONTROLLER clears the content of the INDEXING RAM at the end of each sequence.

2.5 Simulation and Verification Results

The modified counting sort algorithm was implemented in C on a PC with an Intel Pentium-IV 2.4 GHz processor for performance comparison between software and hardware implementations. Simulation results are shown in Fig. 2.3 for $k = 12$, where k is the number of bits in N. The execution time for the FPGA implementation includes data load and result unload time. The proposed FPGA implementation achieves greater than an order of magnitude performance improvements over the software implementation for $N > 10000$.

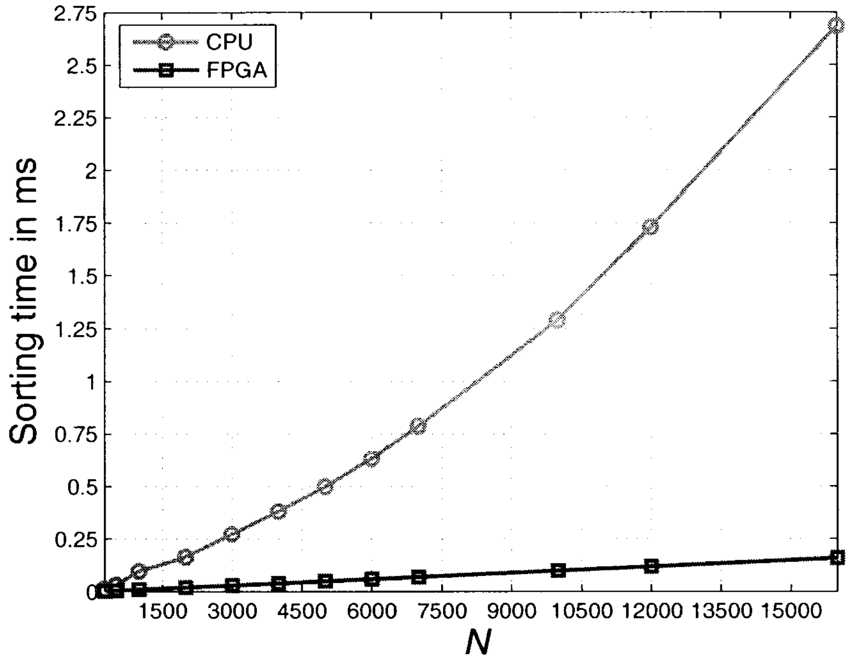

Figure 2.3: Complexity comparison between a CPU and FPGA.

The video noise estimation algorithm [13], written in C, produces variances $V_i, i = 1, .., N, N \in \mathbb{Z}$ of 3x3 or 5x5 blocks in an image, which are required to be sorted for further processing. As it can be seen from Fig. 2.4, a part our verification process consists of using these variances as test stimulus x_i to the VHDL testbench of the proposed sorting design, Unit Under Test (UUT), through the MATLAB environment.

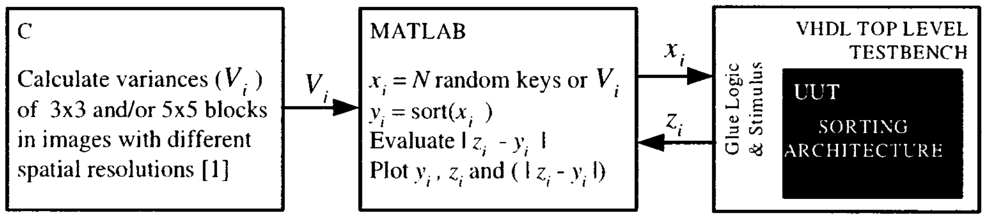
C
Calculate variances (V_i) of 3x3 and/or 5x5 blocks in images with different spatial resolutions [1]
V_i
MATLAB
x_i = N random keys or V_i
y_i = sort(x_i)
Evaluate | z_i - y_i |
Plot y_i, z_i and (| z_i - y_i |)
x_i
z_i
Glue Logic & Stimulus
VHDL TOP LEVEL TESTBENCH
UUT
SORTING ARCHITECTURE

Figure 2.4: Simulation environment with VHDL Testbench and C/MATLAB.

Further verification was performed by using MATLAB to generate N random keys, and setting them as stimulus for the UUT. The sorted keys z_i from VHDL environment are then compared with the appropriate MATLAB sorted keys y_i. We have extensively run this simulation with different number of N and verified the accuracy of our implementation. For instance, we obtained large and different amount of elements (V_i) with [13] by using images with different spatial resolutions and setting the block size to 3x3 or 5x5.

A Virtex II-Pro evaluation board was used for actual hardware verification. We sent x_i and read-back z_i through a parallel port and result were successfully verified with y_i.

Fig. 2.5 reveals that the sorted elements z_i obtained from the proposed implementation are identical to the MATLAB sorted elements y_i.

2.6 Synthesis and Implementation Results

2.6.1 Implementation

The proposed architecture was synthesized with Synplicity Synplify 7.3. and placed and routed targeting an XC2VP20 FPGA with Xilinx ISE 6.3 Alliance place and route tool. The synthesis and place and routed implementation results indicate that the complete architecture is very compact occupying only about 15 % of slices and about 30 % of the BRAMs for 12 bits wide (integer or fractional), 16K (16384) keys.

Furthermore, the proposed design was synthesized for various values of k, and Fig. 2.6 depicts the FPGA resources utilization. Note that the proposed architecture is single-chip solution for sorting a large number of keys and it uses internal BRAMs for local data storage. Hence, BRAM utilization is directly proportional to k, however the slices usage remain very low. The proposed architecture was easily routed constraining a 7.5 ns clock cycle, hence the design can be clocked at 133 MHz with the slowest FPGA speed grade (-5). At a 133 MHz clock, Xilinx XPower tools to estimated 1.8 W power dissipation of our design for a toggle-rate of 50 % .

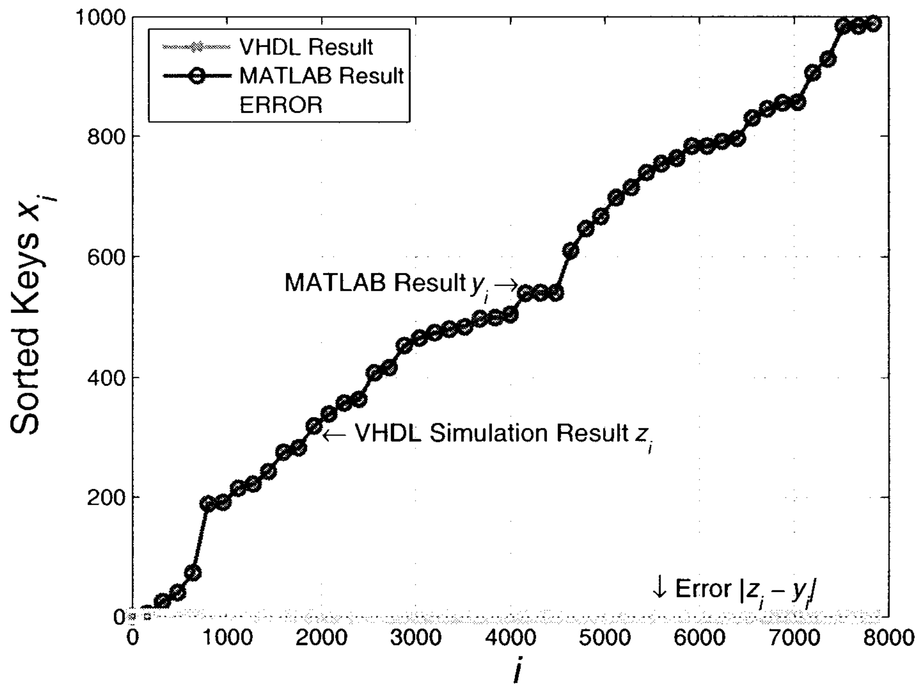

Figure 2.5: Sample comparison between VHDL and MATLAB results.

2.6.2 Comparison to the Existing Methods

Table 2.3 lists performance and architectural differences between the proposed method and the architecture described in [4]. Notice that [4] is a $0.35\mu m$ CMOS based ASIC architecture, therefore timing and area optimization can be performed better than

Table 2.3: Performance and area comparison for sorting 128 keys of 8 bits with [4] and proposed method.

	Ref [4]	**Proposed**
Max Clock	50 MHz	133 MHz
Sorting Time	158.72 μs	0.962 μs
Area (Memory)	128 Bytes	640 Bytes
Gates	3185 Gates	approx. 9000 Gates
Architecture	0.35μm CMOS	FPGA

on an FPGA. For comparison purpose we have chosen 128 keys of 8 bits for our implementation and verification in order to match with the test vectors used in the reference method [4]. As it can be seen from Table 2.3, the performance of our proposed method is significantly higher (165 times faster) than [4]. Although the area complexity of the proposed method is higher than in [4], our implementation does not utilize considerable amount of resources on presently available FPGAs.

Furthermore, the proposed architecture contains many advantages over the sorting architecture presented in [18], which requires a CPU processor in addition to an FPGA to complete the sorting. For sorting very large number of keys in real-time, the technique described in [18] would require multiple FPGAs. In contrast, our proposed architecture is a compact and based on a single chip FPGA device which is capable of sorting a large number of keys.

The comparative methods presented in [14,19–22] can be extended to sort a large number of keys on a resources unconstrained hardware platform such as a dedicated ASIC. However, this would increase both the cost and complexity of the overall implementation, specially when sorting is a very small part of a complete application such as in video processing [13,40].

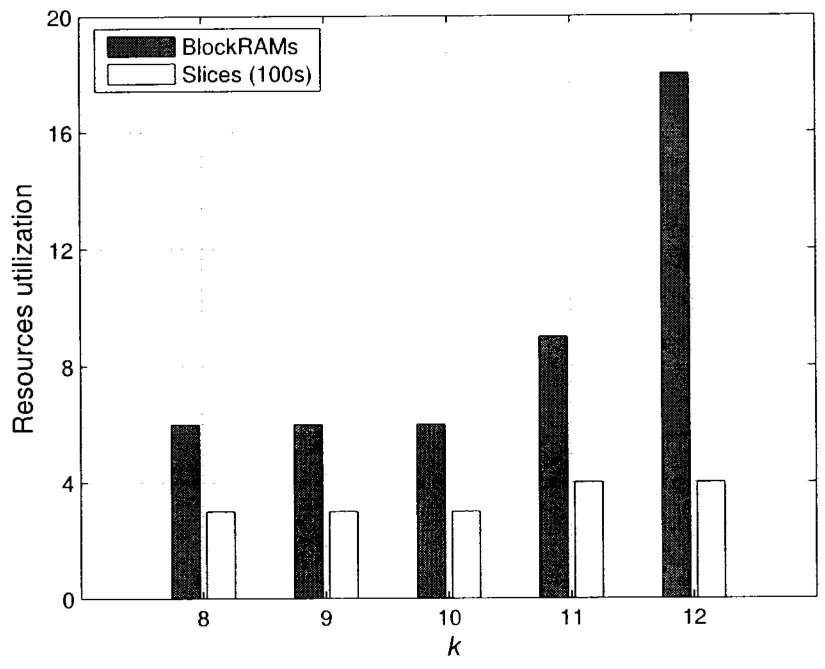

Figure 2.6: Internal FPGA resources utilization.

2.7 Chapter Summary

In this chapter, we proposed a modified counting sort algorithm and a novel, scalable and compact single chip FPGA architecture for stably sorting a large volume of integer and fractional keys in real-time for video processing applications. The proposed architecture of the sorting is scalable to gain higher throughput by trading off only the FPGA internal memory resources. The proposed implementation was successfully verified on an actual hardware platform that consists of a Xilinx XC2VP20 FPGA. Sorting performance comparisons showed that the proposed implementation is significantly better than the existing methods. Furthermore, when compared to pure software implementation for large N, we obtained a speed-up factor of more than 10 with the proposed architecture. In addition, an efficient noise estimation implementation which eliminates the need of SRAM was proposed.

Chapter 3

Improved FPGA-based Implementation of Noise Estimation

In this chapter, we outline an improved FPGA-based noise estimation implementation of Lapalme's et al. [5]. The focus of this thesis is to implement and integrate all of the core video processing algorithms (video noise estimation, moving object detection, and contour analysis) required in the moving video object segmentation [1], into one FPGA. As such, it is imperative to device an efficient hardware architecture.

3.1 Introduction

The effectiveness of video processing systems heavily depend on the amount of unwanted noise contained in a corrupted video signal due to, for e.g., video acquisition, processing, storage or transmission. Hence, reducing the noise in video signals by means of noise estimation algorithms is an active research field. The state-of-the-art noise estimation methods calculate the level of noise, and facilitate the video processing algorithms to enhance the quality of video sequences to the amount of noise present. Thus, a noise estimation algorithm, such as [13], which reliably estimates noise both in highly noisy and good quality images, is imperative to demodulate vital content from a corrupted video sequence. Intra-field and block-based noise estimation method proposed in [13] finds the noise variances of a set of blocks classified as the most homogeneous blocks to estimate the global image noise variance. The algorithm [13] requires to sort large amount of data, which is the primary computational bottleneck in its hardware as well as software implementations.

3.2 Related Work Review

Except the architecture proposed in [5], no previous studies have been carried out on FPGA-based implementation of noise estimation algorithms, to the best knowledge of the author. Although, Lapalme's FPGA implementation [5] of the reliable spatial video noise estimation algorithm [13] runs real-time, it requires a significant amount of FPGA resources and runs at a slower speed. The motivation to improve the architecture [5], by means of the proposed modified counting sort algorithm comes from the need to integrate the proposed improved noise estimation implementation with other high-level video processing tasks, described in the chapters 4 and 5, within a single FPGA device.

3.3 Overview of the Reference Noise Estimation Algorithm

In this section we outline the noise estimation algorithm [13]. The spatial noise estimation method [13], which produces reliable estimates, finds homogeneous regions by taking image structure into account. To reject structured blocks the algorithm [13] first divides the image into equal blocks of the size $W \times W$. For each block, a homogeneity measure ξ_{Bh} is then computed using eight high-pass directional filters. The mask of the horizontal direction scanning window on the image function I for $W = 5$ is illustrated in Eq. 3.1.

$$I_o(i) = -I(i-2) - I(i-1) + 4 \times I(i) - I(i+1) - I(i+2). \tag{3.1}$$

The homogeneity measure ξ_{Bh} is obtained by adding the absolute values of all eight high-pass directional filters.

The variance σ^2_{Bh} of each block is then calculated with

$$\sigma^2_{Bh} = \frac{\sum\limits_{(i,j)\in W_{ij}} (I(i,j) - \mu_{Bh})^2}{W \times W}. \tag{3.2}$$

Here, μ_{Bh} is the sample mean defined as

$$\mu_{Bh} = \frac{\sum\limits_{(i,j)\in W_{ij}} I(i,j)}{W \times W}. \tag{3.3}$$

The homogeneity measures ξ_{Bh} is sorted and variances σ^2_{Bh} corresponding to the top 10% of sorted ξ_{Bh} are selected. However, only the variances satisfying the condition given in Eq. 3.4 are averaged to obtain the global noise variance (estimation) σ^2_n.

$$|log(\sigma^2_{Bh}) - log(\sigma^2_{REF})| < t_\sigma. \tag{3.4}$$

In Eq. 3.4, σ^2_{REF} is defined as a reference variance, which chosen as the median of the variances of the three most homogeneous blocks, and t_σ is a user defined threshold value.

3.4 Proposed FPGA-based Architecture

The improvements achieved to the FPGA implementation presented in [5], primarily with the proposed modified counting sort algorithm described in sections 2.3 and 2.4 are outlined next. Fig. 3.1 illustrates the overall architecture of the noise estimation algorithm. Primary improvement was obtained by adopting the proposed modified counting sort algorithm described in sections 2.3 and 2.4 to [5]. The FPGA implementation in [5] requires two external SRAM memory modules for sorting. SRAMs are costly and increase the physical area and power consumption of the overall system. Thus it is highly desirable to *minimally* use these devices, except for other processing tasks, which inherently require the utilization of SRAMs for their implementation, such as HIGH-SPEED CACHE for contour tracing and filling described in chapter 5. Moreover, [5] implements logarithmic computation by means of a large look-up table, which requires more than 53% of BRAMs in the implemented FPGA device. In the proposed implementation, we employ an efficient logorithmic arhitecture [41], which utilizes *zero* BRAMs but only two multipliers and few slices.

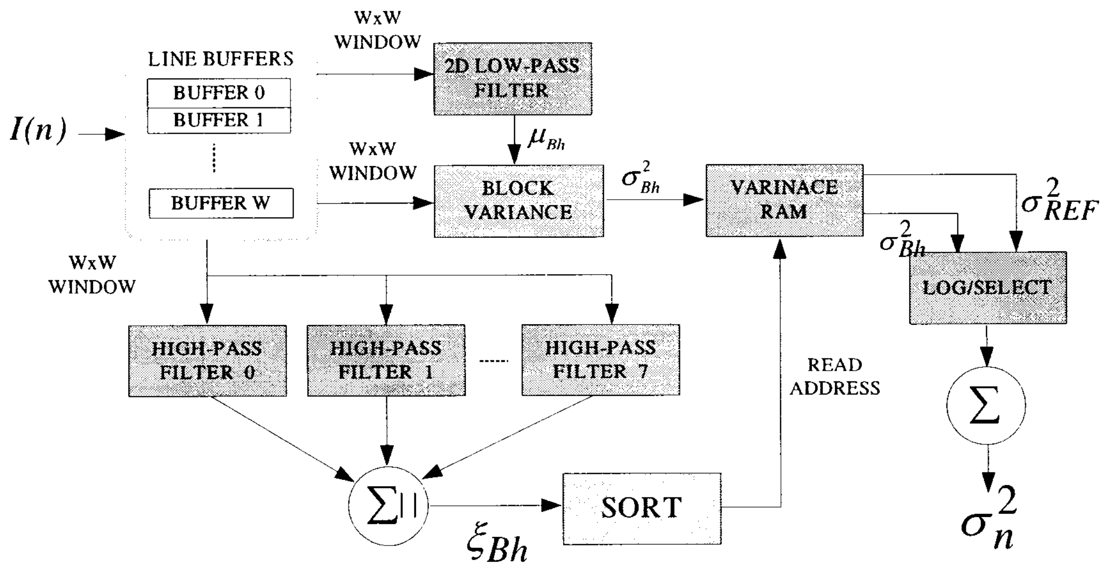

Figure 3.1: Proposed Architecture of the Noise Estimation Algorithm.

As can seen in Fig. 3.1, LINE BUFFERS are utilized with BRAMs, which generate $W \times W$ blocks. In the 2D LOW-PASS FILTER block, the sample mean μ_{Bh} is produced and passed BLOCK VARIANCE module which computes the variance σ^2_{Bh}. These block variances are then stored in the VARIANCE RAM. A set of eight HIGH-PASS FILTERS produces directional filters, and absolute value of the result of the directional filters are summed to produce the homogeneity measures ξ_{Bh}. The proposed modified counting sort algorithm described in detail in sections 2.3 and 2.4 is implemented in the SORT module, which sort the ξ_{Bh}, and indexes of the 10 % most homogeneous blocks are sent as the read address to the VARIANCE RAM, which outputs corresponding μ_{Bh} and μ_{REF}. LOG/SELECT block finds the logarithmic value of μ_{Bh} and μ_{REF}, which are compared to an application-dependent threshold as seen in Eq. 3.4 to select only the valid varinces. These varinces are are then summed in the accumulator to obtain the global noise variance (estimation) σ^2_n.

3.5 Implementation Results

Although the proposed architecture is ultimately implemented on an Xilinx Virtex-4 FPGA, in order to do a fair comparison, we synthesized the our architecture to the same FPGA (XC2V4000) used in [5]. Table 3.1 lists performance and architectural differences between the proposed method and the architecture described in [5].Table 3.1 exemplifies that the timing performance of our proposed method is improved by three fold. Furthermore, our method does not require external SRAMs, which brings many advantages as noted in 2.4, instead it utilizes FPGA internal BRAMs for the proposed sorting implementation.

Table 3.1: Architectural and Performance Comparison of the Noise Estimation Implementation between [5] and the Proposed Method.

	Lapalme et al. [5]	**Proposed**
Max Clock	40.5 MHz	127 MHz
BRAMs	72	89
Slices	4600	5200
External SRAM	Required	No

3.6 Chapter Summary

In this chapter, we proposed an improved FPGA-based architecture of video noise estimation. Although the proposed implementation requires slightly more BRAMs and slices than the referenced implementation, it achieves a three-fold performance improvements and avoids utilizing external SRAM modules which increase the area, cost, power and verification cycle of the overall physical system. Furthermore, the proposed noise estimation implementation is integrated with other high-level post video processing algorithms forming a real-time video processing system for moving video object segmentation.

Chapter 4

FPGA-based Implementation of Spatio-Temporal Object Detection

4.1 Introduction

Object detection plays a key role in many video processing applications such as surveillance or machine vision. However, the computational complexity involved in object detection makes it difficult to achieve real-time performance on a general purpose CPU or DSP. There exists three main architectural approaches to this challenge 1) Application Specific Integrated Circuit (ASIC), 2) parallel computing, and 3) FPGAs. Evolving high density FPGA architectures such as those with embedded multipliers, memory blocks and high I/O (input/output) pin count make FPGAs an ideal solution in video processing applications [11, 42, 43].

The work in [23] demonstrates how a number of image detection algorithms can be implemented on FPGAs. The dynamic reconfigurability feature of the FPGAs allows to reconfigure a part or complete FPGA within a fraction of a microsecond, and the paper shows how multiple image processing algorithms can be sequentially applied to the image by using the same FPGA.

Another FPGA implementation for segmenting text in images is in [24]. Experimental results show that this algorithm implemented in FPGA achieved a speedup of close to 250 compared to a general purpose CPU implementation. However, this implementation runs at 5 MHz which is well below the real-time performance.

The study in [25] partially involves FPGA-based implementation of image detection based on the resistive-fuse network model.

An extensive comparison between FPGA and DSP implementations of image classifier for object detection is in [26]. Although the performance of the FPGA implementation significantly overpasses that of the DSP implementation, its performance and scalability is heavily limited and embedded by the hardware platform chosen.

4.2 Overview of the Reference Spatio-Temporal Object Detection Algorithm

An object detection method categorizes pixels of a frame into two regions: object pixels and background pixels. Video object detection methods [12] can be classified based on their automation, spatial accuracy, temporal stability, and computation load. Computationally expensive methods give, in general, accurate results while low-computation methods may fail. However, few of the methods are tested on a large number of videos, throughout long videos, on noisy videos, and without parameter tuning. We select the non-parametric detection method in [1] due to its low computation and noise and temporal stability. These features forgo spatial accuracy, e.g., at object boundaries. Such a method is most appropriate to applications, e.g., video surveillance, where stability under varying conditions is of more concern than accurate object boundaries. Furthermore, the method described in [1] is well suited for hardware implementation such as a modern FPGA due to its modularity, simplicity and reduced resources requirements. Fig.4.1 illustrates the block diagram of the method [1] which consists of three main modules: motion detection, spatio-temporal thresholding, and morphological edge detection.

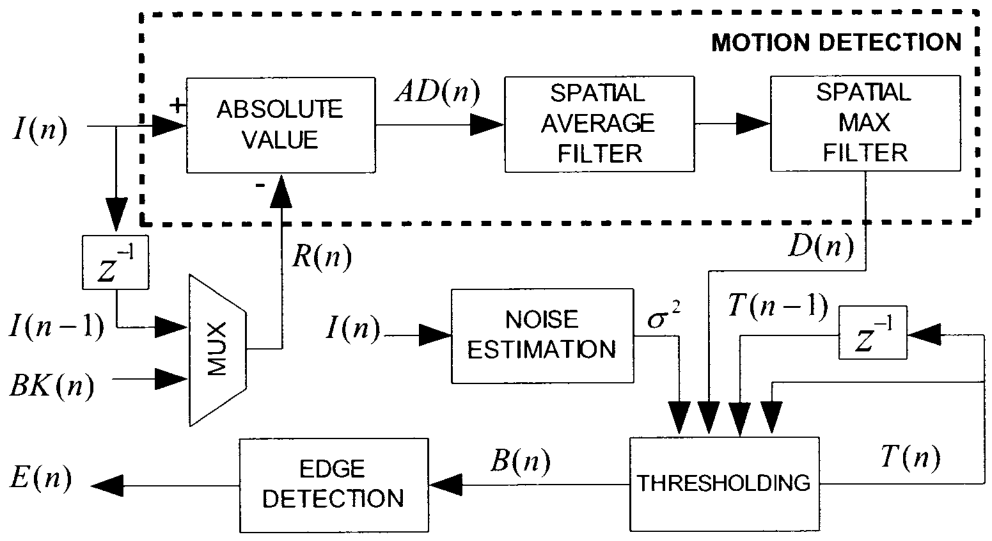

Figure 4.1: Block Diagram of the Object Detection [1].

The motion detection finds first the absolute frame difference, $AD(n)$ at time instant n, between the current $I(n)$ and a reference frame $R(n)$. $R(n)$ can be either a background $BK(n)$ or the previous frame $I(n-1)$ in a video sequence. $AD(n)$ is then spatially filtered by both an average and a max filter.

In the thresholding module, a global spatial threshold T_g is first computed as follows. The spatially-filtered frame $D(n)$ is divided into K consecutive non-overlapping blocks, $W_k, k \in \{1, ..K\}$. The histogram of each W_k is split into L equal sections. The most-frequent gray-level g_{pl} of each histogram section is found and

$$T_g = \frac{\sum_{k=1}^{K} (\lambda_l + \mu_k)}{K.L + K} \tag{4.1}$$

where $\lambda_l = \sum_{l=1}^{L} g_{pl}$ and μ_k is the pixel average of W_k. Note that T_g is obtained using block local and global data. T_g is then proportionally adapted to the noise variance σ^2 using

$$T_\varsigma = T_g + a.\sigma^2 \tag{4.2}$$

where $0 < a < 1$ and σ^2 is estimated using [13]. This noise-adapted T_ς is then quantized to maintain spatio-temporal stability where quantization down to three levels yields good results [1]. The quantized threshold T_q is passed through a memory system that holds the threshold of the previous frame and determines the threshold $T(n)$ based both on new quantized threshold T_q as well as the previous threshold $T(n-1)$. Finally, $D(n)$ is globally thresholded by $T(n)$ creating a binary frame $B(n)$.

To extract object boundaries, the edges $E(n)$ in $B(n)$ are detected in the morphological edge detection module. Here, a 2x2 square kernel is moved over the entire $B(n)$ and if the result of Boolean AND operation on these four binary pixels is false, then the output pixels are set to white if their corresponding pixels in the input frame are white, otherwise output pixels are set to black [1]. The method in [1] requires $E(n)$ to pass through a contour tracing and filling algorithm to label the objects, which is described in detail in chapter 5.

4.3 Proposed Pipelined Architecture and Implementation

The overall system level architecture of the FPGA design is illustrated in Fig. 4.2. It consists of a Direct Memory Access (DMA) module and three main processing blocks for motion detection, spatio-temporal thresholding and morphological edge detection.

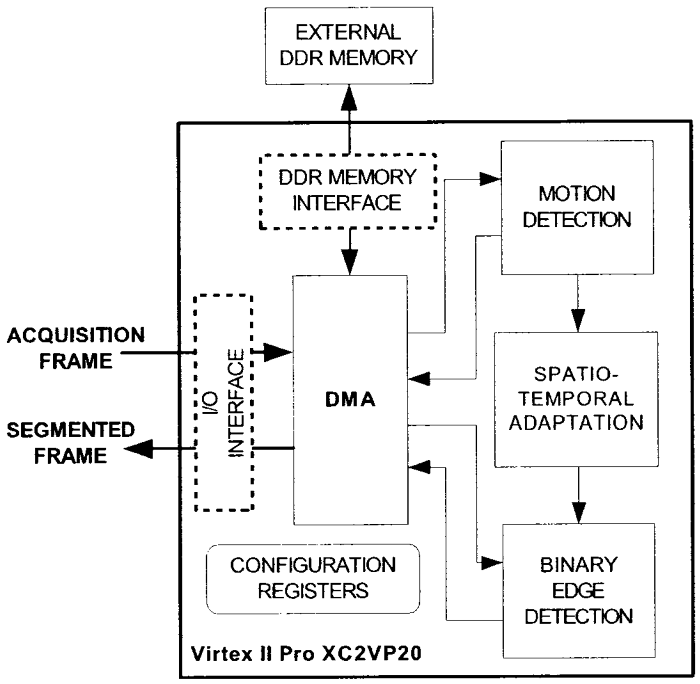

Figure 4.2: System-Level Architecture of Object Detection.

4.3.1 Proposed DMA Architecture

An efficient management of data transfers within a system is the key to any real-time hardware implementation. In our implementation, we designed a scalable and versatile DMA architecture that can be easily configured by a simple set of registers. The proposed DMA consists of 4 KB deep First In First Out (FIFO) memories connected to each read and write DMA channels, a DMA controller (DMACTLR) to manage these FIFOs, and a DDR memory controller. A write transfer to the memory is initialized by filling the corresponding write FIFO (up-to a maximum of 2 KB), and sending a request to DMACTLR. Whenever a read FIFO is half empty, a read request is automatically initialized. An internal cache memory is used to store the addresses and transfer descriptions of each DMA channel. A round-robin arbitrator arbitrates all parallel requests from each channel and serves the selected DMA channel.

Our architecture for motion detection and the DMA is scalable in that motion detection can be configured into the two modes (background $BK(n)$ and previous

$I(n-1)$ frame) on-the-fly. In the former case, the DMA is programmed to store the acquired frame as the background frame in the memory and it continuously read the background frame and sends it to the motion detection module along with $I(n)$. In the later case, the DMA transfers newly arrived $I(n)$ to the memory for future processing as well as to the motion detection module. At the same time, the DMA reads $I(n-1)$ that was stored in the memory (during the last frame time) and sends it to the motion detection module. The output frame $D(n)$ of the motion detection is routed back to the memory and to the spatio-temporal thresholding node. The spatio-temporal thresholding block takes a full frame time to compute a threshold, hence it is necessary to buffer the frame being processed in the memory until a valid threshold is available. Within this duration, the previous motion-detected frame is read from the memory and is sent to the last processing block for morphological edge detection. The proposed DMA architecture manages all these massive data parallelism in such a way that is seamless to any of the processing blocks.

4.3.2 Scalable Motion Detection Implementation

The absolute difference frame $AD(n)$ is computed by a simple subtractor and its absolute value is routed to the spatial average and max filters. We architectured the spatial filters to be flexible and scalable in number of ways: 1) our implementation can change the size of the both filters from any configuration between 1x1 and 5x5 on-line, and 2) the frame resolution is programmable allowing to support different video cameras. The architecture is designed in a modular manner, so that future design expansions can be easily feasible. For instance, if the design has to support a video camera with more than 2 KB line width, it can be achieved by using multiple instances of the existing modules. We also minimized the memory bandwidth that would require to write and read previous lines for two-dimensional filters by using internal BRAMs as line buffers.

4.3.3 Spatio-Temporal Thresholding Architecture

The high-level architecture designed for the spatio-temporal thresholding is shown in Fig. 4.3. Notice the the noise variance σ_n^2 is obtained from the architecture presented in the chapter 3.

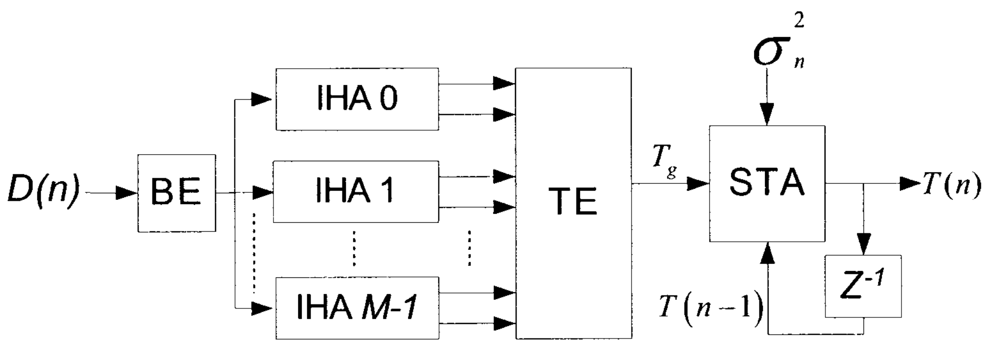

Figure 4.3: High-Level Architecture of Spatio-Temporal Threshold.

The novelty of this architecture is that it does not require any external memory to extract the individual blocks. The block extractor (BE) splits the motion-detected data into M vertical blocks, which are then fed into M Intensity Histogram Analysis (IHA) modules. Each IHA generates μ_k and λ_l for the corresponding block. The Threshold Estimator (TE) takes those values to produce T_g for Spatio-Temporal Adaptation (STA) module. The STA consists of an adder that adds T_g to a weighted value of noise variance to get T_ς, and two priority encoders. The first encoder produces T_q by quantizing T_ς down to three quantization levels which are defined with three user programmable registers. The second priority encoder selects $T(n)$ according to T_q and $T(n-1)$.

Architecture of the IHA Module

The IHA consists of two main processing nodes - Intensity Average and Histogram Analysis which compute μ_k and λ_l respectively, and a Controller and an Address Generation unit, that generates the signals required to control these processing nodes. The overall architecture is shown in Fig. 4.4. In the Intensity Average module, we

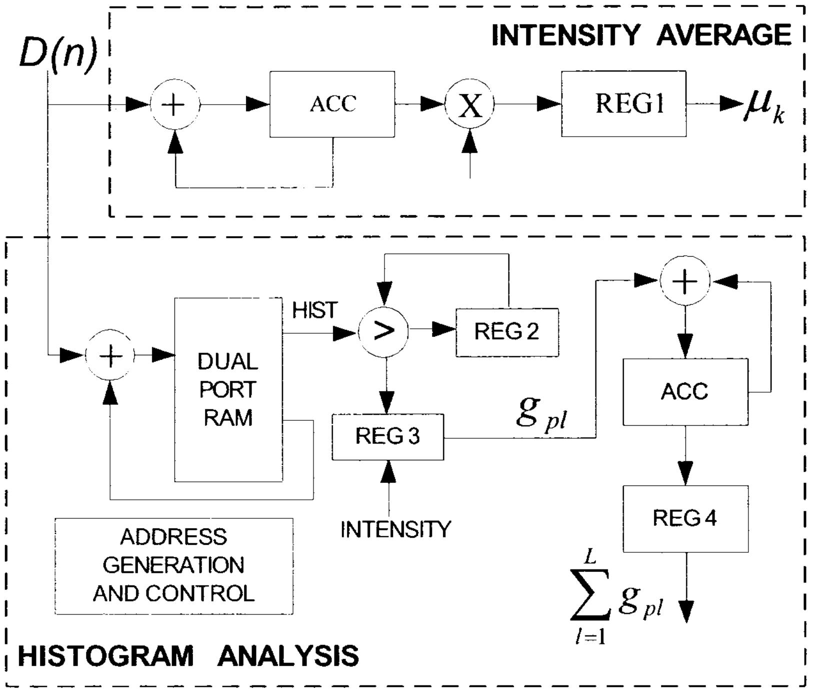

Figure 4.4: Architecture of Intensity Histogram Analysis Module.

used a multiplier as a divider to obtain the average value. Hence, the resources usage is minimal and the result of the average is obtained with less pipelined delay when compared to a pure divider usage. Histogram Analysis block first calculates the histogram of the input frame using a BRAM and an adder. After the entire frame data for a $W \times H$ block is entered, the histogram will be available in the BRAM. When the histogram is sequentially read, Reg 2 holds the maximum value within an interval $l, l \in \{1, ..L\}$, and Reg 3 keeps the corresponding gray value, g_{pl}. Once the complete histogram is read, g_{pl} is accumulated over the entire intervals, and the result of λ_l will be stored in the Reg 4.

Architecture of the TE Module

The architecture of the threshold estimator is shown in Fig. 4.5. The inputs, μ_k and

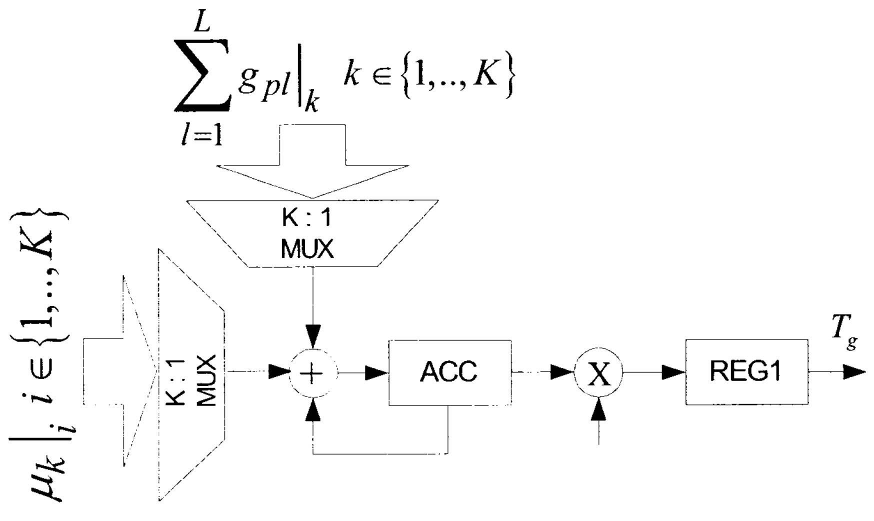
$\sum_{l=1}^{L} g_{pl}\big|_k \; k \in \{1,..,K\}$
K : 1
MUX
$\mu_k\big|_i \; i \in \{1,..,K\}$
K : 1
MUX
+
ACC
X
REG1
T_g

Figure 4.5: Threshold Estimator Architecture.

λ_l, to the TE block arrive in serially. This allows us to use two multiplexers to select the appropriate operands to the accumulator, which minimizes the resources usage. After all the data of an entire frame has arrived, T_g, will be available in the REG1.

4.3.4 Morphological Edge Detection Architecture

The architecture of the morphological edge detection is shown in Fig. 4.6. The Morphological Engine (ME) evaluates the Boolean condition (see Section 2). On the output, we configured an internal BRAM as a Dual Port Line Buffer (DPLB) to keep track of the partially estimated edge frame, hence ME can access and modify the content of the DPLB without degrading its access time. Once ME has accessed and modified the entire line in the DPLB twice, the content of the DPLB is $E(n)$. This is read from DPLB and sent to the DMA to transfer to the output port of the FPGA.

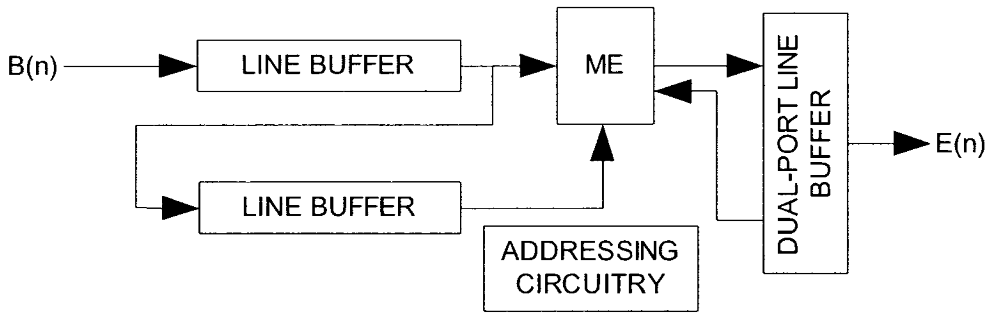

Figure 4.6: Circuitry for Morphological Edge Detection.

4.4 Design Verification, Synthesis and Implementation Result

4.4.1 Verification

We simulated the proposed design with a video sequence, "Hall" (300 frames of 352 pixels x 288 lines) to thoroughly verify the integrity of our implementation. We used a background frame as a reference (see Fig. 4.1). Fig. 4.7 is an example result obtained with the FPGA simulation and the reference C software implementation which subjectively reveals that both results are closely identical.

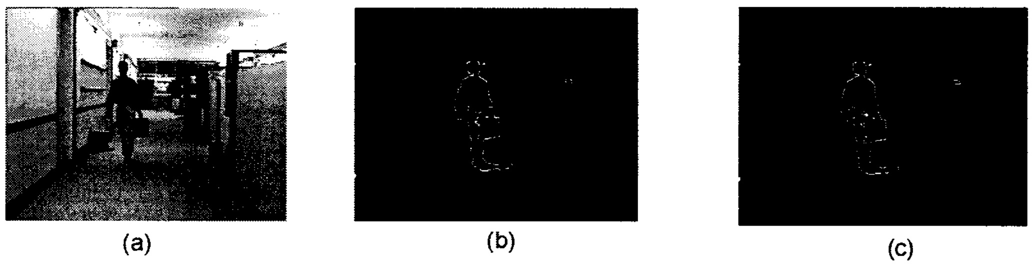

Figure 4.7: (a) 54th frame in the captured video sequence, (b) segments with the reference C implementation, and (c) FPGA segments.

In addition, we used two objective measures to compare our implementation results $E_{hw}(n)$ with the reference C results $E_{sw}(n)$.

Product of Correctly Classified Proportions, PCP [2] is a widely used objective measure to evaluate binary images, and is calculated as follows: The basic of a classification based objective measure is C_{AB}, the number of pixels belonging to class A that have been classified as class B. For binary images, $(A, B) = (0, 1)$, and c_{11} is true positives (TP), c_{00} is the true negatives (TN), c_{01} is the false positives (FP), and c_{10} is the false negatives (FN). PCP is then:

$$PCP = \frac{TP + TN}{TP + FP + TN + FN} \tag{4.3}$$

Here $E_{sw}(n)$ serves as the ground-truth data. We can see in Fig. 4.8(a) that $E_{hw}(n)$ is extremely close to $E_{sw}(n)$. Notice that if PCP = 1, then both results are identical $(E_{hw}(n) = E_{sw}(n))$ and if PCP = 0, then they are inverse $(E_{hw}(n) = \overline{E_{sw}(n)})$. We also computed the sum of the pixels that are different between the results of the proposed implementation and reference C implementation as $\Delta_{hw} = \sum |E_{hw}(n) - E_{sw}(n)|$, and Fig. 4.8 (b) shows that maximum of Δ_{hw} is 15 pixels.

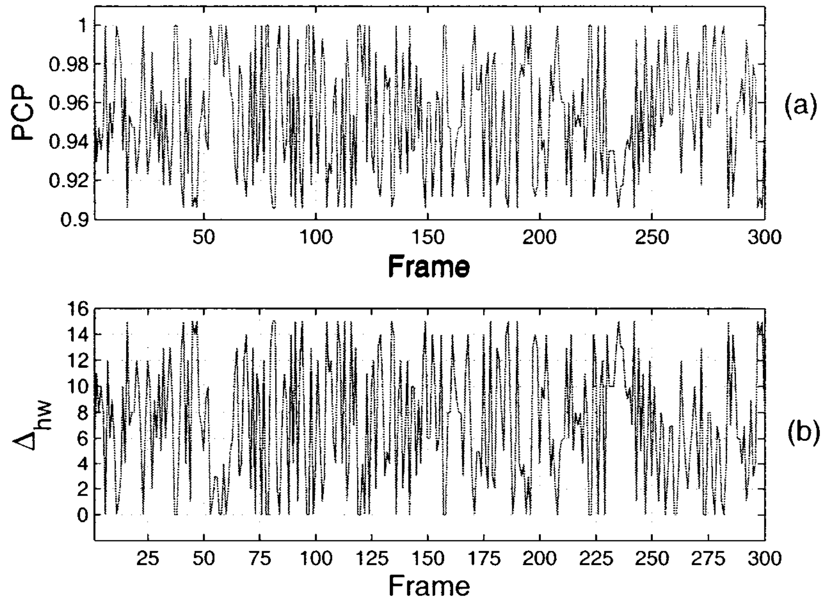

Figure 4.8: (a) Comparison between software and hardware implementations with PCP objective measure [2], and (b) difference of total pixels between software and hardware implementations Δ_{hw}.

We have also successfully tested the accuracy and performance of the proposed implementation on an actual FPGA of a frame grabber using a high-speed camera.

4.4.2 Synthesis Result

We have designed and simulated the proposed FPGA architecture for the spatio-temporal object detection algorithm in VHDL and synthesized with Synplicity Synplify 7.3. The synthesized design was then placed and routed for XC2VP20 FPGA with Xilinx ISE 7.1 Alliance tool. The implemented design occupies approximately 60% of the area of an XC2VP20 FPGA (37 % of slices, 27 % of LUTs (look up tables), 59 % of BRAMs, and 9 % of multipliers). Xilinx XPower tools estimated the power dissipation of the implementation to be less than 2 W for a toggle-rate of 50 % . The design is easily able to run up to 133 MHz, which means that it takes only 7.5 ms to complete object detection of 1024x1024 frame (including input frame load and result frame unload timing), which is more than sufficient for current and near future video processing applications.

4.4.3 Comparison to the Existing Methods

The architecture presented in [24] runs at 5 MHz and it takes 360 ms to segment a 1024x1024 frame. In contrast, our architecture achieved a significant higher clock rate of 133 MHz, hence it takes only 7.5 ms to complete segmenting a 1024x1024 frame, including input read and output write timing. Moving data between memory and FPGA affects the scalability and the overall performance of an implementation. Our versatile DMA architecture is more generic and scalable than the data movement procedure presented in [26].

4.5 Summary

This chapter proposed a novel, robust, scalable and modular FPGA architecture for real-time spatio-temporal object detection. We used advanced design techniques such as heavy pipelining and data parallelism, hence achieved an optimal speed of 133 MHz while utilizing minimal hardware resources. Furthermore, our architecture avoids many re-designing efforts by its inherent scalability and adaptivity, for instance, many of the algorithm specific parameters such as spatial filter size can even be programmed on-the-fly.

Chapter 5

A Real-Time Implementation of Chaotic Contour Tracing and Filling of Video Objects on Reconfigurable Hardware

5.1 Introduction

Contour tracing is a method that links connected neighborhood pixels in a binary edge frame, whereas contour filling fills the area inside a contour with a specific integer value, uniquely labeling each objects in an image. Contour tracing and filling are a fundamental element in many video and image processing applications such as video surveillance [1], medical image processing [15], object based video coding, e.g., MPEG-4 and MPEG-7, [44], computer vision [16] and pattern recognition [17].

Although solutions for robust contour tracing and filling methods have been considerably investigated using the state-of-the-art general purpose sequential processor-based systems [1, 6, 8–10], these software-based implementations are too slow to achieve real-time performance due to the computational complexity involved in the contour tracing and filling algorithms. As such, an efficient hardware acceleration is inevitable. Traditional full custom Application Specific Integrated Circuits (ASICs) suffer from longer development time and expensive engineering cost. Parallel processing machines consume enormous amount of power and occupy large physical area. In contrast, emerging FPGAs with embedded multipliers, memory blocks and high pin counts, are increasingly employed on hardware platforms in many signal/video processing applications [11]. Moreover, FPGA reconfigurability is an attractive feature which allows the system to be adopted for another purpose.

5.2 Related Work

A full custom VLSI CMOS design for extracting contours is presented by Agi et al. in [27]. Here, authors attempt to minimize the memory usage by partitioning the input frame into smaller regions and distributing these regions to an array of processing elements (PEs). Each PE in [27] consists of its own memory and a contour tracing unit, and uses a 2x2 window for extracting partially completed contour lists. However, the method described in [27] fails to produce completed contour tracing, unless a full object can be completely stored in the relatively small processing memory.

Chia et al. [28] propose a parallel VLSI architecture which consists of $N + 1$ processing elements for generating the chain codes of object contours in a binary frame with N raws. The algorithm proposed in [28] can complete contour extraction in $3N$ cycles, assuming the input binary frame is already stored in memory. However, in order to complete tracing in $3N$ cycles, [28] requires simultaneous reading of all N raws and simultaneous writing of chain codes to memory. Moreover, final contours are generated by accessing memory in a random fashion. Contour tracing methods inherently involves random data movements between memory and contour extraction unit(s). Thus, performance and feasibility of implementing a given contour tracing architecture or algorithm depend heavily on the efficiency of the memory and the robustness of the memory data accessing mechanism. Hence, the technique as presented in [28], the architecture is virtually infeasible to implement with presently available memories.

Moreover, the conventional contour tracing algorithms used in [27,28] do not have the intelligent features present in [1] method and subsequently in our proposed implementation. These characteristics, required in many video processing applications, include detection and elimination of dead contour branches and noisy contours.

Some efforts have been dedicated to implement Connected Component Labeling, CCL, algorithms on hardware [29–33], however, no previous studies have been conducted on, to the authors' best knowledge, FPGA-based contour filling methods. The CCL methods assign a label to a pixel such that its adjacent and identical pixels have the same label. As such, the CCL algorithms can only label *filled* objects, which is a significant constraint in many video processing applications, specially in video surveillance [1]. In such applications, only the object contours (edges) are available due to the result of pre-processing algorithms, such as motion detection [1], and therefore, contour filling is required to uniquely label each objects.

A systolic architecture is proposed for CCL by Rasquinha et al. in [29], which uses N processing elements for MxN image. Crooks et al. [30] present an FPGA architecture for CCL, which requires scanning iteratively the input and intermediate images until no change in resulting image occurs. However, [30] achieves real-time performance *only* for images with simple objects, and therefore, fails to completely label real video objects in applications such as video surveillance. Another VLSI architecture, consisting of four processors, for CCL is presented by [31], and Jablonski et al. [32] present an implementation of Classical CCL in Handel-C language. A fast and parallel VLSI architecture for object labeling in binary images, using a 3x4 window, is presented by Shyue et al. in [33].

The primary rationality to select CCL over contour filling for hardware implementations in [29–33] would have been the fact that CCL can be exploited for parallelism, but contour filling methods are inherently sequential. Thus, implementing sequential contour filling algorithms are more challenging on parallel hardware devices such as FPGAs.

5.3 Overview of the Reference Contour Tracing and Filling Algorithm

5.3.1 Tracing Algorithm

The gaps free edge image, $E(n)$, produced with spatio-temporal motion detection followed by morphological edge detection [1] consists of object contours (white points, p_w) and a background (black points, p_b). The goal of a contour tracing algorithm is to link the white points, p_w, into a group. Unlike conventional contour tracing algorithms, [1] extracts contours of all closed complex objects while deleting dead or inner branches, and excluding contours of noisy objects. The detection and exclusion of such contours are important and necessary in video surveillance and other video processing applications, hence, we select [1] for our FPGA implementation. However, inherent sequential nature of [1] brings some challenges to its implementation on parallel hardware devices such as FPGAs. In addition, the process of excluding a contour in [1] requires manipulating previous frame contours. As such, an efficient method of retrieving appropriate contours of previous frame is needed. We propose an efficient cache architecture, in the FPGA, to overcome the sequential issue and a robust mechanism to access previous frame contours stored in a memory without interfering the core processing units.

A block diagram of the tracing method [1] is depicted in Fig. 5.1. More detailed description of this referenced algorithm can be found in [1]. As can be seen from Fig. 5.1, the algorithm can be partitioned into five sub modules, which are briefly outlined next.

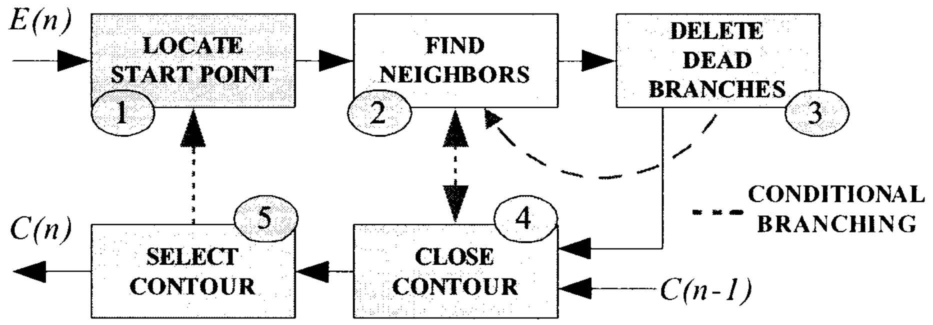

Figure 5.1: Contour Tracing Algorithm [1].

Locating A Start Point (Rule 1): The edge image, E(n), is scanned in raster mode (from left to right and from top to bottom) until an unvisited white points, p_w, which has at least one unvisited neighbor is found. If such a p_w exists, then set starting point, p_s, $= p_w$, set current point, p_c, $= p_s$, and perform Rule 2.

Finding the Rightmost Neighbor Points (Rule 2): The tracing technique in [1] is performed in anti-clockwise direction searching for a rightmost neighbor of the current point in an 8-neighborhood, p_i. The current searching direction, d_s, is defined by the direction from the previous point, p_p, to the current point. The direction to the rightmost neighbor of a current point, which depends on d_s, is formulated in Eq. 5.1. The algorithm searches up-to five and six rightmost neighbors for even and odd value of d_s, respectively. If a rightmost neighbor is present, then p_c is labeled visited, $p_p = p_c$, $p_c = p_i$, and Rule 4 is executed, otherwise p_c is a dead branch and deleted by performing Rule 3.

$$\{d_s + 6 + [(d_s + 1)\ mod\ 2]\}\ mod\ 8 \tag{5.1}$$

Deleting Dead Branches (Rule 3): Rule 3 eliminates current point p_c from $E(n)$, sets $p_c = p_p$ and p_p to its previous neighbor, and finally activates Rule 2.

Contour Closing (Rule 4): If $p_c = p_s$ or p_c is labeled visited, the current contour, C_c, being traced is closed. In both cases, Rule 4 removes all the points of the C_c from $E(n)$ and perform Rule 5. Furthermore, if p_c is labeled visited, remaining dead points of C_c are eliminated from $E(n)$. If C_c is not closed, then Rule 4 stores coordinate and the chain code of p_c and activates Rule 2.

Contour Selection (Rule 5): In this rule, C_c is verified with three measures before adding C_c to the contour list, $C(n)$. C_c is not added to $C(n)$ if 1) current contour length, P_c, is too small, or 2) P_c is small and has no corresponding contour in the previous contour list, $C(n-1)$, or 3) C_c resides in an already traced contour, C_p, causing a low spatial homogeneity of the object of C_p.

Although the method described in [1] records contours in both the chain code and the point coordinates, we use chain code in our implementation since the chain code requires less memory for storage.

5.3.2 Filling Algorithm

Contour filling [7] follows spatio-temporal motion detection and contour tracing [1]. Notice that, unlike conventional contour tracing algorithms, [1] extracts contours of all closed complex objects while deleting dead or inner branches, and excluding contours of noisy objects, which are important characteristics required by many video processing applications.

The reference filling algorithm [7] utilizes chain code information obtained during the tracing to fill every closed contours $C_i \in C(n)$, one by one, with a unique label. It analyzes the chain codes of current and next contour points (cc_i and cc_{i+1}, respectively) to determine whether the point right to the current contour point is a seed. When a valid seed is found, filling continues rightwards until reaching the next contour point in the same scan-line. A seed point is only selected for the following two cases:

(1) $cc_i = \{5\ or\ 6\ or\ 7\}$ and $(cc_{i+1} > cc_i\ mod\ 5)$, or

(2) $cc_i = \{0\ or\ 1\}$ and $cc_{i+1} = 7$.

Furthermore, internal contours are filled with zero, and on the completion of the filling process for all C_i, the result is a *labeled* gray-level image.

5.4 Proposed Architecture

Fig. 5.2 exemplifies our proposed system level architecture of contour tracing and filling. We have also integrated the proposed FPGA-based implementation of object detection and noise estimation presented in chapters 4 and 3 respectively, as a front-end processing engine for our proposed architecture. The proposed contour tracing and filling architecture consists of HIGH-SPEED CACHE, CONTOUR TRACING and CONTOUR FILLING modules. In addition, we have improved the robustness of our Direct Memory Access (DMA) module, presented in chapter 4.

5.4.1 Architecture of HIGH-SPEED CACHE

The performance of the any sequential contour tracing and filling architecture is heavily determined by the efficiency of the memory and its data transferring mechanism. The algorithm [1] demands reading a 3x3 window randomly from memory.

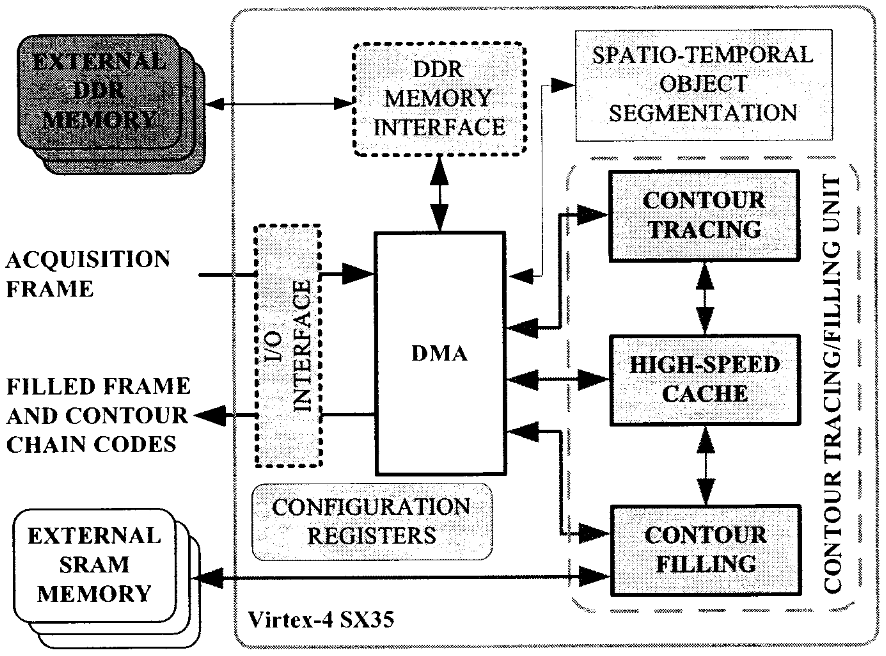

Figure 5.2: System-Level Architecture of Contour Tracing and Filling.

Intuitively, Static-RAM (SRAM) devices are ideal for random access applications, however, contour tracing requires reading a 3x3 window as fast as possible, ideally in one clock cycle. SRAM needs 3 clocks (1 clock/1 raw) and accessing 3 bits in a raw deflates the SRAM bandwidth, as the data bus of conventional SRAM is significantly greater than 3 bits. Thus, we propose a scalable, efficient and high speed cache architecture by exploiting FPGA memory blocks (BRAMs), which is depicted in Fig. 5.3. Main attributes of our cache are: 1) simultaneous read and write of 16 pixels in each direction, 2) 4.8 GBits/s aggregate throughput and 3) scalability with $O(n)$ area complexity.

In order to complete contour tracing and filling of one frame, cache is sequentially required to 1) store $E(n)$, 2) read START PIXEL ARRAY (SPA), 3) generate 3x3 WINDOW, 4) write reconstructed contour traced frame ($RT(n)$) and 5) read contour traced frame ($CT(n)$). Our proposed cache has four HIERARCHICAL MEMORY STACKS, HMS, which consists of four BRAMs constructed as dual port with 1 bit

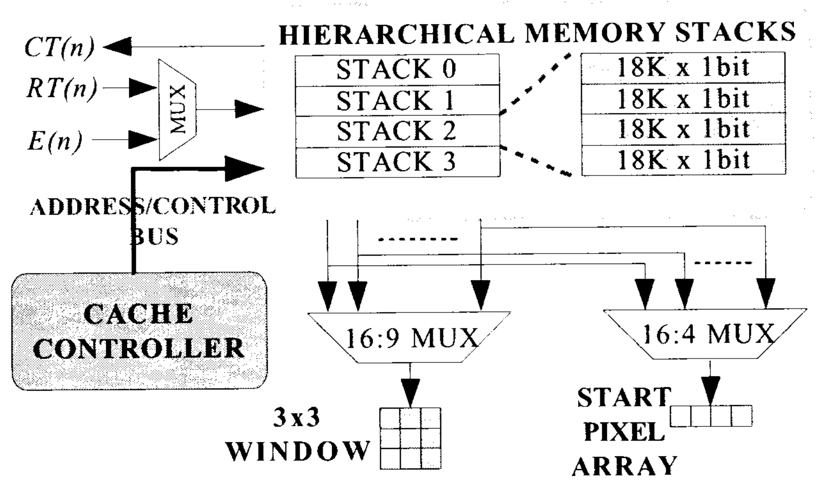

Figure 5.3: Scalable Architecture of the HIGH-SPEED CACHE.

wide and 18K deep. We write the first line of $E(n)$ in HMS0, the second in HMS1,.., the fifth in HMS0 and so on. In the same sequence, we store four pixels in the four BRAMs of each HMS. The main motivation for storing in such a sequence is that it facilitates reading any 16 pixels in one clock cycle. The CACHE CONTROLLER, CACTRL, schedules $E(n)$, $CT(n)$, $RT(n)$, 3x3 WINDOW, and SPA by managing all addressing and read/write controls to each HMS and controlling the three MUXs. As it is shown in the Sec. 5.4.5, the total scheduling pipeline, in clock cycles, is $< 7N$, where N is the total number of pixels in one frame. Thus, total access time required in the cache is less than 4.9 ms for a CIF frame, when the FPGA is running at 150MHz clock.

5.4.2 Architecture of CONTOUR TRACING Module

As illustrated in Fig. 5.4, most of the functionality of the proposed CONTOUR TRACING are controlling various contour tracing events. ADDR GEN perform rule 1 of [1], and generates direct cache addresses, DCA, based on the pixels in SPA and controls signal received from the two controllers, WINDOW CTRL and CHAIN CODE CTLR. WINDOW CTRL takes 3x3 WINDOW and determines if the tracing rules 2-5 are valid by means of a Finite State Machine (FSM) and some trivial logic employed to evaluate rule 5.

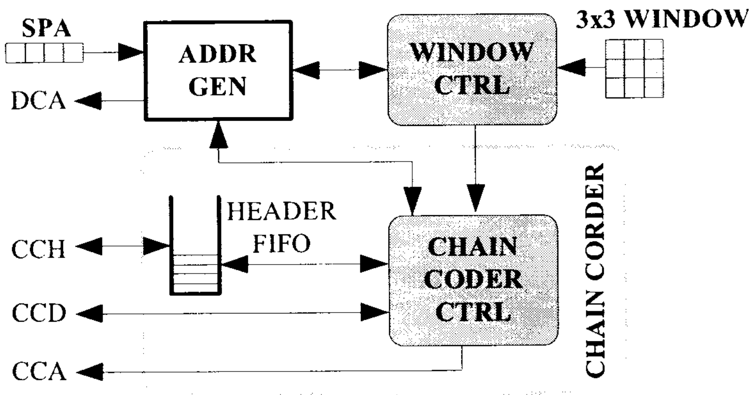

Figure 5.4: Overall Schematic of the CONTOUR TRACING Module.

As the the name implies, the CHAIN CODER produces Chain Code Streams (CCS) of the contours. We write CCS, while they are being produced, to the DDR memory. A CCS already written to the memory may not be valid if it is a dead branch or Rule-5 caused it to remove, therefor, such a CCS should be identified, and be excluded from the contour list. We adopted headers (CCH) and tails for CCS as well as for the Chain Code Frame (CCF) starting with 0x8 nibble as a marker followed by header descriptors. Moreover, we intentionally use 4 bits for chain codes which are from 0x0 to 0x7, therefore, header marker 0x8 can be easily distinguished. Fig. 5.5 defines a complete chain code bit stream. When a CCS belongs to a dead branch, CHAIN CODER sets a flag in the CCH. On completion, of contour tracing of a full frame, CHAIN CODER extracts the header descriptors to remove any CCS of dead branches, and reconstructs contour traced frames $RT(n)$ in the cache. $RT(n)$ is transfered to the DMA along with the chain codes as the final output.

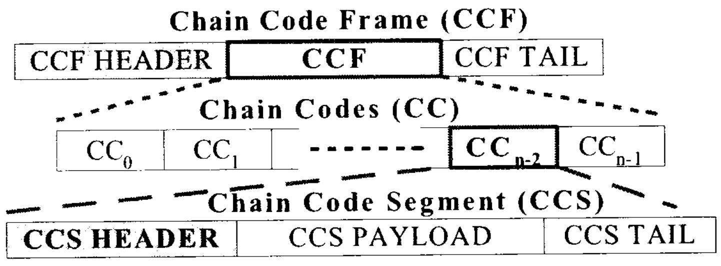

Figure 5.5: Contour Bit Stream Structure.

5.4.3 CONTOUR FILLING Architecture

Contour filling requires to fill each contours, one after another, with a unique label by traversing on its contour. For a given point on the contour, when a seed point is found, filling is started rightwards until reaching the next contour point in the same scan-line. A seed point is determined (see Sec. 5.3.2) based on the current and next chain codes, cc_i and cc_{i+1} respectively.

We propose a simplified architecture for the contour filling, which utilizes the HIGH-SPEED CACHE and a bank of external SRAM. First, we reconstruct each contour, one by one, in the HIGH SPEED CACHE and each contour is labeled in the external SRAM bank, which consists of a 16 bits data-bus and a 20 bits address-bus. Furthermore, the SRAM bank physically runs at 166 MHz. Notice that we selected SRAM over DDR DRAM, for the contour filling implementation, primarily, to avoid drastic bandwidth reduction due to the DRAM page mises. Hence, SRAM utilization improves the overall processing throughput of the system architecture.

Fig. 5.6 exemplifies the architecture of the labeling process. The circuitry requires to buffer chain codes in the CC FIFO to synchronize with the contour traced objects ($CT(n)$) which is streamed from the HIGH SPEED CACHE. PE-0 and PE-1 modules, comprising of a few comparators, take cc_i and cc_{i+1} and determine whether the point right to the current contour point being considered is a seed. This decision is passed to a controller, LABEL GEN CTRL (LGC), which controls write-data and write-

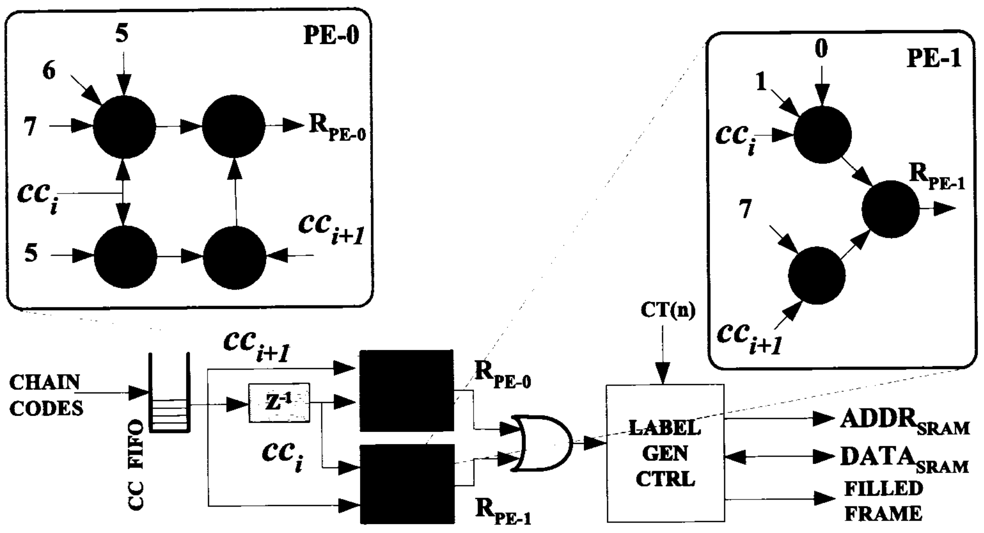

Figure 5.6: Architecture of the Contour Filling Module.

address to the SRAM. The write-data remains unchanged within a contour and is modified for each contour. The SRAM has 16 bits data-bus, thus it allows to label up-to 64K objects within a frame.

Moreover, LGC schedules all the access to SRAM in the correct order: 1) clear the memory, 2) write labeled objects and 3) read labeled frames. [7] labels contours within a contour, therefore LGC may require to modify a given location multiple times. However, we assume that in real video sequences, the maximum number of contours inside an outmost contour is two. Hence, in the worst case scenario, the number of write access to a given location is four, including clearing. This formulates the theoretical lowest processing limit of the labeling module to be 327 frames/s for CIF video resolution, which is greater than the overall throughput of the proposed contour tracing and filling architecture.

5.4.4 Improved DMA Architecture

An efficient management of data transfers within a system is the key to any real-time hardware implementation. In our implementation, we designed a scalable and versatile DMA architecture that can be easily configured by a simple set of registers. The proposed DMA consists of 4 KB deep First In First Out, FIFO, memories connected to each read and write DMA channels, a DMA controller (DMACTLR) to manage these FIFOs, and a DDR memory controller. A write transfer to the memory is initialized by filling the corresponding write FIFO (up-to a maximum of 2 KB), and sending a request to DMACTLR. Whenever a read FIFO is half empty, a read request is automatically initialized. An internal cache memory is used to store the addresses and transfer descriptions of each DMA channel. A round-robin arbitrator arbitrates all parallel requests from each channels and serves the selected DMA channel.

Furthermore, the DMA presented in chapter 4 constitutes the ability to access a memory location with an address provided by a processing unit, while paying special attention to minimize the performance overhead caused when a small amount of data is accessed from the memory. This feature facilitates updating the header of an already stored chain code by the CONTOUR TRACING.

5.4.5 Pipeline Scheduling

Fig. 5.7 shows the overall timing of the proposed architecture. Intuitively, the number of clock cycles required to complete contour tracing and filling for a full frame appears to be $8N$, where N is the total number of pixels in one frame. However, this is true only for the first frame in a video sequence, since reading the current filled frame from the SRAM can be performed while storing the next edge frame $E(n)$ in the cache, in parallel. Thus, the maximum pipeline delay of the proposed architecture is $7N$ clock cycles, or 4.9 ms (204 frames/s) for the CIF video resolution, based on a 150 MHz processing clock running on the FPGA.

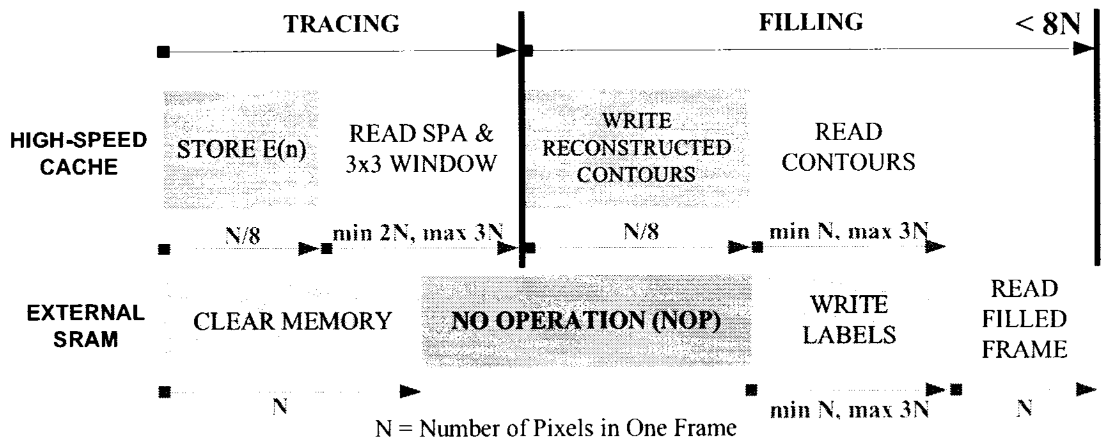

Figure 5.7: Overall Pipeline Timing of the Proposed Architecture.

5.5 Experimental Results

5.5.1 Verification

We verify the integrity of the proposed design, by simulating the *Hall* video sequence, which consists if 300 frames of 352 pixels x 288 lines. The edge frames produced with the result of contour tracing of FPGA simulation and the reference C software implementation for the 54th captured frame in the *Hall* video sequence is shown in Fig. 5.8.

To verify the result of our proposed FPGA implementation objectively with the software implementation, we used the Product of Correctly Classified Proportions [2], PCP, measure which is widely known and used as an objective measure for evaluating images. Serving software contour-filled frames $CF_{sw}(n)$ as the ground-truth data, Fig. 5.9 (a) shows that the PCP is close to 1 and always above 0.9 for the 300 frames of *Hall* video sequence. Notice that when a binary image is identical to the ground-truth frame, then PCP is 1. Thus, contour-filled frames produced by FPGA, $CF_{hw}(n)$, are very close, if not identical, to the $CF_{sw}(n)$.

Moreover, we enumerated the sum of the white pixels, Δ_{hw}, in the absolutely difference frames, $|CF_{hw}(n) - CF_{sw}(n)|$. Fig. 5.9 (b) illustrates that $\Delta_{hw} \geq 23$ for

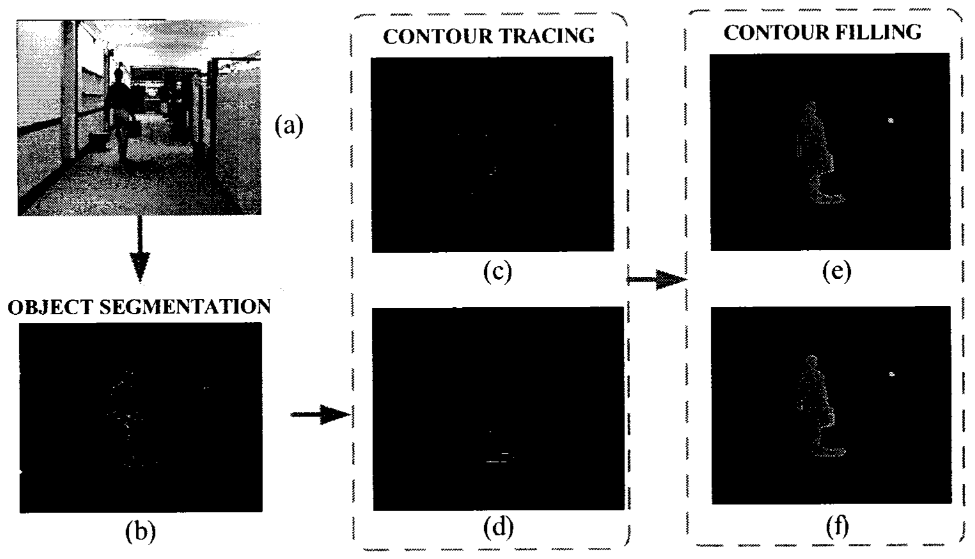

Figure 5.8: Subjective comparison between the FPGA and software implementation results - (a) 54th frame in the captured video sequence. (b) Spatio-temporal object segmentation of [1], (c) Contour tracing results with C, and with the proposed FPGA implementation (d), (e) Contour filling with C, and with the proposed FPGA implementation (f).

Hall video sequence.

Furthermore, we have successfully verified the actual functionality, accuracy and performance of the proposed FPGA implementation on a frame grabber.

5.5.2 Synthesis and FPGA Implementation

We have coded and simulated the proposed architecture in VHDL, synthesized and implemented to a Xilinx Virtex-4 SX35 FPGA. The implemented architecture occupies 9% of registers, 7% of LUTs (look up tables), 12 % of BRAMs, and 1% of multipliers of the FPGA. Our proposed-constrained implementation achived a clock rate of 156 MHz, and consumes 1.25 W for a toggle-rate of 50%. On the actual hardware platform, we set the processing clock in the FPGA to 150 MHz, which enabled the FPGA to capture and complete contour tracing and filling at 204 frames/s (in 4.9 ms) for the CIF video resolution.

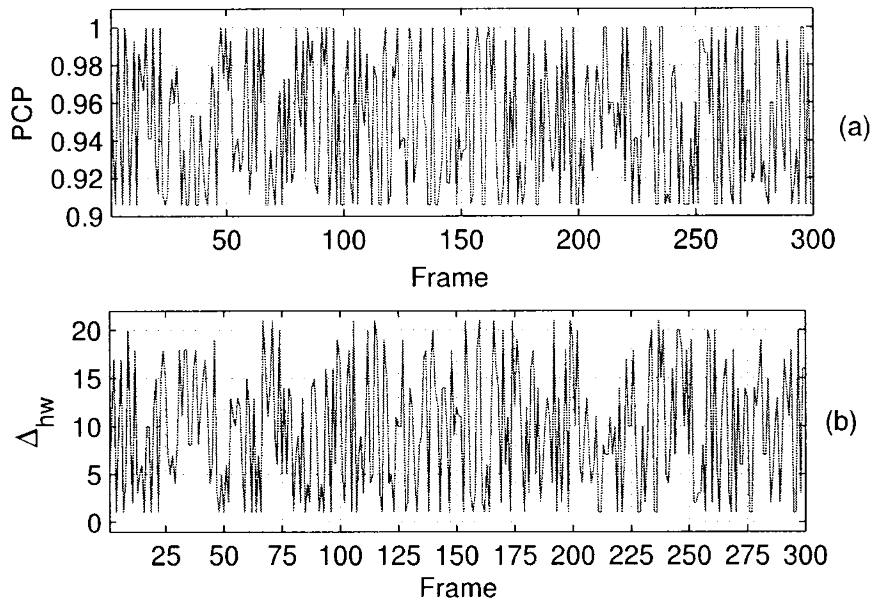

Figure 5.9: (a) Comparison between software and hardware implementations with PCP objective measure [2], and (b) difference of total pixels between software and hardware implementations Δ_{hw}.

5.5.3 Comparison to the Existing Methods

The full custom VLSI CMOS design for extracting contours presented by Agi et al. in [27] fails to trace complete contours of large objects and requires an external post processor to link partially completed contours. Having an external post processor in addition to the core contour tracing circuitry increases cost, power, and physical area.

The hardware architecture presented in [28] is fundamentally infeasible to implement due to its requirements of random and simultaneous memory access. Currently available memories do not possess greater than two read/write ports, but [28] requires a minimum of $N(>> 2)$ ports to read simultaneously N raws of a frame. Furthermore, additional simultaneous, random and fast accesses to the memory are required in [28] when each raw produces partially completed chain codes, and these are read and linked to create the final chain codes. As a result of this heavy memory access requirement, [28] needs an enormous amount of pins. which are not currently available even on the largest FPGA.

The architectural requirements needed in [27,28] prevail having realistic hardware acceleration methods for contour tracing. In contrast, we exploited heterogeneous resources readily available on FPGA devices, adopted them in our architecture and devised a real hardware solution. Both [27, 28] need N contour processing units, whereas our method consists of only one tracing unit. Moreover, [27, 28] lack key contour tracing features such as deleting dead branches and removing noisy contours, and therefore, [27,28] are not suitable for video applications such as video surveillance.

Although some efforts [29–33] have been dedicated to implement CCL algorithms on hardware, to the authors' best knowledge, the work presented in this thesis is the first study carried out in implementing contour filling methods on FPGAs. Notice that CCL algorithms requires *filled* objects as input, imposing a significant constraint in many video processing applications, specially in video surveillance [1]. Moreover, CCL can be exploited for parallelism, but contour filling methods are inherently sequential. Thus, implementing sequential contour filling algorithms are more challenging on parallel hardware devices such as FPGAs.

Moreover, for CIF video resolution, the reference contour tracing and filling algorithms [1, 7, 8] require 475 ms, on an average, when implemented in C++ on a single processor, Linux based PC, with Intel P4@2.4GHz and 768MB of memory. In contrast, our proposed FPGA implementation executes in less than 4.9 ms, accomplishing approximately an order of magnitude performance improvement over the software implementation.

5.6 Summary

In this chapter, we proposed a robust real-time, scalable and compact FPGA-based architecture and its implementation of contour tracing and filling of video objects. Intentional use of heterogeneous resources in FPGAs, and advanced design techniques such as heavy pipelining and data parallelism enabled us to achieve an impressive throughput of 204 frames/s, while consuming minimal power and resources. We verified our proposed implementation on an actual Virtex-4 SX35 FPGA platform for its functionality, accuracy and real-time performance. We showed that, compared to the existing hardware-based methods, our proposed solution is much more feasible, cost effective, and possesses key features such as deleting dead contour branches and exclusion of noisy contours, which are required in many video processing applications. Furthermore, when compared to a pure software-based implementation, we obtained a speed-up factor of ten-fold with the proposed architecture.

Chapter 6

Conclusion and Future Work

The core focus of this thesis, has been the acceleration of video object segmentation algorithms on reconfigurable FPGAs.

In this thesis, we proposed an efficient implementation based on *one* FPGA, which integrated four video processing methods : sorting, noise estimation, object detection, and contour analysis. Our proposed architecture aimed at segmenting moving objects in video signals, which consists of, noise estimation, object detection (i.e., separation of objects and background), and contour analysis.

We, first, proposed an FPGA-based architecture for stably sorting a large volume of integer and fractional keys in real-time for video noise estimation. The proposed architecture is scalable to gain higher throughput by trading off only the FPGA internal memory resources. Sorting performance comparisons showed that the proposed implementation is significantly better than the existing methods.

The thesis proposed an improved FPGA-based implementation of a noise estimation method, which achieved significant performance and area improvements over an existing method.

A novel, scalable and compact FPGA architecture for real-time spatio-temporal object detection was proposed followed by an architecture and its implementation of contour tracing and filling of video objects. These architectures were integrated with the noise estimation implementation on one FPGA device.

There were numerous outfalls encountered during the course of the proposed FPGA implementation. The intricate sorting needed in the video noise estimation method was unprovable for an implementation on resources-constrained hardware such as an FPGA. Consequently, we proposed a modified counting sort algorithm, which is well suited for hardware implementation for sorting large integer or fractional data. Moreover, video processing algorithms, which can be well explored for parallelism (e.g., connected component labeling), are generally implemented on FPGAs. As such, contour tracing and filling methods, which are inherently sequential are more challenging to architect on parallel hardware devices, as these methods demand fast, random, and heavy data movements between memory and processing units. In this thesis, we proposed a novel implementation of these sequential chaotic contour tracing and filling algorithms by efficiently exploiting FPGA resources. We proposed an efficient architecture for a pixel cache, which is centric to the efficiency of the proposed contour tracing and filling implementations. Main attributes of proposed cache were: 1) simultaneous read and write of 16 pixels in each direction, 2) 4.8 GBits/s aggregate throughput and 3) scalability with $O(n)$ area complexity. In addition, an efficient management of data transfers of the proposed architecture was unenviable. These data transfers include simultaneous read/write access to memory (e.g., background subtraction), input and output video signal transfers, synchronizing frames, e.t.c. In our implementation, we designed a scalable and versatile DMA architecture, configured by a simple set of registers, that handled the necessary data movements efficiently among the sources and sinks without degrading the performance of any core processing elements.

Intentional use of heterogeneous resources in FPGAs, and advanced design techniques such as heavy pipelining and data parallelism enabled us to achieve an impressive performance, while consuming minimal power and resources. We verified our proposed implementations on an actual Virtex-4 SX35 FPGA platform for its functionality, accuracy and real-time performance.

We showed that, compared to the existing hardware-based methods, our proposed solutions are much more feasible, cost effective, and possess key algorithmic features, which are inherently required in many video processing applications. Furthermore, when compared to pure software-based implementations, we obtained orders of magnitude performance improvements with each of the proposed architectures.

Thresholding affects severely on the efficiency of the video object segmentation, thus a possible future work to the proposed architecture includes implementing a better but more complex thresholding method such as the Eular method to calculate the threshold value. Another extension would be to implement and integrate other post video processing algorithms such as video object analysis and tracking. Also, the proposed implementation can be extended to calculate the object features, such as perimeter, centre of gravity, area, color or shape, by analyzing the results of the contour tracing and filling.

Bibliography

[1] A. Amer, "Memory-based spatio-temporal real-time object segmentation," *in Proc. SPIE Int. Symposium on Electronic Imaging, Conf. on Real-Time Imaging (RTI)*, vol. 5012, pp. 10–21, Jan. 2003.

[2] P. L. Rosin, "Thresholding for change detection," *Computer Vision and Image Understanding*, vol. 86, pp. 79–95, 2002.

[3] H.H. Seward, "Information sorting in the application of electronic digital computers to business operations," M.S. thesis, Massachusetts Institute of Technology (MIT), 1954.

[4] C. Y. Huang, G. J. Yu, and B. D. Liu, "A hardware design approach for merge-sorting network," *Proceedings of the 2001 IEEE International Conference on Circuits and Systems*, vol. 4, pp. 534–537, May 2001.

[5] F-X. Lapalme, A. Amer, and C. Wang, "FPGA architecture for real-time video noise estimation," *in Proc. IEEE Int. Conference on Image Processing (ICIP)*, pp. 3265–3260, Oct. 2006.

[6] T. Pavlidis, "Contour filling in raster graphics," *in Proc. ACM Annual Conference on Computer Graphics and Interactive Techniques*, pp. 29–36, Aug. 1980.

[7] F. Achkar, "Hysteresis-based selective gaussian-mixture model for real-time background update and object detection," M.S. thesis, Concordia University, Montreal, Quebec, Canada, Nov. 2006.

[8] A. Amer, *Object and Event Extraction for Video Processing and Representation in On-Line Video Applications*, Ph.D. thesis, INRS-Telecommunications (Institut national de la recherche scientique), Montreal, Dec. 2001.

[9] F. Chang, C.J. Chen, and C.J. Lu, "A linear-time component-labeling algorithm using contour tracing technique," *Computer Vision and Image Understanding*, vol. 93, pp. 206–220, Feb. 2004.

[10] F. Chang and C.J. Chen, "A component-labeling algorithm using contour tracing technique," *in Proc. IEEE International Conference on Document Analysis and Recognition*, pp. 741–745, 2003.

[11] C. T. Huitzil and M. A. Estrada, "Real-time image processing with a compact FPGA-based systolic architecture," *Elsevier Journal of Real-time Imaging*, vol. 10, pp. 177–187, 2004.

[12] D. S. Zhang and G. Lu, "Segmentation of moving objects in image sequence: A review," *Springer Circuits, Systems and Signal Processing (Special Issue on Multimedia Communication Services)*, vol. 20, pp. 143–183, 2001.

[13] A. Amer and E. Dubois, "Fast and reliable structure-oriented video noise estimation," *in Proc. IEEE Transactions on Circuits and Systems for Video Technology*, vol. 15, pp. 113–118, Jan. 2005.

[14] A.A. Colavita, A. Cicuttin, F. Frantik, and G. Capello, "SORTCHIP: A VLSI implementation of a hardware algorithm for continuous data sorting," *IEEE Journal of Solid State Circuits*, vol. 38, pp. 1076–1079, Jun. 2003.

[15] B. van Ginneken, B.M. ter Haar Romeny, and M.A. Viegever, "Computer-aided diagnosis in chest radiography: A survey," *IEEE Transcations on Medical Imaging*, vol. 20, no. 12, pp. 1228–1241, Dec. 2001.

[16] L. C. Sanz and D. Petkovic, "Machine vision algorithms for automated inspection of thin-film disk heads," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 10, no. 6, pp. 830–848, 1988.

[17] D.H. Ballard, *Computer Vision*, Prentice-Hall, Englewood, New Jesey, 1982.

[18] M. Bednara, O. Beyer, J. Teich, and R. Wanka, "Tradeoff analysis and architecture design of a hybrid hardware/software sorter," *IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pp. 299–309, Jul. 2000.

[19] R. Maheshwari, S.S.S.P. Rao, and P.G. Poonacha, "FPGA implementation of median filter," *Proceedings on VLSI Design*, pp. 523–524, Jan. 1997.

[20] K. Benkrid, D. Crookes, and A. Benkrid, "Design and implementation of a novel algorithm for general purpose median filtering on FPGAs," *International Symposium on Circuits and Systems*, vol. 4, pp. 425–428, May 2002.

[21] M.S. Hamid and S. Marshal, "FPGA realisation of the genetic algorithm for the design of gray-scale soft morphological filters," *International Conference on Visual Information Engineering*, pp. 141–144, Jul. 2003.

[22] S.A. Fahmy, P.Y.K. Cheung, and W. Luk, "Novel FPGA-based implementation of median and weighted median filters for image processing," *IEEE International Conference on Field Programmable Logic and Applications*, pp. 142–147, Aug. 2005.

[23] D. Demigny, L. Kessaland, R. Bourguiba, and N. Boudouani, "How to use high speed reconfigurable FPGA for real time image processing," *in Proc. IEEE Conference on Computer Architecture for Machine Perception*, pp. 240–246, 2000.

[24] N.K. Ratha, A.K. Jain, and D.T. Rover, "FPGA-based high performance page layout segmentation," *Proceedings of the 1996 Great Lakes Symposium on VLSI*, pp. 29–34, Mar. 1996.

[25] T. Nakano, T. Morie, and A. Iwata, "A face/object recognition system using FPGA implementation of coarse region segmentation," *Annual Conference of the Society of Instrument and Control Engineers (SICE)*, vol. 2, pp. 1552–1557, 2003.

[26] P. McCurry, F. Morgan, and L. Kilmartin, "Xilinx FPGA implementation of an image classifier for object detection applications," *International Conference on Image Processing*, vol. 3, pp. 346–349, 2001.

[27] I. Agi, P. J. Hurst, and A. K. Jain, "A VLSI processor for parallel contour tracing," *in Proc. IEEE Transactions on Signal Processing*, vol. 40, no. 2, pp. 429–438, Feb. 1992.

[28] T. L. Chia, K. B. Wang, L..R. Chen, and Z. Chen, "A parallel algorithm for generating chain code of objects in binary images," *Information Sciences Informatics and Computer Science*, vol. 149, no. 4, pp. 219–234, Feb. 2003.

[29] A. Rasquinha and N. Ranganathan, "C^3L : A chip for connected component labeling," *Tenth International Conference on VLSI Design*, pp. 446–450, Jan. 1997.

[30] D. Crookes and K. Benkrid, "FPGA implementation of image component labeling," *Proceedings of SPIE*, vol. 3844, pp. 17–23, Aug. 1999.

[31] Z. Chen K.B. Wang, T.L. Chia and D.C. Lou, "Parallel execution of a connected component labeling operation on a linear array architechture," *Euromicro Symposium on Digital System Design*, pp. 387–393, Sep. 2004.

[32] M. Jablonski and M. Gorgon, "Handel-c implementation of classical component labelling algorithm," *Euromicro Symposium on Digital System Design*, pp. 387–393, Sep. 2004.

[33] S.W. Yang et al., "VLSI architecture design for a fast parallel label assignment in binary image," *IEEE International Symposium on Circuits and Systems*, vol. 3, pp. 2393–2396, May 2005.

[34] Xilinx Inc., *Virtex-5 Family Overview - LX , LXT, and SXT Platforms*, On line, http://direct.xilinx.com/bvdocs/publications/ds100.pdf, 2006.

[35] Xilinx Inc., *Xilinx Chipscope Pro*, On line, http://www.xilinx.com/ise/optionalprod/cspro.htm, 2006.

[36] K. Ratnayake and A. Amer, "An FPGA-based implementation of spatio-temporal object segmentation," *in Proc. IEEE Int. Conference on Image Processing (ICIP)*, pp. 3265–3268, Oct. 2006.

[37] K. Ratnayake and A. Amer, "An FPGA-based architecture of stable-sorting on a large data volume: Application to video signals," *in Proc. 41st IEEE Conference on Information Sciences and Systems*, (Accepted) Mar. 2007.

[38] K. Ratnayake and A. Amer, "Sequential, irregular and complex object contour tracing on FPGA," *in Proc. IEEE Int. Conference on Image Processing (ICIP)*, (Accepted) Sep. 2007.

[39] K. Ratnayake and A. Amer, "A Real-Time Implementation of Chaotic Contour Tracing and Filling of Video Objects on Reconfigurable Hardware," *in Proc. IEEE International Conference on Systems, Man, and Cybernetics*, (Accepted) Oct. 2007.

[40] F. Dufaux and J. Konrad, "Efficient, robust, and fast global motion estimation for videocoding," *IEEE Transactions on Image Processing*, vol. 9, no. 3, pp. 497–501, Mar. 2000.

[41] H. Kim, B-G. Nam, J-H Sohn, and H-J Yoo, "A 231MHz, 2.18mW 32-bit logarithmic arithmetic unit for fixed-point 3D graphics system," *in Proc. IEEE Asian Solid-State Circuits Conference(A-SSCC)*, pp. 305–308, Nov. 2005.

[42] C. Rambabu, I. Chakrabarti, and A. Mahanta, "An efficient architecture for an improved watershed algorithm and its FPGA implementation," *In Proc. IEEE International Conference on Field-Programmable Technology*, pp. 370–373, Dec. 2002.

[43] K. V. Asari, T. Srikanthan, S. Kumardemi, and D. Radhakrishnan, "A pipelined architecture for image segmentation by adaptive progressive thresholding," *Journal of Microprocessors and Microsystems*, vol. 23, pp. 493–499, 1999.

[44] H. Tsuji, S. Saito, H. Takahashi, and M. Nakajima, "Estimating object contours from binary edge images," *in Proc. IEEE Int. Conference on Image Processing (ICIP)*, vol. 3, pp. 453–456, Sep. 2005.