Image Processing Architectures for Binary Morphology and Labeling

Hugo Hedberg

Lund 2008

The Department of Electrical and Information Technology
Lund University
Box 118, S-221 00 LUND
SWEDEN

This thesis is set in Computer Modern 10pt,
with the LaTeX Documentation System
using Pontus Åströms thesis template.

Series of licentiate and doctoral thesis
No. 2
ISSN 1654-790X

Abstract

Conventional surveillance systems are omnipresent and most are still based on analog techniques. Migrating to the digital domain grants access to the world of digital image processing enabling automation of such systems, which means extracting information from the image stream without human interaction. The resolution, frame rates, and functionality in these systems are continuously increasing alongside the number of video streams to be processed. The sum of all these parameters imposes high data rates and memory bandwidths which are impossible to handle in pure software solutions. Therefore, accelerating key operations and complex repetitive calculations in dedicated hardware architectures is crucial to sustain real-time performance in future advanced high resolution and frame rate systems.

To achieve this goal, this thesis presents four architectures of hardware accelerators to be used in real-time embedded image processing systems, implemented as an FPGA or ASIC. Two morphological architectures performing binary erosion or dilation, with low complexity and low memory requirement, have been developed. One supports static, and the other locally adaptive flat rectangular structuring elements of arbitrary size. Furthermore, a high-throughput architecture calculating the distance transform has also been developed. This architecture supports either the city-block or chessboard distance metric and is based on adding the result of parallel erosions. The fourth architecture performs connected component labeling based on contour tracing and supports feature extraction. A modified version of the morphological architecture supporting static structuring elements, as well as the labeling architecture, has been successfully integrated into a prototype of an automated digital surveillance system for which implementation aspects are presented. The system has been implemented and is running on an FPGA based development board using a CMOS sensor for image acquisition. The prototype currently has segmentation, filtering, and labeling accelerated in hardware, and additional image processing performed in software running on an embedded processor.

In loving memory of my father

Contents

Acknowledgment

I would like to express my gratitude to all the people who have inspired me and contributed to the writing of this thesis. What a journey this has been! From nightswimming in the crystal-clear waters of lake Geneva, to strolling the beaches of sunny California, and experiencing the beautiful Fiji islands, I feel privileged to have been given the possibility of gathering an amazing collection of memories from all the traveling, the discussions, and hard work, that has taken place during these years.

First of all, I would like to express my deepest gratitude to my supervisor associate professor PhD Viktor Öwall. Your guidance and helping hand throughout my years at the department made it all possible. Not to forget, thank you for all the constructive criticism regarding this thesis and for keeping calm when receiving messages during late nights and weekends.

The second person I owe a great deal of gratitude to is PhD Fredrik Kristensen. Thank you for providing valuable input to my work, for being a constant support, and a good friend. You even spent some of your spare time reading parts of this thesis though you left the group some time ago.

I would like to thank PhD Petr Dokladal at the Center of Mathematical Morphology in France for your guidance and support. My research visit to you in Fontainebleau during fall'06 was one of the most exiting experiances during my entire PhD studies.

My gratitude goes to all my colleagues in the whole Electroscience department, nowadays the department of Electrical and an Information Technology; you have all contributed to my work in various ways. I am especially grateful to the past and present members of the DASIC group: Thomas Lenart, Henrik Svensson, and Joachim Neves Rodrigues, for reading parts of this thesis, Johan Lövgren, Mattias Kamuf, Deepak Dasalukunte, and Jiang Hongtu, for all the interesting discussions about all and nothing. Thank you all for your profound knowledge in our field of research, and for being good friends, taking care of more socially related events, which made my time at the department a pleasant journey. I am also very grateful for the work of Elsbieta Szybicka, Erik Jonsson, Pia Bruhn, and Lars Hedenstjerna, who have been taking care of administrative, practical and computer related issues for me during these years.

I would like to thank the other persons involved in this research project, especially associate professor PhD Peter Nilsson. I would also like to thank Anders Olsson, and Daniel Elvin from AXIS Communications AB for their valuable input and professional perspective, setting the pace in this research project.

This work has been financed by the Competence Center for Circuit Design (CCCD) and sponsored by the Xilinx University Program by donating development platforms and software.

To my family, Christina, Björn, and Jonatan, thank you for all the joy, help, and support you have given me so far along the way. Christina, thank you for helping me to straighten out some of the tricky formulations in this thesis.

Last but certainly not least, I would like to say a few words to my love, Tove. From the bottom of my heart, thank you for being who you are, constantly re-energizing my batteries, and complementing the perspective of developing hardware architectures with other things in life. So, let us begin a new journey...

March, Lund, 2008

Hugo Hedberg

Preface

This thesis summarizes the author's academic work in the digital ASIC group at the Department of Electrical and Information Technology for the PhD degree in circuit design. The main contributions to the thesis are derived from the following journal publications:

> H. Hedberg, P. Dokladal, and V. Öwall, "Binary Morphology with Locally Adaptive Structuring Elements: Algorithm and Architecture," first round of revision for publication in *IEEE Transactions on Image Processing.*
>
> H. Hedberg, F. Kristensen, and V. Öwall, "Low Complexity Binary Morphology Architectures with Flat Rectangular Structure Elements," accepted for publication in *IEEE Transactions on Circuits and Systems I.*
>
> F. Kristensen, H. Hedberg, H. Jiang, P. Nilsson, and V. Öwall, "An Embedded Real-Time Surveillance System: Implementation and Evaluation," accepted for publication in *Journal of VLSI Signal Processing Systems*, Springer.

And the following peer-reviewed international conference contributions:

> H. Hedberg, and V. Öwall, "An Architecture for Calculation of the Distance Transform Based on Mathematical Morphology," to be submitted for publication, 2008.
>
> H. Hedberg, F. Kristensen, and V. Öwall, "Implementation of Labeling Algorithm Based on Contour Tracing with Feature Extraction," in *Proc. of IEEE International Symposium on Circuits and Systems (ISCAS'07)*, New Orleans, USA, May 2007.
>
> F. Kristensen, H. Hedberg, H. Jiang, P. Nilsson, and V. Öwall, "Hardware Aspects of a Real-Time Surveillance System," in *Proc. of IEEE International Workshop on Visual Surveillance* held at *European Conference on Computer Vision (ECCV'06)*, Graz, Austria, May 2006.
>
> H. Hedberg, F. Kristensen, P. Nilsson, and V. Öwall, "A Low Complexity Architecture for Binary Image Erosion and Dilation using Structuring Element Decomposition," in *Proc. of IEEE International Symposium on Circuits and Systems (ISCAS'05)*, Kobe, Japan, May 2005.

The papers "*Hardware Aspects of a Real-time Surveillance System*" and "*An Embedded Real-Time Surveillance System: Implementation and Evaluation*" are common efforts, the first focusing on the outline of the research project and the second on the details of a prototype of a complete automated digital surveillance system. The author's main contributions in both these publications are the parts addressing morphology and labeling. However, the author has also been involved in the integration of the system and, therefore, system level results are included in the thesis.

The following papers concerning education are also published but not considered part of this thesis:

H. Hedberg, J. N. Rodrigues, F. Kristensen, H. Svensson, M. Kamuf, and Viktor Öwall, "Teaching Digital ASIC Design to Students with Heterogeneous Previous Knowledge," in *Proc. of Microelectronic Systems Education, MSE'05*, pp. 15–16, Anaheim, California, USA, June 12-13, 2005.

J. N. Rodrigues, M. Kamuf, H. Hedberg, and Viktor Öwall, "A Manual on ASIC Front to Back end Design Flow," in *Proc. of Microelectronic Systems Education, MSE'05*, pp. 75–76, Anaheim, California, USA, June 12-13, 2005.

H. Hedberg, T. Lenart, and H. Svensson, "A Complete MP3 Decoder on a Chip," in *Proc. of Microelectronic Systems Education, MSE'05*, pp. 103–104, Anaheim, California, USA, June 12-13, 2005.

H. Hedberg, T. Lenart, H. Svensson, P. Nilsson and V. Öwall, "Teaching Digital HW-Design by Implementing a Complete MP3 Decoder," in *Proc. of Microelectronic Systems Education, MSE'03*, pp. 31–32, Anaheim, California, USA, June 1-2, 2003.

List of Acronyms

ASIC	Application-Specific Integrated Circuit
ADC	Analog-to-Digital Converter
BW	Bandwidth
CC	Clock Cycle
CCD	Charge-Coupled Device
CCTV	Closed Circuit Television
CoG	Center of Gravity
CMOS	Complementary Metal Oxide Semiconductor
DDR	Double Data Rate
EDA	Electronic Design Automation
DSP	Digital Signal Processor
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
fps	frames per second
FSM	Finite State Machine
GMM	Gaussian Mixture background Model
GPP	General Purpose Processor
HDL	Hardware Description Language
HW	Hardware
LUT	Lookup Table
MAC	Multiply-Accumulate
MM	Mathematical Morphology
PCB	Printed Circuit Board

PPC	PowerPC
P&R	Place and Route
RAM	Random-Access Memory
SE	Structuring Element
SDRAM	Synchronous Dynamic Random Access Memory
SW	Software
VGA	Video Graphics Array
VHDL	Very high-speed integrated circuit Hardware Description Language
WL	Word-Length
WLAN	Wireless Local Area Network

List of Definitions and Mathematical Operators

All definitions listed here are taken from [1] and [2].

$\mathbb{Z}$	1-D integer space
$\mathbb{Z}^+$	Positive integer space
$\mathbb{Z}^2$	2-D integer space
$\mathbb{R}$	1-D real (continuous) space
$\lfloor i \rfloor$	Floor function, rounds i to nearest lower integer towards minus infinity
$\lceil i \rceil$	Ceiling function, rounds i to the nearest upper integer towards infinity
$\exists x$	There exists an x such that...
$\forall x$	For all x,...
$x \in A$	The element x belongs to the set A
$x \notin A$	The element x does not belong to the set A
B	Structuring element
$\ominus$	Erosion
$\oplus$	Dilation
$\circ$	Opening
$\bullet$	Closing
$\emptyset$	Empty set
$A' = A^c$	Complementation or inverse
$\hat{A}$	Reflection, i.e. geometric inverse
$(A)_z$	The translation of A by z
$A \subseteq B$	A is a subset of B
$A \cap B$	Intersection of set A and B
$A \cup B$	Union of set A and B

Chapter 1

Introduction

Ever since the introduction of television, real-time monitoring has been a growing market. Adding a video recorder opened up a new world for the security industry. Video surveillance soon made its way into the court rooms and became convicting evidence. Today, video surveillance systems are omnipresent and part of everyday life and can be found in department stores, banks, bus terminals, etc. They are not only used for crime prevention purposes but also play their role in more social and industry related applications, e.g. traffic monitoring, processing monitoring, and customer statistics. With continuously increasing fields of application and integration into our lives, at least two serious questions arise: *What about personal integrity*? *How will this information be used*? Naturally, none of these questions will be answered in this thesis, but the author is aware of the possible link between the presented technology and social interests that are attached to this field of research. The only conclusion that can be drawn is that some of these applications are easier to accept than others but the boundary for acceptance is, of course, subjective. Without knowing what the future of automated surveillance will bring, as technology advances, the applicability of such systems is continuously increasing and is nowadays part of everyday life.

Conventional real-time surveillance systems are known as Closed Circuit Television (CCTV) systems. A typical system is traditionally controlled by a human operator, and supports multiple cameras, event recording, Pan-Tilt-Zoom (PTZ), auto-focusing etc. Automating such a system would not only reduce the time spent on monitoring the system itself, but can also increase the number of attached cameras in the system, thus increasing surveillance efficiency. Stepping into the digital domain and applying digital image processing is a natural evolution, since it enables the possibility to extract information from the image stream without human interaction. The extracted information can be used as decision support in an effort to reduce the number of errors or false alarms caused by human operators. For those concerned with the personal integrity in such automated systems, some argue that this will even increase, since the image processing gives the possibility of hiding the identities of the objects present in the scene, e.g. by blurring faces or even replacing the complete object with a synthetic model.

However, the ever increasing demands on higher resolutions and functionality within the automated systems impose a high bandwidth requirement that is not possible to handle in software running only on a general purpose processor which attempts to achieve real-time performance. Therefore, a design challenge in any automated surveillance system is to handle or reduce the bandwidth. In this thesis, to be able to address the high bandwidth, the system is implemented as an embedded system in which the main idea is to have key operations and repetitive calculations placed in dedicated hardware accelerators which efficiently reduces the amount of data needed to be processed in the software. This is identified as crucial to sustain real-time performance and simultaneously have a high resolution and frame rate. Therefore, the need for such dedicated hardware accelerators that can be deployed in an embedded system environment becomes evident.

1.1 Research Project

The research presented in this thesis is part of the development and implementation of a complete automated surveillance system based on a single self contained network camera in hardware. The aim of the system is to be able to detect, track and classify objects in consecutive frames. Such a system not only competes in terms of a higher frame rate and higher system resolution compared to other general processor solutions, but also with a reduced power dissipation due to higher hardware resource utilization. Integrating the single camera system with more cameras, or sound, would certainly increase the accuracy of the system but is beyond the scope of the project. Furthermore, although the system resolution, i.e. 320×240, is low compared to commercial products, this resolution is sufficient for development purposes, and is therefore often used when developing the hardware accelerators.

Three PhD students have been involved in developing different parts of the system. The author of this thesis is responsible for morphology and labeling, J. Hongtu for implementation of the sensor interface and segmentation [3], and F. Kristensen for feature extraction and tracking Software (SW). He is also responsible for implementation of additional Hardware (HW) units, e.g. the

PowerPC (PPC) interface. Furthermore, F. Kristensen also performed an investigation of the impact different color spaces have on shadows [4]. All three are involved in developing the system architecture and integration. The work is carried out in close collaboration with Axis Communication AB [5] thought the Competence Center for Circuit Design (CCCD).

1.2 Main Contribution and Thesis Statement

The main contribution of this thesis is to present four hardware architectures together with their corresponding implementation results. The architectures are to be used as hardware accelerators in embedded image processing systems, and are compiled in the following list:

- Two low complexity morphological architectures performing binary erosion or dilation with flat rectangular structuring elements, where one of them supports locally adaptive structuring elements,
- an architecture calculating the city-block and chessboard distance transform on binary images, and
- an architecture for connected component labeling based on contour tracing.

A general overview of a complete embedded automated surveillance system will be presented, outlining and setting the goals for this research project. Additional implementation aspects of a prototype of such a system will also be presented, but is not considered to be a part of this thesis' main contribution. Furthermore, incorporating two of the hardware accelerators in the prototype, means that the following thesis statement can be derived:

- Accelerating key operations in hardware is crucial to achieve real-time performance in an automated digital surveillance system.

1.3 Thesis Overview

This thesis consists of five chapters and five parts. The chapters comprise introductions, backgrounds, overlapping definitions and commonly shared concepts from the parts, while the main contributions of this thesis, as described in Section 1.2, are placed in the five parts. The research project has resulted in a number of articles and conference contributions, of which the most important constitute the foundation of this thesis. In order to highlight the correlation between the publications and in which part they are placed, an overview of the content in the form of a compilation of the abstracts from these publications are compiled below:

Part I - Low Complexity Binary Morphology With Flat Rectangular Structuring Elements

This part describes and evaluates three hardware architectures for binary morphological erosion and dilation. Since the architectures are intended to be used as hardware accelerators in real-time embedded system applications, the objective is to minimize the number of operations, memory requirement, and memory accesses per pixel. Therefore, a fast stall-free low complexity architecture is proposed that takes advantage of the morphological duality principle and structuring element decomposition. The main advantage of this architecture is that for the common class of flat and rectangular structuring elements, complexity and number of memory accesses per pixel is independent of both image and structuring element size. Furthermore, by exploring parallelization, the memory requirement can be minimized. An evaluation of the three architectures is presented in terms of complexity, memory requirements and execution time, both for an actual implementation, and as a function of image resolution and structuring element size. The architecture is implemented in the UMC 0.13 μm CMOS process using a resolution of 640×480 and supporting a maximum structuring element of 63×63.

The content in this part are modified versions of the following publications:

- H. Hedberg, F. Kristensen, and V. Öwall, "Low Complexity Binary Morphology Architectures with Flat Rectangular Structuring Elements," accepted for publication in *IEEE Transactions on Circuits and Systems I.*
- H. Hedberg, F. Kristensen, P. Nilsson, and V. Öwall, "A Low Complexity Architecture for Binary Image Erosion and Dilation using Structuring Element Decomposition," in *Proc. of IEEE International Symposium on Circuits and Systems (ISCAS'05)*, Kobe, Japan, May 2005.

The main contribution of these publications has been developed in close cooperation with PhD colleague F. Kristensen.

Part II - Binary Morphology with Locally Adaptive Structuring Elements: Algorithm and Architecture

This part describes a novel algorithm with a corresponding architecture which supports a locally adaptive structuring. Allowing locally adaptive structuring elements is advantageous whenever one can let the structuring element locally adapt to certain high-level information, e.g. apparent size of the objects, texture, or direction. For example, in real-time automated video surveillance applications, letting the structuring element locally adapt to the apparent size of the objects, i.e. explore the depth information, makes processing of the binary segmentation result more efficient and accurate. Therefore, in an effort to enhance performance, this paper presents a novel algorithm for binary morphological erosion with a flexible structuring element, together with a corresponding hardware architecture. The algorithm supports resizable rectangular structuring elements, and has a linear computational complexity and memory requirement. In order to achieve high throughput, the proposed architecture maintains the important raster-scan pixel processing order, and requires no intermediate storage for the image data. The paper concludes with implementation results of the architecture when targeted for both FPGA and ASIC.

The content in this part is a modified version of what has been submitted for publication in:

- H. Hedberg, P. Dokladal, and V. Öwall, "Binary Morphology with Locally Adaptive Structuring Elements: Algorithm and Architecture," first round of revision for publication in *IEEE Transactions on Image Processing.*

The foundation of this publication was initiated during a research visit to the Center of Mathematical Morphology (CMM) in Fontainebleu, France. The work has been carried out in close cooperation with PhD P. Dokladal, and the author's main responsibility in this publication concerns the hardware architecture and its implementation result.

Part III - An Architecture for Calculation of the Distance Transform Based on Mathematical Morphology

This part presents a hardware architecture for calculating the city-block and chessboard distance transform on binary images. It is based on applying parallel morphological erosions and adding the result, enabling preservation of the raster pixel scan order and having a well defined execution time. The low memory requirement makes the architecture applicable in any streaming data real-time embedded system environment with hard timing constraints, e.g. set by the frame rate. Depending on the application, if a priori knowledge of the image content is known, i.e. the maximum size of the clusters, this information can be explored reducing execution time and memory requirement even further. An implementation of the architecture has been verified on an FPGA in an embedded system environment with an image resolution of 320×240 at a frame rate of 25 fps running at 100 MHz. Implementation results when targeted for ASIC are also included.

- H. Hedberg, and V. Öwall, “An Architecture for Calculation of the Distance Transform Based on Mathematical Morphology,” to be submitted for publication, 2008.

Part IV - Implementation of Labeling Algorithm Based on Contour Tracing with Feature Extraction

This paper describes an architecture of a connected-cluster labeling algorithm for binary images based on contour tracing with feature extraction. The implementation is intended as a hardware accelerator in a self contained real-time digital surveillance system. The algorithm has lower memory requirements compared to other labeling techniques and can guarantee labeling of a predefined number of clusters independent of their shape. In addition, features especially important in this particular application are extracted during the contour tracing with little increase in hardware complexity. The implementation is verified on an FPGA in an embedded system environment with an image resolution of 320×240 at a frame rate of 25 fps. The implementation supports the labeling of 61 independent clusters, extracting their location, size and center of gravity.

The content in this part is a modified version of the following publication:

- H. Hedberg, F. Kristensen, and V. Öwall, “Implementation of Labeling Algorithm Based on Contour Tracing with Feature Extraction,” in *Proc. of IEEE International Symposium on Circuits and Systems (ISCAS’07)*, New Orleans, USA, May 2007.

Part V - An Embedded Real-Time Surveillance System: Implementation and Evaluation

This part describes an HW implementation of an embedded automated digital video surveillance system with tracking capability. The system is partitioned so that it has key operations implemented as dedicated HW accelerators, e.g. video segmentation, morphological filtering and labeling, while tracking is handled in software running on an embedded processor. By implementing a complete embedded system, bottlenecks in computational complexity and memory requirements can be identified and addressed. Accordingly, a memory bandwidth reduction scheme for the video segmentation unit is deployed together with the development of efficient low memory requirement architectures for morphological and labeling operations. Furthermore, system level optimizations are also explored and applied, e.g. the application does not require unique labels which reduce the memory requirement in the labeling unit and thereby also the total memory requirement in the system. The hardware accelerators provide the tracking software with object properties, i.e. metadata, resulting in the complete decoupling of the tracking algorithm from the image stream, which is crucial to achieve and sustain real-time performance. A simplified system prototype is running on an FPGA development board using a resolution of 320×240 and a frame rate of 25 fps. Furthermore, the impact on the system's resource utilization (scalability) when increasing the resolution is also investigated.

The content in this part is mainly a reprint of the following publication:

- F. Kristensen, H. Hedberg, H. Jiang, P. Nilsson, and V. Öwall, "An Embedded Real-Time Surveillance System: Implementation and Evaluation," accepted for publication in *Journal of VLSI Signal Processing Systems*, Springer.

Despite earlier efforts to avoid repetition and overlap, since the content in this part is mainly an unmodified reprint, overlapping definitions, figures, tables, and results regarding morphology and labeling are obvious. The author's main responsibility in this contribution are the parts concerning morphology and labeling.

Chapter 2

Digital System Design

Designing digital processing systems is a complex task that involves many abstraction levels of digital circuit design. A traditional overview of the abstraction levels used in digital circuit design is illustrated in Figure 2.1 [6]. The major part of work presented in this thesis is located at the system- and architecture-levels. Furthermore, there are many design specific parameters to consider, e.g. design time, cost, throughput (speed), power, area, and flexibility. Many of these parameters depend on each other, e.g. increasing the throughput may result in an increased power dissipation and area, making the design process even more difficult. The subsequent sections will describe some of the design parameters considered during the development of the architectures presented in this thesis.

2.1 Implementation Platform

One of the most important decisions to make when developing a digital signal processing system is the choice of implementation platform. Here, the implementation platform is used as the designation for the hardware on which the actual processing is performed. The choice involves a trade-off between design time, development cost, flexibility, and performance in terms of throughput and energy efficiency, i.e. energy consumed per operation. There are mainly four implementation platforms to choose from:

GPP – General Purpose Processor,

DSP – Digital Signal Processor,

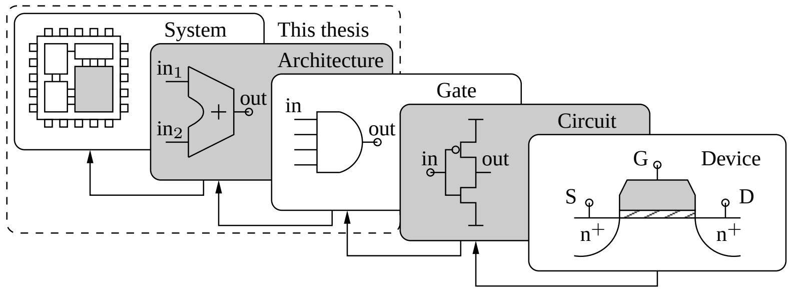


Figure 2.1: An overview of typical abstraction levels used in the field of digital circuit design, where the arrows indicate increasing level of abstraction. The figure also shows on which level the main part of the work in this thesis has been carried out, i.e. architecture and system level.

FPGA – Field Programmable Gate Array, and

ASIC – Application-Specific Integrated Circuit,

which are categorized and placed in Figure 2.2(a) depending on the degree of specialized architecture they are based upon. The more specialized the architecture, the more optimized it is for a specific application. This makes the categorization stretch from a GPP being the most general, to a dedicated full custom ASIC being the most optimized. Note that since the categorization is based on the architecture, a GPP implemented as an ASIC still falls under the category of GPP, a DSP implemented on an FPGA is still a DSP, and so on. There are also hybrids that can be placed in several categories, e.g. structured ASICs being a combination of an FPGA and an ASIC. However, since these hybrids do not add anything to the conceptual discussion conducted here, they are simply omitted.

A GPP is found in common desktop computers, typically having a generic instruction set that is not optimized for any particular application [7]. Since a GPP offers the possibility of running practically any code, thus supporting both floating point and fix-point operations, there are several issues limiting their applicability in real-time processing systems. The main reason for this limitation is due to the fact that a GPP does not offer any type of specialized hardware support for specific or repetitive operations found in digital processing algorithms. As a consequence, this type of implementation platform has a throughput and an energy efficiency among the lowest of all four categories.

The second category is the DSP, which is a type of processor specialized for digital signal processing applications [8]. A DSP has a specialized instruction

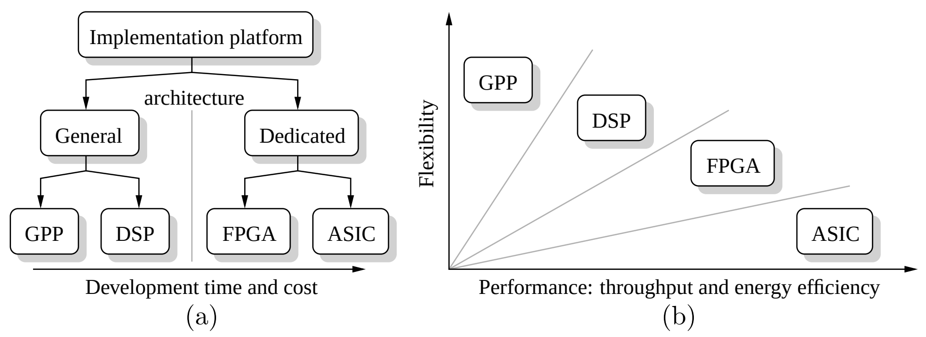


Figure 2.2: (a) The four categories divided by their architectural properties. (b) Flexibility as a function of performance for the four major categories of implementation platforms.

set with dedicated hardware support for operations commonly used in digital signal processing algorithms, e.g. Multiply-Accumulate (MAC), and extended memory support [9]. Having these extensions, this type of processor has both higher throughput and a better power efficiency. Regarding the supported number representation, there are DSPs that support floating point operations, but the majority only supports fix-point operations [10].

An FPGA is a reconfigurable implementation platform which typically consists of logic blocks, interconnects (routing), and I/O blocks [11]. Their reconfigurability comes from the fact that the logic blocks and routing can be programmed. Therefore, an FPGA offers the possibility of exploiting parallelism, resulting in an increased performance compared to GPPs and DSPs. Nowadays, most FPGAs come with embedded on-chip macro blocks (dedicated HW blocks), e.g. multipliers, memory, and in some cases even embedded GPPs. In addition, the vendors may provide the user with software tools to generate common arithmetic components, increasing their applicability in digital processing systems even further. However, one major drawback of this implementation platform is the limited support of floating point-arithmetic operations. Another drawback is the increased development time of an FPGA solution compared to the one based on a GPP or a DSP, somewhat taking the focus from the actual algorithmic development. However, the support for floating-point operations is continuously increasing as well as the ease-of-use of the Electronic Design Automation (EDA) tools from the FPGA vendors. From a power perspective, the configurability comes at the cost of a high power dissipation, e.g. due to the over dimensioned interconnect structure and the use of

Lookup Tables (LUT) instead of dedicated gates, making this implementation platform less applicable in low power applications.

The last category is the ASIC, which offers the highest performance in terms of throughput and power efficiency, but has the longest development time of all categories. ASICs and FPGAs have many properties in common, e.g. the possibility of exploiting parallelism, but with the main difference that the architecture in ASICs can not be altered. Hence, all algorithmic flexibility and reconfigurability must be supported and included in the design before fabrication. The increased performance in terms of throughput and power consumption in an ASIC over an FPGA is due to optimized arithmetic operations, design structures, and routing, together with the possibility of being able to apply more refined low power techniques, discussed further in Section 2.4 and 2.5. Furthermore, if the implementation is to be mass produced, implementing an ASIC may be more cost effective (even if maximum performance is not required) since the production cost per chip will decrease. This leads to the conclusion that if flexibility may be traded for high throughput and low power consumption together with the fact that the budget may cover the increased design time associated with developing an ASIC, this is the best implementation platform for the job. In fact, the superior performance makes this implementation platform the only choice in certain applications, e.g. medical applications [12].

Traditionally, the four categories are shown as a function of flexibility and performance. Performance is relatively easy to measure and the horizontal axis in Figure 2.2(b) very well corresponds to reality. However, since flexibility is not easy to measure, the placement of the categories on the vertical axis may be argued. As an example, an FPGA with a hard embedded GPP macro block is more flexible than a single GPP core, since it gives access to both the strengths of the GPP and offers the possibility of placing key operations in dedicated hardware accelerators (architectures). Again, by omitting the hybrids and by letting the vertical axis correspond to the applicability of a system solution based on this category to various applications, stating that the GPP is the most flexible and an ASIC is the least, is not that far from the truth.

To summarize: No matter how fast a GPP or DSP will become in the future, having a dedicated hardware accelerator placed in an FPGA or an ASIC will always be advantageous in terms of throughput, due to the ability to exploit parallelism [9]. Although the FPGAs of today are not really targeted for low power applications, the continuous development of new low power techniques incorporated in every new FPGA generation is closing the gap between FPGAs and ASICs in terms of low power consumption.

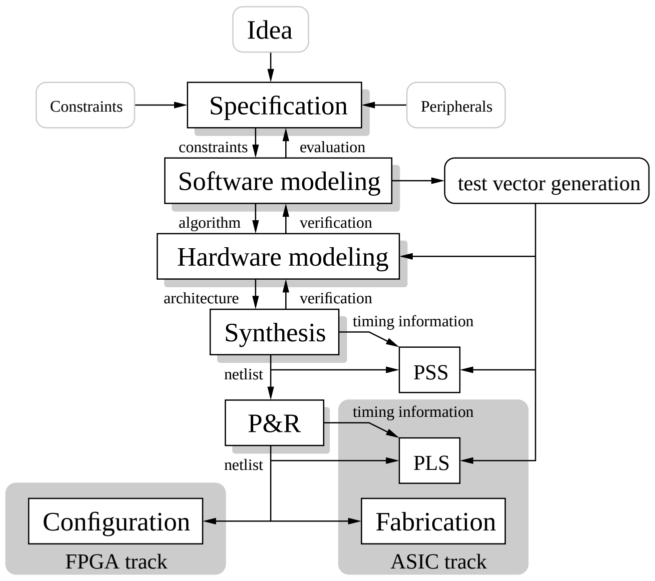


Figure 2.3: An overview of the design flow used when developing the architectures presented in this thesis.

2.2 Design Flow

A digital hardware design flow defines the steps when transforming a specification into a reproducible implementation, e.g. an embedded system running on an FPGA or an ASIC. A well defined design flow not only ensures the quality of the outcome but also reduces design time, since the time spent in each step can be estimated and measured, so resources may be allocated accordingly. Figure 2.3 illustrates details of a simplified design flow for hardware design using a Hardware Description Language (HDL) targeted for an FPGA or an ASIC, that covers the main development process used in this research project. Naturally, a design flow is application dependent, and so is the time spent in each step, which is dependent on several factors, e.g. algorithmic complexity, size of the design, and speed constraints.

The design process starts with an idea about what is to be developed, e.g. a hardware accelerator or a complete embedded system, which is turned into a specification. Starting at the top of Figure 2.3, the main steps in a hardware design flow are:

Specification - contains detailed information of what is to be developed with available resources, e.g. design time, budget, technology, interfaces, and peripherals.

Software modeling - is used to identify and model key components, typically performed using a high level description language, e.g. C, C++, or Matlab. Software modeling is a fundamental step in the design flow since it includes algorithm selection, software/hardware partitioning, timing evaluation, deciding arithmetic precision, evaluation of parameter settings, and complexity analysis of the components. In addition, test vectors to and from the components are generated to be used for verification of succeeding steps.

Hardware modeling - describes the hardware components' behavior on a bit-level, typically described in VHDL or Verilog. This step determines the actual hardware architecture of the components.

Synthesis - transforms the hardware model into a netlist. A netlist is a description of the hardware model in terms of physical gates, which are selected from a standard cell library, and their relative interconnection.

Place and route (P&R) - physically places the gates, and routes the interconnections of the netlist on the FPGA or the die (in case of an ASIC implementation).

Configuration/Fabrication - depending on the target implementation platform, the FPGA track ends in a programming file that configures the FPGA, and the ASIC track ends by sending the design for fabrication.

When designing an ASIC, a good strategy to verify the functionality of the design is to extract timing information for the netlist and to perform a Post Synthesis Simulation (PSS) or Post Layout Simulation (PLS) before manufacturing the design, illustrated in Figure 2.3. Rapid prototyping may also be performed using an FPGA to test the functionality and interfaces between the processing units. Naturally, there are practical situations when functional verification on an FPGA is superfluous or even not feasible, e.g. when the design is too large, or when full custom blocks on a circuit level are being developed. This is due to the fact that full custom blocks may consist of gates, transistors, or have a routing that is not possible to achieve in an FPGA. However, if possible, performing PSS, PLS or prototyping on an FPGA, minimizes the risk of sending erroneous designs for fabrication and should therefore always be considered. Note that the netlists may also be used to simulate the power consumption of the design, which gives an indication of how much power will be dissipated in the design.

2.3 Real-time Processing

In a general real-time processing system, the system is required to respond to an input event within a predestined time using given resources so that it keeps up with an external process [7]. In an image sensor based processing system, the outside process is the image acquisition step. The speed of the image acquisition step is equivalent to the frame rate f_{rate} and is measured in frames per second (fps) or Hz. The frame rate is a critical parameter for system performance since it affects the timing budget by putting an overall constraint on the system processing time t_p, i.e. the total time it takes for the system to process a complete frame. System latency is also an important property which corresponds to the delay between an event and the system's response to this event. In practice, the latency corresponds to the delay between input and output data, and should be kept as low as possible. Assuming a single processing block, regardless of latency and other system parameters and settings, e.g. resolution, the following relation must hold

$$t_p \leq \frac{1}{f_{rate}}. \tag{2.1}$$

The frame rate depends on the system resolution and may have limited degrees of freedom, as they are sensor specific. This means that a particular sensor supports certain modes of operation, i.e. a certain resolution results in a given f_{rate}. The higher the f_{rate}, the shorter the time which has elapsed between two consecutive frames, and the more "similar" or more quasi-stationary the two images become. This may be important in certain applications, e.g. automated surveillance, since object motion predictions become more accurate resulting in a more robust tracking. However, a high frame rate imposes a high bandwidth or bit rate to and from the hardware blocks, e.g. memories, which is a major bottleneck in many image processing systems [13]. A high bandwidth also consumes more power, which is always an issue in any system design and may be critical in battery operated devices. From a system perspective, f_{rate} should be kept as low as possible while still being able to process the fastest event in the scene.

Deciding a frame rate is more than just a question of processing speed and algorithmic robustness, particularly if the output is a video sequence that is to be viewed by humans. In general, humans perceive a motion as continuous if $f_{rate} \geq 20$ [14], but this figure depends to a large extent on the image content. However, many people still experience discontinuities in the motion at this frame rate, and as a consequence, common television standards use $f_{rate} \geq 25$, e.g. PAL=25 and NTSC=30 fps [15](PAL=50 and NTSC=60 fps interlaced). Based on this discussion, in this thesis, real-time video performance is defined as $f_{rate} \geq 25$ fps and as presenting the result within a latency of a few frames.

2.4 Low Power Circuit Design

The total power dissipation P_{tot} in a digital integrated circuit consists of three components [6]: static power P_{stat}, dynamic power P_{dyn}, and direct-path power P_{dp}. P_{stat} is due to leakage currents, I_{leak}, which are both temperature and technology dependent: increasing as temperature rises, and as the threshold voltage in the transistors is lowered, which occurs when technology is scaled down. P_{dyn} is due to the charging and discharging of capacitive loads C_l of a transistor. P_{dp} is due to the nature of the Complementary Metal Oxide Semiconductor (CMOS) design in which a direct-path between the supply voltage and ground is present during the switching phase of the gates. This direct path results in a short circuit current I_s, present during a short period of time t_s, when both transistors are conducting.

All power contributions, P_{stat}, P_{dyn}, and P_{dp}, are dependent on the supply voltage V_{dd}, but only P_{dyn} and P_{dp} are proportional to the clock frequency f. The total power dissipation in a design is defined as

$$P_{tot} = P_{stat} + P_{dyn} + P_{dp} = V_{dd} \cdot (I_{leak} + f \cdot (C_l \cdot V_{dd} + I_s \cdot t_s)), \qquad (2.2)$$

where P_{dp} may be neglected since it is small compared to the other two components. Traditional low power design techniques and tools focus on minimizing the dynamic power since it is traditionally the largest component in (2.2). Typical low power design techniques are: voltage scaling, power-gating, multiple threshold cell libraries, and clock-gating, among which voltage scaling is popular due to the square relationship in (2.2). Furthermore, the propagation delay t_p of a given gate depends on the threshold voltage V_t, and the supply voltage according to

$$t_p \propto \frac{V_{dd}}{(V_{dd} - V_t)^2}, \qquad (2.3)$$

which shows that power is traded for speed [16]. Both increasing V_t, which reduces the static power, and decreasing V_{dd}, which reduces the dynamic power dissipation in (2.2), result in a longer propagation delay, and hence slower circuitry. In practice, (2.3) is the basis of lowering the supply voltage so that the t_p of the design matches the timing constraint imposed by the application, e.g. resolution and frame rate. To further refine this technique, (2.3) may be applied on specific regions of the chip, resulting in multiple power domains with an optimal supply voltage within that region [17]. Based on this reasoning, any time slack in the timing model may be used to decrease the dynamic power consumption by lowering the supply voltage.

2.5 Low Power FPGA Design

In FPGA design, the power contribution principles discussed in Section 2.4 still holds but the freedom of applying various low power techniques is limited to what is supported by the specific FPGA model, e.g. voltage scaling and power-gating [18]. Due to the recent power awareness among the FPGA vendors, nowadays, low power techniques are also included in the EDA tools, e.g. power driven placement [19]. The power saving when applying these techniques is highly application dependent, but their applicability to the design should always be explored. In addition, there are also other less FPGA model and vendor dependent low power techniques that may be applied to the design:

- **System level** - manipulates the switching activity, e.g. multiple clock domains each operating at the lowest possible frequency and clock-gating.
- **Architectural level** - reduces the number of memory accesses and computational complexity.

A typical system level power saving technique is to optimize the throughput frequency balance. This means minimizing the operating frequency of the individual blocks but keeping the throughput constraint (assuming a fixed V_{dd}). This technique will save power since the clock distribution network has high switching activity together with large capacitances, f and C_l in (2.2). This may be achieved since the FPGA typically supports multiple clock domains. Clock-gating may effectively reduce the switching activity in the clock tree by not letting the clock toggle inside unused blocks. If a block is unused, gating the clock to the registers prohibits unnecessary switching activity in the combinatorial parts and signals to further propagate through the system, dissipating unnecessary power. However, the power savings when applying clock-gating in an FPGA are not as high as in the ASIC counterpart, which is due to the high static power consumption in an FPGA [20]. Furthermore, clock-gating is not applicable when the blocks are continuously executing, limiting the power savings in certain applications even further, e.g. the system described in Part V.

On the architectural level, memories and memory accesses are power consuming since they typically have high switching activity and large capacitances on the bit-lines. This is especially true for off-chip memories which have additional capacitances on the I/O-ports. Thus, much effort should be spent on developing algorithms with a minimized memory requirement. Furthermore, another strategy to save power is to reduce the computational complexity. As an example, $a(b + c)$ is preferred over $ab + ac$, since one multiplier less is used and will therefore consume less power [21].

Chapter 3

Digital Image Processing

This chapter presents basic digital image processing concepts and definitions relevant to this thesis. The intention is that it should supply the reader with a theoretical background which is used and referenced in subsequent chapters.

3.1 Digital Image Acquisition

A first step in any digital image processing system is to capture an input image or frame I, typically performed by an image sensor. The sensor produces a frame by spatially dividing a light sensitive region into an ordered array of picture elements referred to as pixels, which are aligned to a grid or lattice, with $I_h = M$ rows and $I_w = N$ columns. This spatially ordered light capturing procedure is referred to as *spatial sampling.* The lattice constitutes the domain D of I and is typically a subset of $\mathbb{Z}^2$ for image sensors. Image resolution is defined as the number of pixels per frame, i.e. $M \times N$. Although $M < N$ holds for most commercial resolutions, for the sake of simplicity when describing algorithmic properties, e.g. memory requirement, an image is sometimes assumed to be a square with a resolution of N^2 pixels. Inferring a coordinate system on the array, the origin with coordinates $(0, 0)$ is often defined to be located in the upper left corner, as depicted in Figure 3.1. Having the origin defined, each pixel p has corresponding coordinates (i, j), where $i = \{0, \ldots, M-1\}$ and $j = \{0, \ldots, N-1\}$, defining the pixel's spatial position in the grid. Furthermore, each pixel is mapped onto a set of discrete values corresponding to the light intensity level. This value is quantized and taken from a set of discrete values $V = \{v_{min}, \ldots, v_{max}\}$, where v_{min} and v_{max} are the minimum and maximum possible intensity values. Mathematically, I can now be described

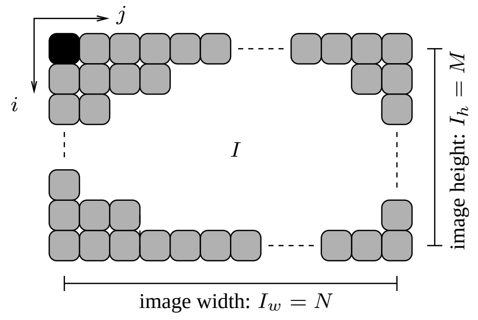
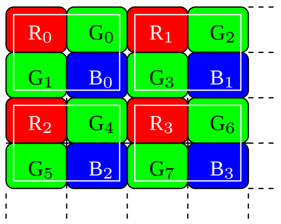


Figure 3.1: A digital image and the coordinate convention used in this thesis, where the pixels are shown in gray and the origin is marked in black.

as a function that maps a certain spatial domain onto this set of values V, and may be written as $I : D \to V$. Without color processing, the cardinality of V corresponds to the number of gray levels in the frame, typically a power of two, i.e. $2^1 = 2$ or $2^8 = 256$. Typically, $v_{min} = 0$ which makes the binary representation of v_{max} determine the number of bits in hardware required to represent the maximum pixel value and thereby also the dynamic range for this frame. As an example, to represent and store a binary image I in a memory, since $V = \{0, 1\}$ which implies $|V| = 2$, only one bit per pixel is required.

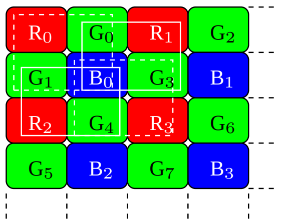
3.1.1 **Human Vision**

A color is a light source with a certain wavelength distribution, where the wavelengths that stretch from about 400 to 700 nm lies within the human visual spectrum [22]. A light source with a uniform wavelength distribution within the visual spectrum is referred to as white, and a light source containing only one wavelength is monochromatic (dirac distribution). The sum of the wavelength distribution is equal to the intensity. Furthermore, since a sensor only measures spatial light intensity, it becomes gray scale by nature. However, to be able to keep the wavelength distribution information, a color space is inferred, splitting the intensity into different and separate wavelength contributions: each color pixel is represented as a point in the color space with each color component contribution as the projection onto each respective axis. A common technique to create a multi-dimensional color space, e.g. RGB, is to separate the wavelengths of the incoming light by inferring a color filter on top of the pixel grid. The filter, called a Color Filter Array (CFA), is periodic and the actual colors in the filter corresponds to the desired color component. As an example, a



$$p_{(0,0)} = (R_0, \tfrac{G_0+G_1}{2}, B_0)$$
$$p_{(0,1)} = (R_1, \tfrac{G_2+G_3}{2}, B_1)$$
$$p_{(1,0)} = (R_2, \tfrac{G_4+G_5}{2}, B_2)$$
$$p_{(1,1)} = (R_3, \tfrac{G_6+G_7}{2}, B_3)$$

(a)



$$p_{(0,0)} = (R_0, \tfrac{G_0+G_1}{2}, B_0)$$
$$p_{(0,1)} = (R_1, \tfrac{G_0+G_3}{2}, B_0)$$
$$p_{(1,0)} = (R_2, \tfrac{G_1+G_4}{2}, B_0)$$
$$p_{(1,1)} = (R_3, \tfrac{G_3+G_4}{2}, B_0)$$

(b)

Figure 3.2: An illustration of a Bayer mask together with examples of two pixel setups. (a) is the one used in the system described in Part V.

CFA with one red, two green, and one blue filter component will produce the *RGB* color space. This filter, depicted in Figure 3.2, is of special interest since it is commonly used by sensor manufacturers and is named as the Bayer filter after its inventor [23]. After the *RGB* color intensities have been measured, the sensor output pixels are created as a set of three color components. Since there are four color components in the CFA and only three in a pixel, a decision on how to create the three output values has to be made. When using the Bayer CFA, this is usually done by manipulating the green components with various techniques, e.g. calculating the mean of two green values, illustrated in Figure 3.2(a), or by using each color value in multiple pixels, illustrated in Figure 3.2(b).

The human eye has two types of receptors [24]: rods and cones. Rods are only sensitive to incoming light intensity and cones only to color information. Furthermore, cones are divided into three subtypes each sensitive to a specific color: blue, green, and red. The reason for choosing two green pixels in the CFA is that the human eye is more sensitive to green than to red or blue. This is due to the fact that the sum of the probability of the color receptors absorbing a light quantum with a certain wavelength has its maximum around ≈ 550 nm, which corresponds to a greenish color, illustrated in Figure 3.3.

3.1.2 Sensor Techniques

There are two major sensor techniques: Charge Coupled Device (CCD), and CMOS. Both techniques were developed in the late 60's and early 70's. First,

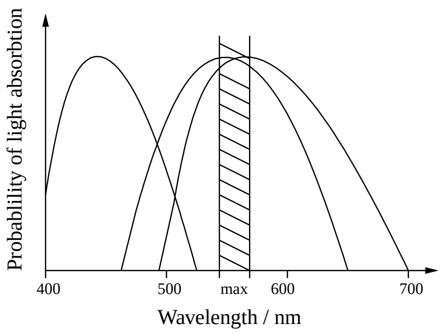


Figure 3.3: Normalized spectrum sensitivity of the human eye for each of the three cone receptor subtypes versus wavelength [14]. The sensitivity is proportional to the probability of a receptor absorbing a light quantum with a specific wavelength.

CCDs became dominant due to superior image quality, and not until the 90's could the fabrication methods deliver the dimensions and uniformity needed when using the CMOS technology. As a consequence, CCDs were found in high-end cameras and CMOS sensors in budget models. However, nowadays this distinction is not valid anymore, since CMOS sensors are also placed in high-end models [25]. The subsequent section presents a brief comparison between the two techniques [26] [27]. However, before going into details, some important parameters are presented that may be used to compare the two techniques and which are of special interest in automated digital surveillance systems:

- **Dynamic range** - The ratio between a pixel's saturation level and its signal threshold, i.e. the ratio between maximum and minimum signal level which should be kept as high as possible.
- **Uniformity** - Consistency in pixel response for different spatial locations under identical illumination conditions, i.e. the same light intensity at different locations of the sensor should result in the same output pixel value. The uniformity may vary at different illumination intensities.
- **Speed** - The data rate of the sensor output pixel stream, measured in MHz.

3.1.3 CCD versus CMOS Sensors

Both CCD and CMOS techniques are Metal Oxide Semiconductors (MOS) and function as spatial light samplers; they both have a light sensitive region that

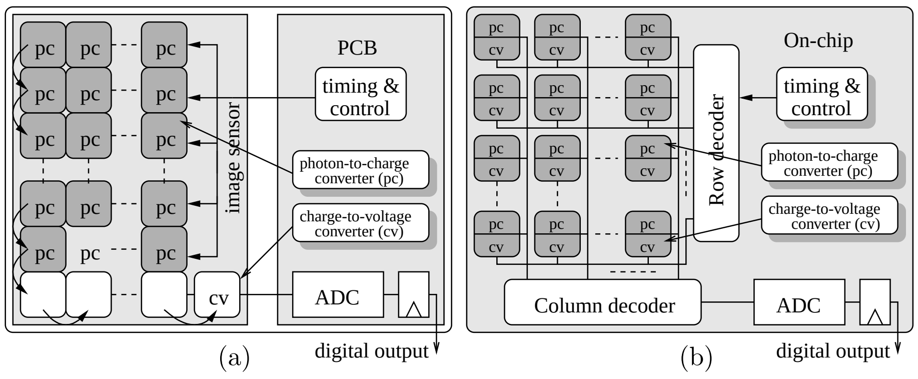


Figure 3.4: (a) A typical CCD sensor, where the photon-to-charge converter is placed inside the pixel array and the charge-to-voltage converter is placed outside of the pixel array. (b) A typical CMOS sensor, where both the photon-to-charge and charge-to-voltage converters are placed directly in the pixels.

converts photon quantums into charge at each location in the pixel grid. The electrical charge is converted into a voltage and finally sent to the output. However, it is the read-out procedure of the charge that is the major difference between the two techniques.

A CCD sensor transfers the charge quantum sequentially from one photon-to-charge converter to another towards the output charge-to-voltage converter. A typical CCD sensor with some required circuitry and logic blocks is depicted in Figure 3.4(a).

In a CMOS sensor, illustrated in Figure 3.4(b), the photon-to-charge and charge-to-voltage conversions takes place directly in the pixel. Therefore, signal routing to each pixel is required, reducing the pixel density to some extent.

Comparing the dynamic range between the sensor techniques, the CCD has an advantage due to less noise in the substrate. Traditionally, the uniformity was problematic in CMOS sensors, but the gap is continuously decreasing. The image pixel output speed is one of the most important parameters since at a given frame rate, a faster pixel output speed results in a longer processing time per frame for the system. In general, CMOS sensors have a slight advantage over CCDs in terms of speed. Another important issue is the amount of required post-processing. CMOS sensors typically have more post-processing integrated on-chip, e.g. timing generation, Analog-to-Digital Conversation (ADC), and noise reduction, as opposed to the CCD which has most post-processing on the

Table 3.1: A comparison of important parameters between the CCD and the CMOS sensor techniques.

Sensor property	CCD	CMOS
Dynamic range	Better	Good
Uniformity	Better	Good
Speed	Fast	Faster
Power	Higher	Lower
Cost	More	Less

camera Printed Circuit Board (PCB). This, together with the fact that CMOS sensors can be manufactured in standard MOS processes, makes the cost per sensor less for CMOS than for CCDs. Based on this brief evaluation, summarized in Table 3.1, together with the fact that the CMOS sensor consumes less power, this type of sensor is more suitable in our application.

3.1.4 Raster Scan Order

An important concept in real-time processing is the data pattern of the input stream. This is important since it has a large impact on several aspects of hardware design, e.g. the required amount of hardware resources, the timing model, and type of memory. In many digital real-time image processing applications, the data pattern is a stream of sequential pixels starting in the top left and ending in the bottom right corner of the image, which is referred to as the raster scan order. This is a typical read-out pattern from a sensor or when multiple pixels are burst read from a memory source. To avoid intermediate data storage and random memory accesses during the pixel processing, and to achieve minimal system latency, the pixel processing chain should maintain the raster scan property as long as possible. In practice, this means sequential input and sequential output, to and from the included HW blocks.

3.2 Fundamental Pixel to Pixel Based Relationships

3.2.1 Neighborhood

A commonly used spatial pixel neighborhood defined on a square lattice can be formed by taking a square of size 3×3, centered around a pixel p at location (i, j). This pixel has two horizontal and two vertical neighboring pixels at coordinates

$$(i, j+1), (i+1, j), (i, j-1), (i-1, j), \tag{3.1}$$

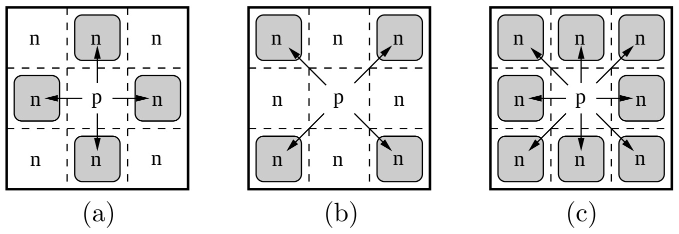


Figure 3.5: (a) A pixel p and its 4-neighbors ($N_4(p)$), (b) D-neighbors ($N_D(p)$), (c) and 8-neighbors ($N_8(p)$).

defined as the 4-neighbors of $p(i, j)$ and denoted $N_4(p)$, depicted in Figure 3.5(a). Furthermore, $p(i, j)$ also has four diagonal neighbors at coordinates

$$(i-1, j+1), (i+1, j+1), (i+1, j-1), (i-1, j-1), \tag{3.2}$$

denoted $N_D(p)$, depicted in Figure 3.5(b). These pixels, together with $N_4(p)$, are defined as the 8-neighbors denoted $N_8(p) \in N_4(p) \cup N_D(p)$ [28]. In words, a pixel and its closest neighboring pixels in all directions, are often referred to as nearest neighbors, illustrated in Figure 3.5(c).

3.2.2 Connectivity

The relationship between two or more pixels is called *connectivity* [29]. For two pixels p_1 and p_2 to be connected, they must both fulfill the *adjacency criterion* and be in the same neighborhood. The *adjacency criterion* or *adjacency* considers the pixels' intensity value. Given the intensity values of two pixels p_1 and p_2 taken from a discrete set V, defined in 3.1, the adjacency criterion is fulfilled if a predefined condition is met, e.g. $|p_1 - p_2| < T_{threshold}$ or $p_1 = p_2$, where the latter is typical in binary images.

Two pixels p_1 and p_2, in a binary image are

- **4-connected** – if $p_1 \in N_4(p_2)$
- **8-connected** – if $p_1 \in N_8(p_2)$

if they simultaneously fulfill the adjacency criterion. Connectivity is a fundamental concept in digital image processing from which other important concepts are derived, e.g. contours, regions, and distance measures. An example of how 4- and 8-connectivity affects clustering is illustrated Figure 3.6(a)-(c). Notice how the 4-connectivity rule misses diagonal transitions as opposed to the 8-connectivity rule, resulting in three clusters instead of one.

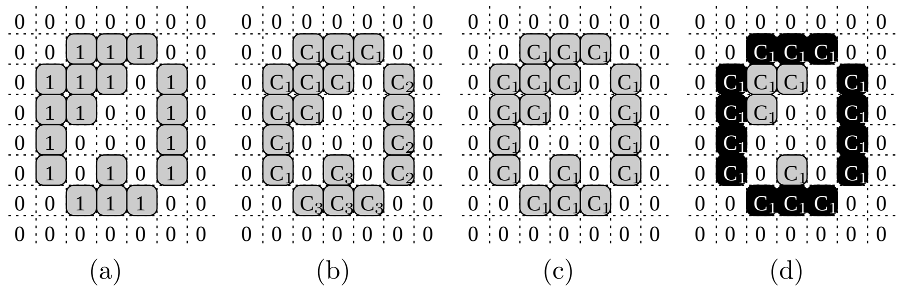


Figure 3.6: (a) An arbitrary binary image, (b) corresponding 4-connected clusters, i.e. C_1, C_2, and C_3, (c) corresponding 8-connected cluster C_1, (d) and corresponding 8-connected contour pixels marked in black.

3.2.3 Clusters

A cluster C is a set of pixels defined by their connectivity, typically 4- or 8-connected. Each cluster has a contour P, which consists of contour pixels p and is defined as

$$P = \{p \mid p \in C, \exists q \in N_4(p), q \notin C\}, \tag{3.3}$$

which means that a contour pixel p has at least one 4-connected neighboring pixel outside of C. An example of 8-connected contour pixels of an arbitrary-shaped cluster is illustrated in Figure 3.6(d).

3.2.4 Spatial Filters

A spatial filter is a mathematical function that typically applies to the center pixel of the sliding window taking input data from the pixels that are currently coinciding with the window in the input frame [30]. A general spatial operator is defined as

$$g(i,j) = T[f(i,j)], \tag{3.4}$$

where $f(i,j)$ is the input pixel, T is the filter operation, and $g(i,j)$ is the output at position (i,j) in the resulting image. The filters are defined as linear if the following condition is fulfilled

$$T[af(i,j) + bg(i,j)] = aT[f(i,j)] + bT[g(i,j)], \tag{3.5}$$

where a and b are any two scalars, and nonlinear if not. Image convolution is an example of a typical linear spatial filter which may be used in a variety of applications depending on the coefficients in the kernel, e.g. image smoothing and sharpening [31]. Let I be a grayscale image in $\mathbb{Z}^2$ of size $M \times N$, then the 2-D image convolution in the spatial domain can be defined as

$$g(i,j) = I(i,j) * b(i,j) = \sum_{s=-\lfloor \frac{S}{2} \rfloor}^{\lfloor \frac{S}{2} \rfloor} \sum_{t=-\lfloor \frac{T}{2} \rfloor}^{\lfloor \frac{T}{2} \rfloor} b[s,t] I[i+s, j+t] \tag{3.6}$$

where $i = 0, 1, \ldots, M-1$ and $j = 0, 1, \ldots, N-1$ and $b(s,t)$ is the convolution kernel of size $S \times T$ containing the coefficients [30] (S and T are assumed to be odd numbers). Examining (3.6), two key arithmetic operations may be distinguished: multiplication and summation. These two operations characterize image convolution, since each value in the input image that is covered by the superimposed convolution kernel is first multiplied with the coinciding coefficient which then are all accumulated, forming the result. When the kernel reaches outside the borders, i.e. when $0 > i+s > M-1$ and $0 > j+t > N-1$, these values are set in such a way that they do not affect the result.

Rank filters are based on internally ordering or *ranking* the intensity values in the input image currently covered by the superimposed kernel, and then performing a nonlinear operation on these values, i.e. which does not fulfill the criterion in (3.5). Typical nonlinear rank filters are formed by taking the median, average, minimum, and maximum, of these values, where the median is particularly important, since it is widely used for noise suppression [30].

3.3 Basic Set Theory Definitions

A general set is a cluster of objects or members, referenced as a whole. The members are grouped by certain rules certifying their membership in the set. In image processing, the sets are collections of pixels that are grouped by their connectivity and/or by fulfilling the adjacency criterion (refer to Section 3.2.2) and are mapped onto a certain lattice, e.g. $\mathbb{Z}^2$ in the 2-D case. Each pixel is associated with a n-tuple of values where n is the smallest integer required to describe the properties of this particular pixel, e.g. its coordinates and intensity [32]. As an example, let A be a set in $\mathbb{Z}^2$ describing foreground pixels in a binary image, i.e. pixels with intensity value 1, then n is equal to 2 since only a pair of coordinates, i.e. i and j, are required to describe the pixels in the set (the intensity is omitted). Throughout this thesis, sets are referenced with capitals and their corresponding elements with the same lower case letter, i.e. A is the set and a its elements.

Let A, B be sets in $\mathbb{Z}^2$, then there are some basic concepts defined on these sets relevant to this thesis, which are

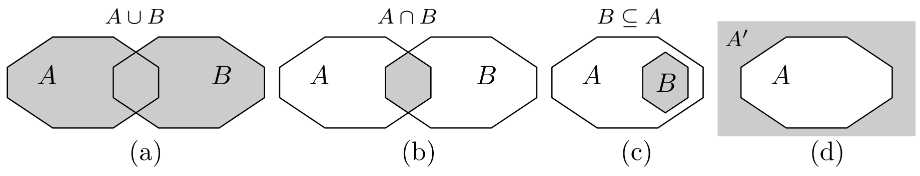


Figure 3.7: Venn Diagram illustrations of basic set theory concepts. (a) Union, (b) intersection, (c) subset, (d) and complement of A.

- union, ($\cup$)
- intersection ($\cap$)
- subset ($\subset$ or $\subseteq$)
- complement (^{c} or ')

The union of A and B is formed by merging the two sets and removing duplicate elements, and is denoted $A \cup B$. The union corresponds to a logical *OR*-operation and is illustrated in Figure 3.7(a). The intersection of A and B, denoted $A \cap B$, is a set constituting the elements that are shared in common by the sets. This corresponds to a logical `AND`-operation and is illustrated in Figure 3.7(b). Note that the result of the intersection between two disjoint sets is an empty set $\emptyset$. B is as a subset of A, if and only if $\forall b \in B$ together with $b \in A$, and is denoted $B \subset A$, illustrated in Figure 3.7(c). This implies that every pixel in B is also contained in the set A, but $A \neq B$ since $\subset$ is analogous to $<$ and $\subseteq$ to $\leq$. The complement of a set A is denoted A^c or A' and constitutes elements that are not in the set, i.e. $a \notin A$, shown in Figure 3.7(d).

3.3.1 Reflection

An important concept is the *reflection* of a set B, denoted $\hat{B}$, and this is defined as

$$\hat{B} = \{\hat{b} \mid \hat{b} = -b, \forall b \in B\},$$

which implies that $\hat{B}$ constitutes elements $\hat{b}$ which are equal to all elements $b \in B$ multiplied by -1, resulting in its geometric inverse about its origin, depicted in Figure 3.8(a). A special case is when B is reflection invariant, which is denoted $B = \hat{B}$. This occurs when the reflection is symmetric in both the i and j direction, e.g. a square or a circle. An example of a reflection invariant set B is depicted in Figure 3.8(b).

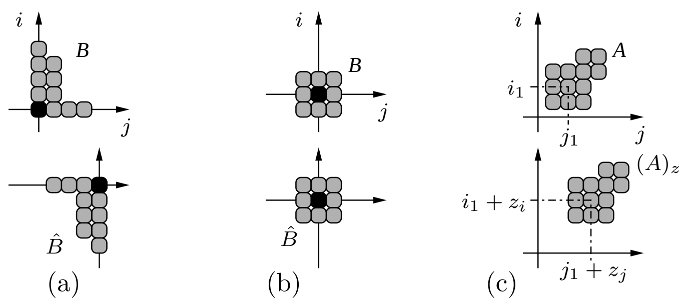


Figure 3.8: (a) A set B and its reflection, where the pixel located at the origin is marked in black, (b) a rectangular set B and its reflection, i.e. $B = \hat{B}$, (c) and a set A translated by z, i.e. A_z.

3.3.2 **Translation**

A set A, translated by a point $z = (z_i, z_j)$, denoted A_z, is defined as

$$A_z = \{a_z \mid a_z = a + z, \forall a \in A\}, \tag{3.7}$$

which means that A_z constitutes elements a_z that are equal to the elements $a \in A$ translated by z, depicted in Figure 3.8(c).

3.3.3 **Minkowski Addition and Subtraction**

Having basic set theoretic operations defined, an important aspect of set theory is that arithmetic operations are defined on the sets where the Minkowski addition and subtraction are relevant to this thesis [33] [34] [35]. The Minkowski addition is defined as

$$A \oplus B = \{a + b \mid a \in A, b \in B\} = \bigcup_{b \in B} A_b, \tag{3.8}$$

where the result is formed by taking the union of the elements after adding every element in B to every element in A. In other words, this can be seen as taking the union of all sets when A is translated $\forall b \in B$. The Minkowski subtraction is defined as

$$A \ominus B = \{x \mid x - b \in A, b \in B\} = \bigcap_{b \in B} A_b, \tag{3.9}$$

which implies that the result is every element in B, subtracted from every element in A. This can be seen as taking the intersection of all sets when A is translated $\forall b \in B$.

Chapter 4

Morphology

The word morphology is a combination of *morphe*, Greek for “form” or “shape”, and the suffix -ology, which means “the study of”. Consequently, the word morphology means the study of shapes. In digital image processing, Mathematical Morphology (MM) is used as the designation for the nonlinear mathematical tools used to manipulate the shape or understand the structure of functions or objects (clusters of pixels). The technique was originally developed by Matheron and Serra [36] at the Ecole des Mines in Paris in the late sixties. Morphology is based on set theory and provides a quantitative description of geometrical structures and plays a key role in many digital image processing applications. Furthermore, morphology was originally developed for binary images, i.e. the 2-D integer space $\mathbb{Z}^2$, but was later extended and now applies to several image representations, e.g. 2-D gray scale integer space, 3-D gray scale and color integer space [37] [38] [39]. However, in this thesis, the scope has been limited to binary morphology with flat structuring elements if nothing else is stated.

There are numerous applications using different binary morphological operations reported in literature, e.g. noise filtering, boundary detection, and region filling [35]. The binary image representation can emerge not only due to the nature of the application, e.g. performing character recognition on a black and white document, but also as output from an image processing step, e.g. intensity thresholding, segmentation, or thresholded convolution [40] [41].

4.1 Structuring Element

All morphological operators are based on evaluating subsets, or local neighborhoods, with a limited geometric area in I. The subsets are extracted by probing

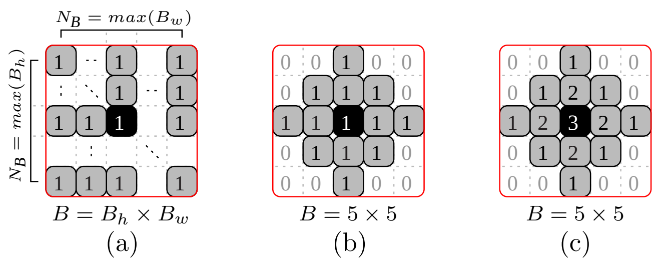


Figure 4.1: (a) An example of a flat SE where $B_{i,j} = 1$ for all $i = \{0,..,\max(B_h) - 1\}$ and $j = \{0,..,\max(B_w) - 1\}$. (b) An example of a flat diamond shaped SE. (c) An example of a non-flat SE. All positions in the SEs where $B_{i,j} \neq 0$ are active and the origins are marked as the shaded regions.

a sliding window (kernel) on I, which is referred to as the Structuring Element (SE) using morphological nomenclature. The SE determines the actual size and geometrical shape of the subsets by selecting the pixels in I to include in the current calculation. The SEs range from fully arbitrary to classes with limited shape, resulting in a reduced complexity e.g. lines, circles, diamonds, and rectangles. As a consequence, the shape of the SE affects the shape of the output and is therefore strongly application dependent [42]. There are no theoretical bounds or limitations on the choice of SE shape. However, a rule of thumb is to use a shape that resembles the shape that is searched for or that is to be distinguished, e.g. if you want to extract high and thin objects, the use of high and thin SEs are preferable. Naturally, algorithms supporting arbitrary SEs are also the most computationally and memory demanding. Therefore, the algorithmically supported shape is a trade-off between computational complexity and what is required in the application.

Mathematically, the SE is a set with an upper limit in size, typically $\max(B_h) \times \max(B_w)$ pixels, with elements mapped onto the same lattice configuration as the input image, e.g. $B \in \mathbb{Z}^2$. Regarding the numerical properties of the elements in SE, if the SE is binary, B is said to be *flat*, i.e. $b = \{0, 1\}, \forall b \in B$, illustrated in Figure 4.1(a). If an element is equal to $b = 0$, the pixel that coincides with this element in I is omitted from the evaluation, i.e. marking *don't care* positions. A diamond shaped SE containing ZEROS is depicted in Figure 4.1(b). Although flat SEs are the most common case and used to filter various image representations, e.g. binary, gray scale, and color images [43], morphology is defined for non-flat SEs as well. The logical operation is then performed on the result after having added the value of each b to the coinciding value in I. A non-flat structuring element is shown in Figure 4.1(c).

Another important task assigned to the SE is to determine the position of the evaluation result in the output frame, which is determined by the *origin* $\in B$. Usually, the origin is located at the center of B, which implies that both $\max(B_h)$ and $\max(B_w)$ are odd integers. Each pixel evaluation around the origin in I, generates an output value with the same coordinates in the output image.

Let I be a binary input image and B a flat and rectangular SE, i.e. $b = 1\ \forall b \in B$. Then B slides over I so that the superimposed origin visits each coordinate in I once. This results in an output image with the same dimension as I and a content based on the evaluation of the subsets determined by B.

4.2 Erosion and Dilation

A morphological operation is an image to image transformation similar to image convolution and rank filters since they are all sliding window operations, as discussed in Section 3.2.4. Furthermore, erosion (ε) and dilation (δ) are two fundamental morphological operations from which many others are derived, e.g. opening (erosion followed by dilation), closing (dilation followed by erosion), and hit-or-miss transformation [36] [30]. Therefore, developing efficient erosion and dilation algorithms and architectures for HW implementation is of the highest interest since it will implicitly make a collection of operations more applicable in the field of embedded image processing.

4.2.1 Erosion

Let $I, B \in Z^2$ represent a binary input image and a structuring element respectively. Erosion is defined as

$$I \ominus B = \{x \mid (B)_x \subseteq I\}, \tag{4.1}$$

which means that the erosion of I by B is a set that contains elements x such that B translated by x is a subset of I [44]. For an arbitrary SE, this means that the output pixel with the same coordinates in the pixel grid as the origin in the SE becomes ONE, if and only if all the pixels in the input image that coincides with a ONE in the SE are equal to ONE. In other words, a binary ε produces a ONE at every position where the superimposed B has the same geometrical shape as I. It can be compared to a logical `AND` operation, or taking the minimum of the pixels in the input image at the coordinates where the SE is equal to ONE. An alternative definition of a binary erosion can be written as

$$\varepsilon_B(I) = \bigcap_{b \in B} I_{-b}, \tag{4.2}$$

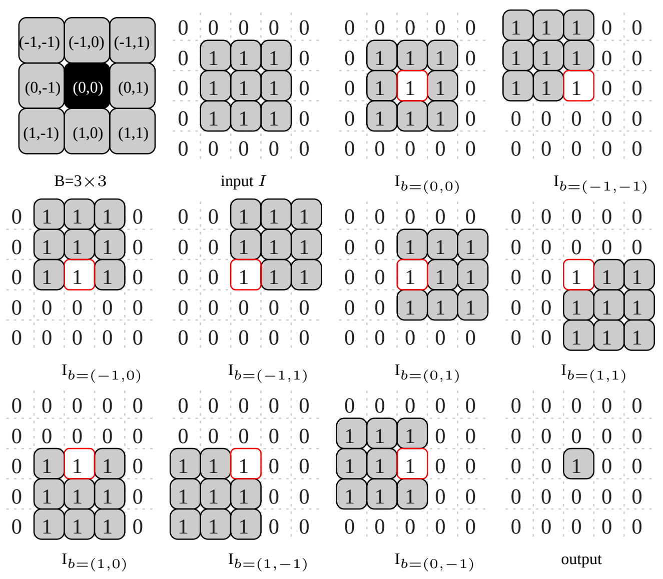


Figure 4.2: A fragment of the partial result images when translating I by the elements in B, in this case a square of size 3×3 with the origin marked in black. By taking the intersection of each translation I, the only pixel present in all images is marked in white, which therefore corresponds to the output of this fragment of I.

which can be seen as taking the intersection of I when translated according to the elements of the reflected B, illustrated in Figure 4.2. By comparing (4.2) with (3.9), it becomes evident that erosion is nothing but a Minkowski subtraction with a reflected SE.

An example of binary erosion using an SE of 3×3 is illustrated in Figure 4.3. Sliding the SE over the image, and shifting the origin over I, the first input pixels resulting in an output equal to ONE with corresponding output are depicted in Figure 4.3(a) (pixels marked as "-" have not been processed). Continuing the scan, the last pixel to produce an output equal to ONE with corresponding output is illustrated in Figure 4.3(b). The final output for this particular example can be envisioned by replacing "-" by zeros in Figure 4.3(b).

4.2.2 **Dilation**

Let $I, B \in Z^2$ represent a binary input image and a structuring element, respectively. Binary dilation is defined as

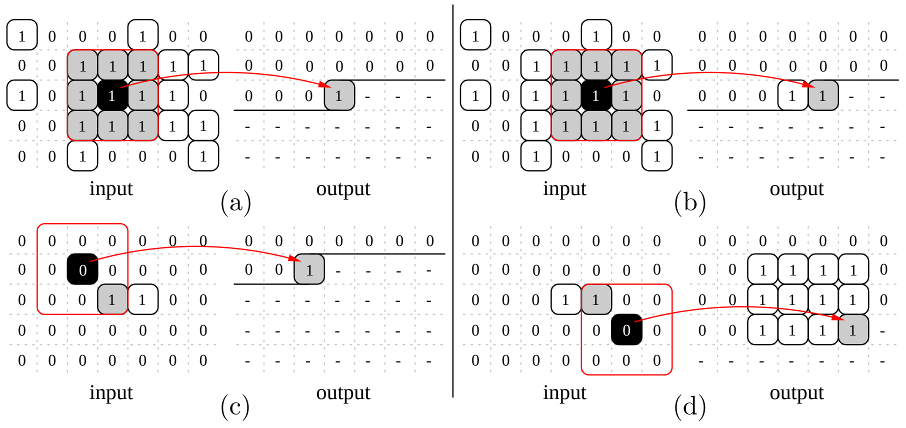


Figure 4.3: (a) - (b) An example of a fragment of the input and output to a binary erosion using a 3 × 3 SE. The origin is marked in black, and foreground pixels covered by the SE are shaded in gray. (c) - (d) An example of a binary dilation dilation using the same settings. Pixels marked as ”-” have not been processed.

$$I \oplus B = \{x \mid (\hat{B})_x \cap I \neq \emptyset\}, \tag{4.3}$$

which means that the dilation of I by B is a set that contains elements x such that the intersection between the I and B when translated by x is not an empty set $\emptyset$. For arbitrary SEs, this means that the output with the same coordinates as the center pixel in the SE becomes ONE if there exists and intersection of at least one pixel of the input image and the SE. This can be compared to performing a logical `OR` operation, or taking the maximum of the pixels in the grid that are currently being covered by a ONE in the SE. As for erosion, an alternative definition for dilation can be written as

$$\delta_B(I) = \bigcup_{b \in B} I_b, \tag{4.4}$$

where the dilation is formed by taking the union of I when translated according to each of the elements in B. By comparing (4.4) with (3.8), it becomes evident that a dilation is equal to a Minkowski addition.

An example of binary dilation using an SE of 3×3 is illustrated in Figure 4.3. The first and last position in the input image resulting in a one is depicted in Figure 4.3(c) and (d), respectively. Completing the scan, the final output for this particular input is can be envisioned by replacing the "-" with zeros in the lower part of Figure 4.3(d).

4.3 Opening and Closing

When combining erosion and dilation, one followed by the other, it is possible to form other important morphological operations, i.e. opening and closing. Opening is an erosion followed by a dilation and closing is a dilation followed by an erosion. An opening of I by B is defined as

$$I \circ B = (I \ominus B) \oplus B,$$

commonly used to filter noise. The filtering is achieved by first eroding the image, resulting in isolated clusters smaller than the SE being removed, followed by a dilation which restores the remaining clusters to their original size. Note that pixels deviating in shape from the SE in the input image are also removed, illustrated in Figure 4.3. An example of a typical noisy image containing three human silhouettes together with the result after an opening has been performed using a flat quadratic SE of size 3×3 containing only ones, are illustrated in Figure 4.4(a) and (b), respectively.

A closing of I by B is defined as

$$I \bullet B = (I \oplus B) \ominus B,$$

which can be used to reconnect split objects. Assuming a flat rectangular SE of size $B_h \times B_w$ with a centered origin, I is first dilated, expanding the clusters in all directions, which results in clusters closer than $B_h - 1$ and $B_w - 1$ pixels apart being merged. The dilation is followed by an erosion, resulting in the clusters being restored to their original size but leaving the connections from the dilation intact.

Combining an opening with a closing can improve the filtered result, since noise may be filtered out and remaining objects can be reconnected. Furthermore, applying a sequence of erosions and dilations, e.g. $(((I \ominus B_1) \oplus B_2) \ominus B_3)$, and allowing the SE size to change between these operations can improve the result even further. As an example, the result after an opening has been applied is shown in Figure 4.4(b), which has many isolated clusters of pixels, especially for the human silhouette in the middle. However, by first applying an erosion with a small SE of size 3×3, noise is removed. Applying a dilation on the results with an elongated SE, reconnects vertically placed objects. Applying a final erosion shrinks the objects, removing extensive pixels that are not part of the objects. The output of applying such a sequence is illustrated in Figure 4.4(c) which is superimposed on the color input frame in (d). Notice how

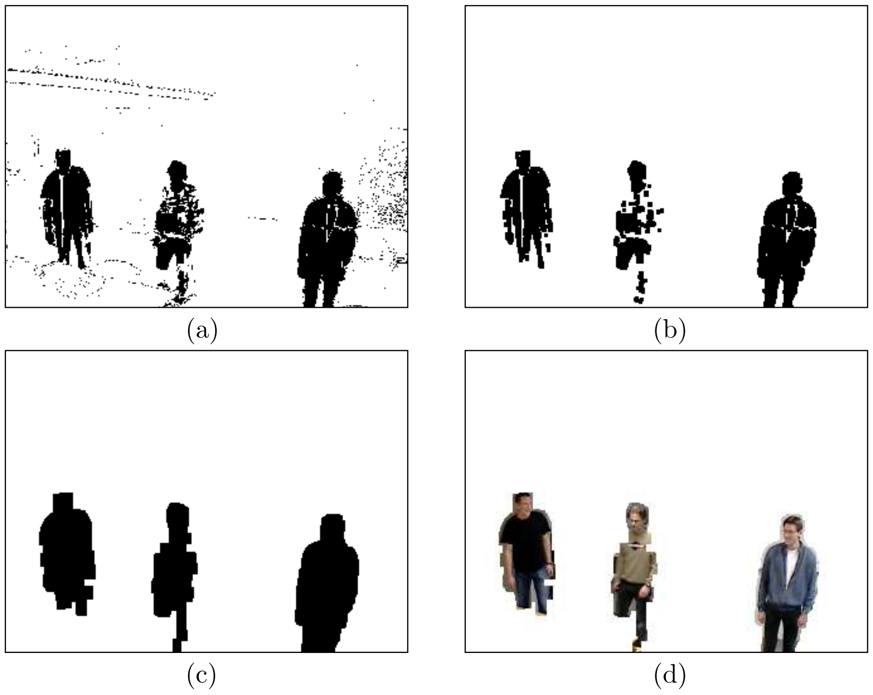


Figure 4.4: (a) A typical binary input, (b) corresponding output after an opening has been performed, (c) corresponding output after an opening followed by a closing, (d) and corresponding output after masking the sensor output with the result obtained in (c).

the number of small isolated clusters are reduced in Figure 4.4 (c) compared to (b).

4.4 Structuring Element Decomposition

Erosion and dilation are associative, which means that if the SE can be decomposed into smaller SEs according to

$$B = B_1 \oplus B_2, \tag{4.5}$$

illustrated for a rectangular SE of size 3×5 in Figure 4.5, then dilating I with B, results in the same output as first dilating I with B_1 and then dilating the result with B_2 according to

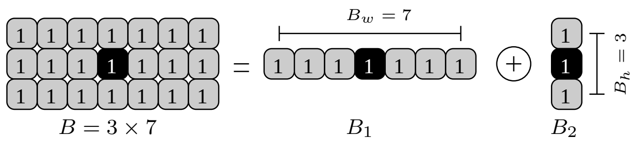


Figure 4.5: Decomposition of a 5×3 structuring element B into B_1 and B_2.

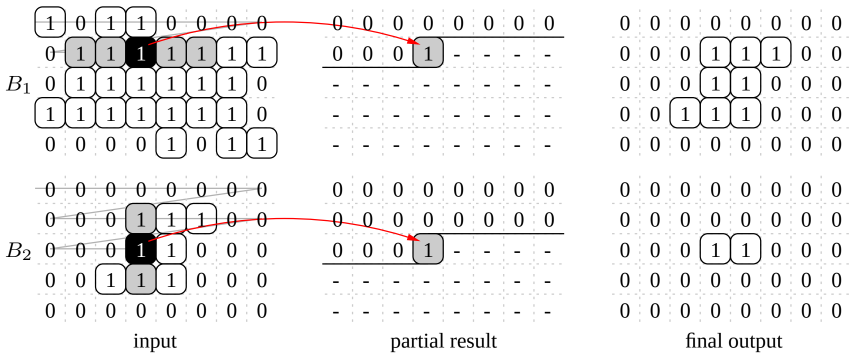


Figure 4.6: Input and output of an erosion using an SE of 5×3 decomposed into $B_1 = 1 \times 5$ and $B_2 = 3 \times 1$. Foreground pixels are shown in white, the origin of the SEs are shown in black, and pixels currently covered by the SEs are shown in gray. Note that the SEs are placed on the first location in the input generating a one in the output, shown as the partial result. Pixels marked as "-" have not been processed.

$$I \oplus B = I \oplus (B_1 \oplus B_2) = (I \oplus B_1) \oplus B_2. \tag{4.6}$$

With a decomposed SE, the number of comparisons per output is decreased from the number of ones in B to the number of ones in B_1 plus B_2. For the example shown in Figure 4.5, the number of bit operations per output is decreased from 15 to 8. An example of an erosion using SE decomposition is illustrated in Figure 4.6. First, the input is eroded with B_1. The output of this erosion is then eroded using B_2 resulting in the final output.

Finding decompositions to an arbitrary-shaped SE is not a trivial problem [45] [46]. However, one common class of SEs that is reflection invariant and easy to decompose, are rectangles of ones. This type of SE is well suited for the opening and closing operations needed in our application, since the noise is uniformly distributed in the input image and allows reconnection of possibly split objects.

4.5 Duality with Respect to Complementation

An important property of both erosion and dilation is that one is the dual of the other according to

$$I \oplus B = (I' \ominus \hat{B})' \tag{4.7}$$

$$I \ominus B = (I' \oplus \hat{B})', \tag{4.8}$$

where $'$ is bit inversion [42]. It is assumed that the height and width of B are odd numbers and that the origin is located in the center. Furthermore, if the SE is both reflection invariant, defined in Section 3.3.1, and decomposable, i.e. $B = \hat{B}$ and $B = B_1 \oplus B_2$, and by combining (4.5), (4.6), (4.7), and (4.8), the following two equations can be derived

$$\begin{aligned} I \oplus B &= (I \oplus B_1) \oplus B_2 = (I' \ominus B_1)' \oplus B_2 \\ &= ((I' \ominus B_1)'' \ominus B_2)' = ((I' \ominus B_1) \ominus B_2)' \end{aligned} \tag{4.9}$$

$$\begin{aligned} I \ominus B &= I \ominus (B_1 \oplus B_2) = (I' \oplus (B_1 \oplus B_2))' \\ &= ((I' \oplus B_1) \oplus B_2)' = ((I'' \ominus B_1)' \oplus B_2)' \\ &= ((I \ominus B_1)'' \ominus B_2)'' = (I \ominus B_1) \ominus B_2, \end{aligned} \tag{4.10}$$

which implies that both erosion and dilation can be expressed as an erosion. This means that the same hardware can be used to perform both erosion and dilation using a decomposed SE, further discussed in Parts I and II.

4.6 Handling the Borders

Sliding window operations are prone to boundary problems. These occur when the B reaches outside of the boundaries of I, as shown in Figure 4.7(a). There are two straightforward methods to address this issue, where one of them is to simply omit these values from the calculation. This strategy can be managed by a controller keeping track of the current position of the SE and thereby masking out pixels outside the borders, i.e. adapting the active region of the SE to the borders. This results in a more complex controller, but may in many cases enable the architecture and data stream to share the same clock domain (inferring only latency). The other strategy is to insert extra pixels outside the image, which is called padding. This can be seen as temporarily increasing the resolution until a well defined output is obtained, as shown in Figure 4.7(b). In case of binary ε and δ, the padding values should not affect the output result and the inferred bits are therefore defined as ONES and ZEROS for ε and δ [42],

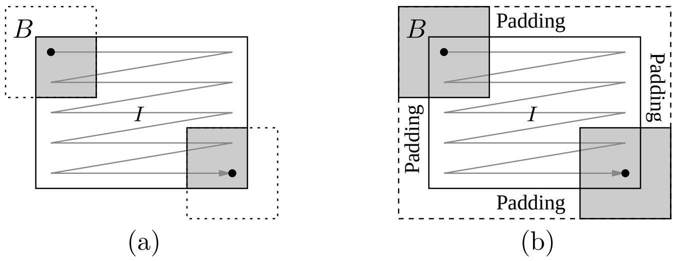


Figure 4.7: (a) An illustration of the boundary problem where B stretches outside of the image borders. Only pixels covered by the shaded region of the SE are included in the calculation. (b) An example of a temporary resolution increase by inserting padding values.

respectively. With these definitions, information around the boundaries of I will not be corrupt, since the output only depends on the image content.

Chapter 5

Labeling

Connected component labeling, or simply labeling, is an operation used to group clusters of pixels with a common denominator in order to separate and distinguish between different clusters in a frame. This is accomplished by assigning a unique label to each cluster. Each label l is taken from a set of discrete values $L \subset \mathbb{N}$, which thereby transforms the frame into a symbolic object mask S, that is the output of the operation. From a set theoretic view, labeling means adding an integer to all elements of a specific set. Let $I \in \mathbb{Z}^2$ be a binary input image with pixels $p \in I$, $F_b = 0$ the set of all background pixels, and $F_f = 1$ the set of all foreground pixels which consists of disjoint clusters C_k, where $1 \leq k \leq K$, for this particular frame. Then without considering relative cluster order, labeling of I can be written as $g : I \mapsto \mathbb{Z}^+$, where $g(i, j)$ is described by

$$g(i,j) = \begin{cases} F_b, & \text{if } I(i,j) = F_b, \\ l_k, & \text{if } p(i,j) \in C_k. \end{cases} \tag{5.1}$$

By this procedure, the labeling process not only counts the number of clusters in each frame, i.e. as the value of K, but also gives the possibility of tying certain features to a specific cluster (label), since each cluster can be referenced by its label. A feature reveals a property of the cluster, e.g. position (coordinates) and size. Thus, labeling can be seen as the link between the clusters and their corresponding features, and is therefore a fundamental operation as a pre- or post processing step in a sequence of algorithms in many applications, e.g. pattern recognition systems and medical image processing applications [47].

5.1 Algorithms

Labeling dates back to the early days of image processing [48] and is considered a fundamental operation [49]. Although the most common case is labeling of a binary image in raster scan order containing 4- or 8-connected clusters mapped on a *cartesian* lattice, the labeling operation is not limited to specific image representations but is dependent on the ability to form sets of connected pixels, i.e. clusters. As an example, in a gray scale image, clusters (sets) can be formed by selecting a specific, or a range of, intensity values which fulfill the connectivity and adjacency criterion; hence, labeling can be applied to this type of image.

Various algorithms have been proposed over the years and the majority can be placed in one of two categories depending on which technique they are based upon, namely

- Sequential Local Operations (SLOs), or
- Contour Tracing (CT).

The first category utilizes label propagation from local neighboring pixels and therefore requires multiple image scans to solve ambiguities and uniform the labels. Algorithms from the second category trace the contour of the clusters, requiring random memory accesses, assigning these pixels a label directly. The remaining cluster pixels are then assigned the correct label when encountered. A sound survey and performance evaluation of algorithms from both categories in terms of execution time versus number of clusters per frame can be found in [50]. Although the two categories differ in the way they assign the labels, a common property for all labeling algorithms is that they have large memory requirements in terms of both size and bandwidth. As a result, complexity in the algorithms can sometimes favorably be traded for memory resources, e.g. a more complex controller requiring less overall memory can still be advantageous.

5.2 Sequential Local Operation Based Algorithms

In SLO-based algorithms, a pixel is labeled based on its value combined with locally propagating labels from previously labeled pixels. Assuming raster scan order, the previously labeled pixels are extracted from a local pixel neighborhood which is determined by superimposing a scan mask M_s, on the labeled result S. A typical forward scan mask M_f is illustrated in Figure 5.1(a) and a corresponding backward scan mask M_b is illustrated in Figure 5.1(b). The forward scan mask extracts pixels above and to the left of the current pixel in the labeled output, which comes naturally when streaming data in raster scan

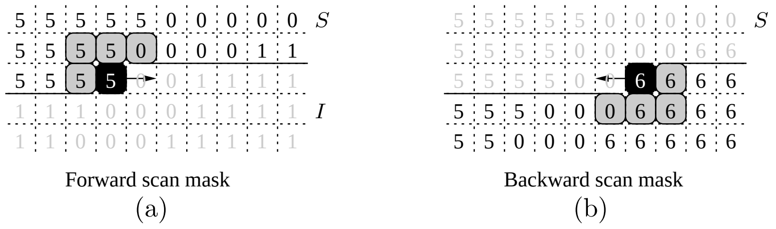


Figure 5.1: (a) A typical forward scan mask M_f used in raster scan order, where the location of the pixel to be labeled is marked in black and the pixels currently covered by the scan mask are shown in gray. In this case, the pixel is assigned the label 5, based on evaluating the labels currently covered by the scan mask. (b) A typical backward scan mask M_b used in anti-raster scan order, where the pixel is assigned the label 6. Labeled pixels in S are shown in black, while unlabeled pixels in I are shown in gray. The arrows indicate the direction of masks.

order. However, due to the nature of the label propagation, all SLO-based labeling algorithms have to address the same obstacle: label collisions. In a binary image, a typical label collision occurs if a u shaped object is encountered, illustrated in Figure 5.2(a). Scanning the image, the first encountered pixel from this cluster is p_1, which is assigned the label l_1. Proceeding the scan, when reaching p_2 in Figure 5.2(b), a new label l_2 will be assigned to this pixel since there is no momentary information that p_1 and p_2 are part of the same cluster, hence that they should have the same label. Reaching pixel p_3 in Figure 5.2(c), a label collision occurs since there is an ambiguity in regard to which label to assign to this pixel (l_1 or l_2). How to address these ambiguities is further discussed in Section 5.3. Naturally, the number of label collisions per cluster depends on the complexity of the contour. A synthetic example of a cluster with multiple label collisions occupying several labels is illustrated in Figure 5.3. The impact that these collisions have on certain hardware resources, e.g. the memory requirement, is addressed in detail in Section 2.2.

5.3 Resolving Label Collisions

In its simplest form, SLO-based algorithms handle label collisions by multiple scans of the image in raster- and anti-raster until the labels reach stability [51]. A first forward scan can be completed on the fly as the incoming frame I is written into memory, assigning every cluster pixel a preliminary label according to

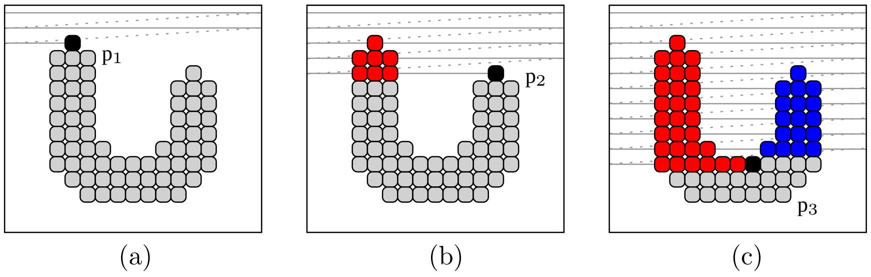


Figure 5.2: An example of a typical label collision during the initial scan, (a) the first upper and leftmost pixel of a cluster is reached p_1, and labeled with l_1. (b) Continuing the scan, another part of the same cluster is reached p_2, and is labeled with l_2. (c) Reaching p_3, a label collision will occur since pixels in the same row are labeled using l_1 and pixels above right are labeled with l_2.

$$g(i,j) = \begin{cases} F_b & \text{if } I(i,j) = F_b, \\ l_k, (k = k+1) & \text{if } g(i+s, j+t) = F_b \ \forall (s,t) \in M_f, \\ g_{\min}(i,j) & \text{otherwise,} \end{cases} \tag{5.2}$$

where

$$g_{\min}(i,j) = \min[\{g(i+s, j+t) | (s,t) \in M_s\}], \tag{5.3}$$

and k is initialized to 1, and $(k = k+1)$ indicates the increment of k and $M_s = M_f$. This initial scan is followed by multiple scans of the labeled image back and forth in raster and anti-raster scan order to propagate labels in all directions, alternating the scan mask in (5.3), i.e. $M_s = M_f$ and $M_s = M_b$ during the forward and backward scan, respectively. During these succeeding scans, the pixels are labeled according to

$$g(i,j) = \begin{cases} F_b, & \text{if } g(i,j) = F_b, \\ g_{\min}(i,j), & \text{otherwise,} \end{cases} \tag{5.4}$$

until all possible label ambiguities have been resolved for every pixel cluster, i.e. until S is stable. Although having a regular scanning pattern which allows burst reads from memory, since the number of image scans depends on the complexity of the clusters, this type of algorithm is not suitable in real-time processing systems.

Using the same procedure but adding an equivalence table, to administrate label collisions, the number of required image scans is reduced to two. As for the

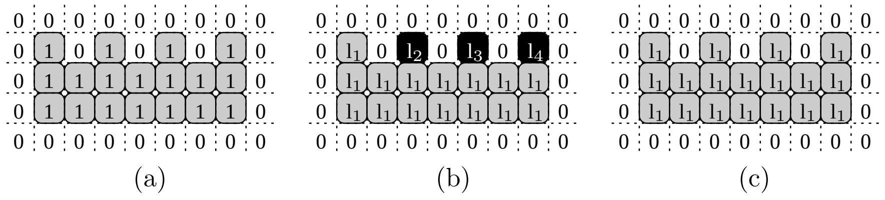


Figure 5.3: (a) An example of a cluster containing multiple label collisions, (b) with corresponding labels after the initial scan, where the label collisions are marked in black. (c) The final label result after the second image scan, where the label collisions have been resolved.

simpler version of the algorithm, the first image scan can be completed on the fly as the incoming frame is written into the label memory. However, during this initial scan, as the preliminary labeled image is written into the label memory, possible label ambiguities are simultaneously written into the equivalence table. Using the information in this table, label ambiguities can be resolved during a second scan assigning the final label to the clusters, e.g. replaced by the equivalent label with the lowest number. This approach is often referred to as the two-scan process, and will function as reference design in the sense that competing algorithms must achieve better or have major advantages. Many algorithms are based on this approach but with improved data administration [48] [52] [53] [54].

Assuming that the data is stored, or is going to be written into a memory, the second scan can, under certain circumstances, be omitted. By placing control circuitry at the output to administrate the label collisions on the fly during read-out of the labeled data for further processing, a single scan together with the equivalence table is enough to label an image. However, although this labeling procedure is highly application dependent, if throughput is the main constraint and the post-processing this technique requires is applicable, this is an interesting alternative.

5.4 Contour Tracing Based Algorithms

As the name implies, CT-based algorithms is a technique that traces the contour of each cluster assigning these pixels a label. The approach was developed by Chang *et al.* [55] [56]. Assuming that the image to be labeled is stored in a memory, then by tracing the contour of the clusters immediately when encountered during a global raster scan, label collisions are avoided. Continuing the scan, intermediate pixels, i.e. pixels in between two contour pixels with the same label, are regarded as part of the same cluster and therefore assigned the

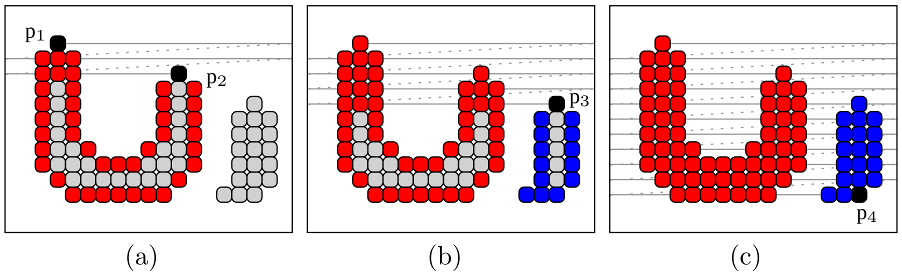


Figure 5.4: (a) The scan starts at the previously marked start point p_1, i.e. the first cluster pixel equal to 1, and the contour of this cluster is traced and labeled. Continuing the scan, when reaching p_2, no label collision can occur since this pixel has been previously labeled. (b) Another cluster is encountered, which is contour traced and labeled. (c) The labeling is complete when reaching p_4, i.e. the previously marked end point, and the image is now the final labeled output.

same label.

In its original form, the algorithm labels the exterior contours together with the intermediate pixels with the same label. The CT phase of an exterior contour is initialized if the algorithm encounters an unlabeled foreground pixel $p(i, j) = F_f$, and the pixel directly to the left or above is part of the background, i.e. $p_(i, j - 1) = F_b$ or $p_(i - 1, j) = F_b$. The intermediate pixels are labeled during the ongoing global scan, i.e. if an unlabeled pixel is encountered and the criterion to start either CT phase is not fulfilled. When a cluster with a previously labeled contour is encountered, the scan proceeds without modification. As an example, in Figure 5.4(a), no label collision will occur since when reaching p_2, this pixel has already been labeled during the prior contour tracing and the scan proceeds without modification. If another unlabeled cluster is reached, the contour tracing procedure restarts, illustrated in Figure 5.4(b). When the last pixel is reached, the labeling of this particular frame is complete and the final labeled output is illustrated in Figure 5.4(c).

In detail, the algorithm requires two global memory scans together with additional random accesses for the CT procedure in order to label a frame. The algorithm starts with writing the incoming frame into the label memory marking the first and last pixel equal to ONE as global start and end point, respectively, corresponding to p_1 and p_4 in Figure 5.4. Assuming that the input image is not empty, a second memory scan starts from the previously marked global start pixel p_1. This pixel is now marked as the local start pixel

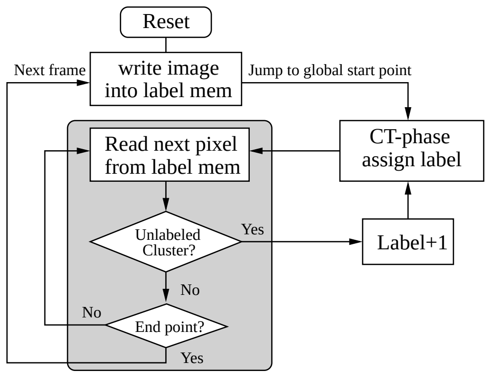

Reset
write image into label mem
Next frame
Jump to global start point
Read next pixel from label mem
CT-phase assign label
Unlabeled Cluster?
Yes
Label+1
No
No
End point?
Yes

Figure 5.5: A simplified block diagram of a CT-based labeling algorithm.

and the contour of this cluster is traced, writing the label at every contour pixel into the label memory. A local start point corresponds to the first pixel of an unlabeled contour. The CT phase of this cluster is completed when the local start pixel is reached a second time, and when this occurs, the label is increased by 1 and the global scan continues in raster scan order. Continuing the memory scan, one out of several pixel values for $p_{i,j}$ will be encountered:

- $p_{i,j} = 0$ - a background pixel. The scan continues without modification.
- $p_{i,j} = 1$ - an unlabeled foreground pixel. The contour of this clusters is traced and labeled.
- $p_{i,j} > 1$ - a previously labeled pixel. The global scan continues without modification since intermediate pixels are regarded as part of the same cluster.
- A border pixel ($i = I_h - 1$ or $j = I_w - 1$). If a border pixel on the right side is reached, the scan continues on the left side on the next scan line.
- The previously marked end pixel. If the marked end pixel is reached, p_3 in Figure 5.4(b), the labeling of this particular frame is completed and the algorithm can proceed with the next frame.

A block diagram of the algorithm with the described steps is depicted in Figure 5.5. Note that starting a second scan in the start and end pixel is an optimization that is valid for SLO-based algorithms as well.

A cluster has one exterior contour but can also have multiple interior contours around holes, e.g. as the cluster illustrated in Figure 5.6(a). The original

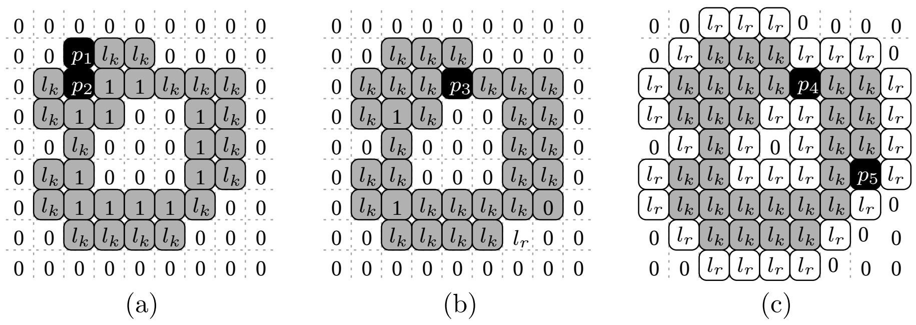
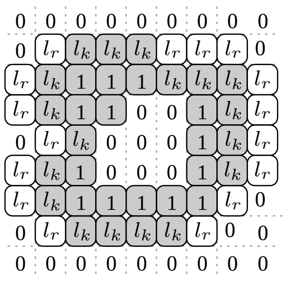


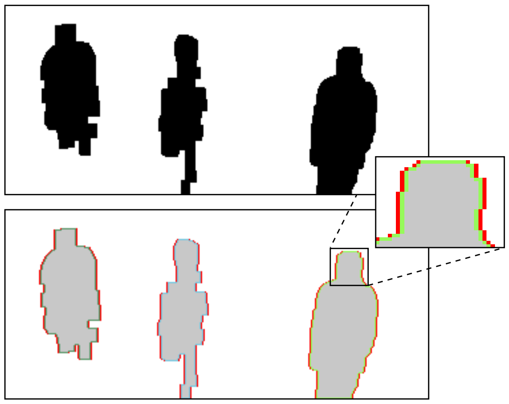
Figure 5.6: (a) An example of an arbitrary non-solid cluster of which the exterior contour is traced and assigned the label l_k when p_1 is reached the first time. Continuing the scan, when reaching p_2, which represents unlabeled intermediate pixels, these pixels are assigned the same label as the one to the left, i.e. l_k. (b) When reaching p_3, the interior contour is traced and assigned the same label. (c) To avoid restarting the interior CT phase when reaching certain pixels, e.g. p_4 and p_5, a reserved label l_r is written on the outside of the exterior and on the inside of the interior contour, respectively. This is the final labeled result of this particular cluster using the original CT algorithm.

algorithm labels both the exterior and interior contours together with assigning intermediate pixels with the same label. The CT phase of an interior contour is initialized if the algorithm encounters and unlabeled foreground pixel and the pixel directly below it is part of the background, e.g. when reaching p_3 in Figure 5.6(b). To avoid pitfalls like tracing the same contour multiple times, as would be the case when reaching p_4 in Figure 5.6(c), Chang *et al.* proposed that a reserved label l_r, should be written on both the outer and inner side of the exterior and interior contours, respectively, as illustrated in Figure 5.6(c). By this procedure, when the algorithm encounters an unlabeled pixel, the value of the pixel above and below this pixel is checked to determine if the current pixel is part of a labeled contour, or not. If l_r is found, this contour has been previously labeled and the scan continues without modification.

If the read-out pattern is also performed in raster scan order, both tracing interior contours and writing l_r on its inside becomes redundant, which only results in excessive memory accesses. The excessive memory can be removed by adding some simple modifications to the original algorithm. The CT phase of an exterior contour is initialized as before, i.e. whenever the condition $p(i,j) = F_f$ and $p_(i,j-1)$ is fulfilled. However, due to the nature of the raster scan order,


0 0 0 0 0 0 0 0 0
0 l_r l_k l_k l_k l_r l_r l_r 0
l_r l_k 1 1 1 l_k l_k l_k l_r
l_r l_k 1 1 0 0 1 l_k l_r
0 l_r l_k 0 0 0 1 l_k l_r
l_r l_k 1 0 0 0 1 l_k l_r
l_r l_k 1 1 1 1 1 l_r 0
0 l_r l_k l_k l_k l_k l_r 0 0
0 0 0 0 0 0 0 0 0

(a)



(b)

Figure 5.7: (a) An example of the output from the modified CT-algorithm when applied on the input from Figure 5.6. (b) An example of the input and output from the modified CT-labeling algorithm. Each contour is labeled with a unique color and intermediate pixels are shown in gray. Notice the reserved label written on each side of the contour.

a cluster will always be entered from the left and exited to the right. This results in l_r not being required to be written on all sides of the contour so that it completely encapsulates the cluster, but rather only on the left and right side of the cluster to avoid falsely restarting the CT phase. Furthermore, by adding a boolean variable V_{bool}, the algorithm can keep track of whether the global scan is currently inside a cluster or not, which makes the tracing of all interior contours superfluous. Hence, having the exterior contour traced and assigned the label l_k and l_r written on both sides of a cluster C_k, as illustrated in Figure 5.7(a), entering a slice of n pixels of C_k on a specific row must occur in the following order:

1. $p_0 = l_r$ – The first reserved label (if the contour is not aligned to the left border of the frame).

2. $p_1 = l_k$ – The first contour pixel. The inside cluster flag is raised $V_{bool} = 1$.

3. Intermediate pixels which can be either contour pixels $p = l_k$, cluster pixels $p = 1$, or holes $p = 0$.

4. $p_{n-2} = l_k$ – The second contour pixel on the other side of the slice.

5. $p_{n-1} = l_r$ – The second encountered reserved label (if the contour is not aligned to the right border of the frame). The inside cluster flag is lowered $V_{bool} = 0$.

Based on the status of V_{bool}, intermediate pixels can be assigned the correct label during the global scan. This means that intermediate pixels equal to ONE are assigned the same label as the neighboring pixel to the left, while pixels equal to 0 (hole) are either labeled or left unchanged. This procedure offers the possibility of filling holes inside clusters which can be useful in certain applications, e.g. in automated surveillance when detecting solid objects (humans). However, if labeled intermediate pixels are not required by the application, assigning these pixels the correct label can be omitted or administrated by a control unit located at the output during read-out, which would reduce the number of write operations even further. If the contour of the cluster is aligned with the boundaries of the frame, no reserved label will be encountered and this results in some exceptions in the control logic. The scan continues row by row until the end pixel is reached and the algorithm can start over with labeling of the next frame. An example of the output from the algorithm after having added all these modification is illustrated in 5.7(b).

Part I

Low-Complexity Binary Morphology with Flat Rectangular Structuring Elements

Abstract

This part describes and evaluates hardware architectures for binary morphological erosion and dilation. In particular, a fast, stall-free, low-complexity architecture is proposed, one that takes advantage of the morphological duality principle and of structuring element decomposition. The design is intended to be used as a hardware accelerator in real-time embedded processing applications. Hence, the aim is to minimize the number of operations, memory requirement, and memory accesses per pixel. The main advantage of the proposed architecture is that for the common class of flat and rectangular structuring elements, the complexity and number of memory accesses per pixel is low and independent of both the image and structuring element size. The proposed design is compared to the more common delay-line architecture in terms of complexity, memory requirements and execution time, both for an actual implementation and as a function of image resolution and structuring element size. The architecture is synthesized for the UMC 0.13 μm CMOS process using a resolution of 640×480. A maximum structuring element of 63×63 is supported at an estimated clock frequency of 333 MHz.

Based on: H. Hedberg, F. Kristensen, and V. Öwall "Low Complexity Binary Morphology With Flat Rectangular Structuring Elements," accepted for publication in *IEEE Transactions on Circuits and Systems I.* **and**
H. Hedberg, F. Kristensen, P. Nilsson, and V. Öwall"A Low Complexity Architecture for Binary Image Erosion and Dilation using Structuring Element Decomposition," in *Proc. of IEEE International Symposium on Circuits and Systems*, Japan, 2005.

1 Introduction

When designing a morphological hardware unit, there are many application-specific questions and issues that need to be addressed, e.g. the required class of supported SEs as well as issues related to the imposed bandwidth. Depending on the answers, certain trade-offs can be made in the architecture. However, there are some properties, apart from the obvious ones such as low complexity and fast execution time, that are advantageous and should be taken into consideration under any circumstance. First, most images are acquired and stored in raster scan order, as discussed in Section 3.1.4. Therefore, to easily incorporate the units into this type of system environment while simultaneously avoiding intermediate data storage and random memory accesses during pixel processing, the architecture should use the input and produce the output in raster scan order. This permits burst reads from memory and the possibility of placing several units sequentially without intermediate storage. Secondly, extracting and performing calculations on large neighborhoods can become particularly computationally intensive. Therefore, the main obstacle when designing a morphological hardware architecture is to extract this neighborhood and perform the calculation with minimal hardware resource utilization while still preserving raster scan order. To increase the flexibility and thereby the applicability of the architecture, it is desirable that the size of the B can be changed during run-time. As an example, in the automated surveillance application outlined in [57], a flexible B size can be utilized to compensate for different types of noise and to sort out certain types of objects in the mask, e.g. high and thin objects (standing humans) or wide and low objects (the side view of cars).

1.1 Previous Research

Mathematical morphology is and has been a subject of extensive study resulting in numerous books and articles, covering both the theoretical and hardware aspects of this field. However, to gain some perspective on this work, only other important hardware architectures performing binary erosion and dilation are discussed here.

Fejes and Vajda [37] and Velten and Kummert [58] propose a delay-line architecture for 2-D binary erosion or dilation. This classical approach supports arbitrary-shaped Bs, but the hardware complexity is proportional to ${N_B}^2$, where $N_B = \max(B_h) = \max(B_w)$, defined in Section 4.1. The pixels that are to be reused are stored in delay-lines, resulting in a memory requirement proportional to $N_B N$, where N is the image width. Therefore, this type of implementation becomes unsuitable for large SEs and high resolution applications. In [59], an architecture using the same type of delay-lines is proposed, thus having the same memory requirement. However, based on the observation that many calculations between two adjacent pixels are redundant and that partial results can be reused, the architecture is given its name, Partial-Result-Reuse (PRR). With this procedure, the number of comparators per output value can be reduced to $2\lceil\log_2(N_B)\rceil$ for certain B shapes, e.g. rectangles.

Źarandy *et al.* propose a cellular neural network approach to perform binary erosion and dilation in [60]. It is shown that binary morphology applies to this type of structure but, as for the delay-line architecture, the computational complexity is proportional to ${N_B}^2$. This is due to the fact that the SE is mapped onto the array of cells, where each element requires a separate cell. In addition, since the pixels needs to be reused, they need to be stored, which is preferably in delay-lines, once more resulting in with a memory requirement proportional to $N_B N$. However, this approach opens up other possibilities, with regard to the learning feature of such networks to control the size and shape of the SE; this is not further addressed in this thesis.

Malamas *et al.* present a fast systolic architecture performing a 1-D binary erosion or dilation (or a combination of these) in [61]. The architecture can be extended to 2-D by parallel processing of the 1-D units. Processing each row in parallel makes it faster, but the drawback of this architecture is that the complexity of each 1-D branch is proportional to the SE width, making it unsuitable for applications requiring large SEs.

In [62], a Low Complexity (LC) and low memory requirement architecture performing a 2-D binary erosion or dilation that takes advantage of the morphological duality and SE decomposition is proposed. The main characteristic of this architecture is that it has constant computational complexity, i.e. each output value is calculated with only four operations per pixel independent of the B size, and a memory requirement proportional to $N\log_2(N_B)$. The class of supported Bs is limited to flat rectangles of arbitrary size. The main drawback of this architecture is that the input stream has to be stalled during padding, which requires additional memory in the form of a First In, First Out (FIFO) located at the input. However, through parallel processing, the stall cycles during padding can be avoided, so that no additional memory is required. This modification results in an architecture that uses a single clock domain and is superior both in terms of computational complexity and memory requirement, hereafter referred to as stall-free architecture.

2 Architecture

2.1 Delay-line Architecture

The delay-line architecture is a direct mapping of the ε or δ operation [37] [63]. The main idea is to store pixels to the left and right of the B as long as they

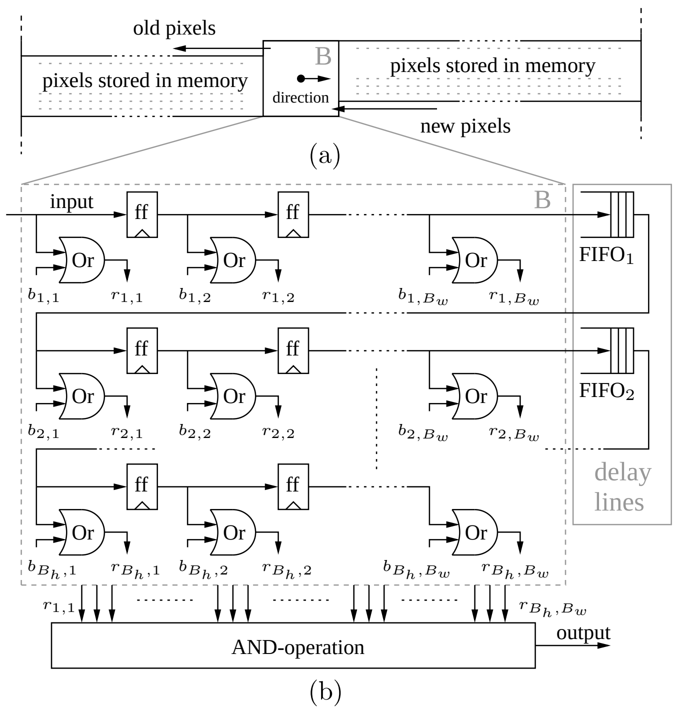

old pixels
pixels stored in memory
B
direction
pixels stored in memory
new pixels
(a)
input
ff
Or
B
FIFO$_1$
FIFO$_2$
delay lines
$b_{1,1}$ $r_{1,1}$ $b_{1,2}$ $r_{1,2}$ b_{1,B_w} r_{1,B_w}
$b_{2,1}$ $r_{2,1}$ $b_{2,2}$ $r_{2,2}$ b_{2,B_w} r_{2,B_w}
$b_{B_h,1}$ $r_{B_h,1}$ $b_{B_h,2}$ $r_{B_h,2}$ b_{B_h,B_w} r_{B_h,B_w}
$r_{1,1}$ r_{B_h,B_w}
AND-operation
output
(b)

Figure 1: (a) An illustration of the incoming pixel stream and pixels stored in memory in a direct mapped implementation.(b) A delay-line architecture of a morphological erosion block. The grey dash indicates the pixels covered by B. B_h and B_w is the height and width of B. Control logic is omitted for clarity.

are to be used in future output calculations, i.e. all consecutive pixels from the upper-left corner to the lower-right corner of the SE are stored in one long memory chain, as shown in Figure 1(a). As an incoming pixel is shifted in, the oldest pixel currently in the architecture is shifted out in raster scan order, discussed in Section 3.1.4. An example of an architecture that implements this functionality for erosion is shown in Figure 1(b), and is used as a reference throughout the thesis. All pixels covered by B are stored in flip-flops and are thus individually accessible; all pixels in between two rows of B are stored in FIFOs (First In First Out). This allows B to be moved to the next position by reading a new input and moving all other pixels one step forward in the memory chain, achieved either by shifting data or changing memory pointers.

The major benefit of this architecture is its ability to support streaming input data and arbitrary-shaped Bs. Control logic to manage the enable signals for each element in the structuring element $b_{i,j}$ and to change the morphological operation to dilation is omitted in Figure 1(b).

In addition, to handle the boundary issue discussed in Section 4.6, the ability to control each position in B is used. The parts of B that extend outside the image are forced to ONE in accordance with the definition, see Section 4.6. In the architecture in Figure 1(b), the control signals $b_{i,j}$ are set to ONE for the parts of B that extend beyond the image border.

2.2 Low Complexity Architecture

A common class of Bs that is both decomposable and reflection invariant is flat rectangles [64], which are well-suited to perform operations such as the opening or closing required in the real-time application described in Part V. With a decomposed B, the number of comparisons per output pixel is decreased from the number of ones in B to the number of ones in B_1 plus B_2, as discussed in 4.4. The input is first eroded by B_1 and then by B_2, or the other way around.

When using flat rectangular Bs containing only ones, ε can be performed as a summation followed by a comparison. The summation consists of keeping track of the number of bits in I that are currently covered by B; this number is compared to the number of ones in B. If they are equal, output a ONE, otherwise output a ZERO. In decomposing B, the summation can be broken up into two stages. The first stage compares the number of consecutive ONES in I to the width of B_1. The second stage sums the result from the first stage for each column and compares it to the height of B_2. If both these conditions are fulfilled, the output at the coordinate of the SE origin is set to ONE, ZERO otherwise.

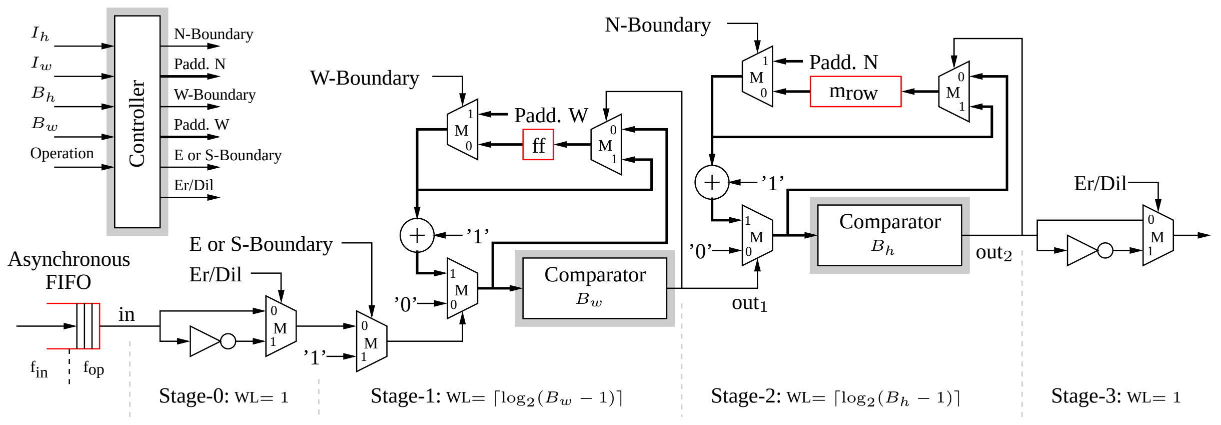


Figure 2: Architecture of the erosion and dilation unit. Multiplexors are marked with M, the flip-flop with ff and the row memory with m_{row}. I_h and I_w corresponds to the height and width of the input image and thick lines indicate buses with the corresponding WL shown in each stage.

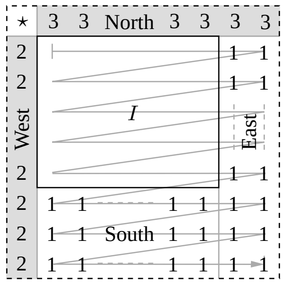
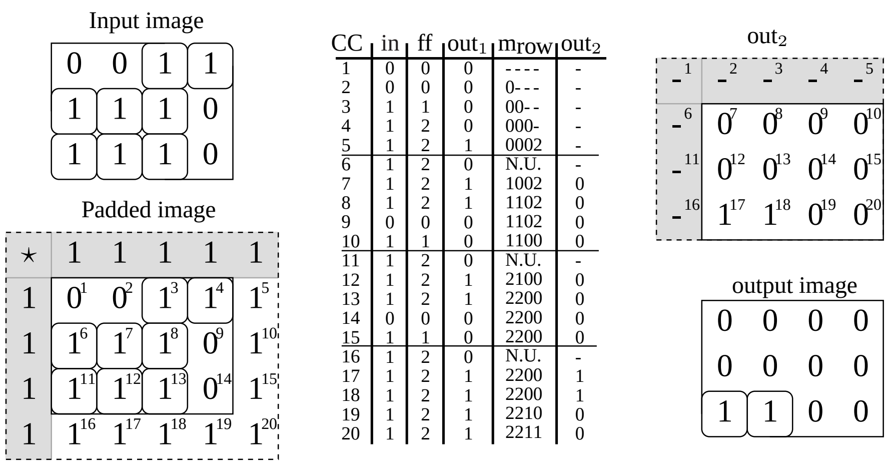


Figure 3: An example of padding values when $B = 7 \times 5$. The $\star$ marks a *don't care* position. Padding to the east and south is inserted in the data stream which is shown by the gray arrow. The west and north padding are not part of the data stream and is only used as initial values for the memory.

The proposed architecture is based on the observations above and is shown in Figure 2 with the corresponding Word-Length (WL) in each stage. Taking advantage of the duality property, discussed in Section 4.5, the same inner kernel is used for both δ and ε; to perform dilation on an erosion-unit, simply invert the input I and the output. This function is performed in Stage-0 and stage-3.

To handle the boundary, the padding is split into four parts: north, east, south, and west, illustrated in Figure 3. Assuming a centered origin, the east and west padding extend $\lfloor B_w/2 \rfloor$ columns and the north and south padding extend $\lfloor B_h/2 \rfloor$ rows outside I, where B_h and B_w correspond to the height and width of B, respectively. Since δ is transformed into ε by inverting the input (I'), the padding will be ONE regardless of the executed operation. In the proposed architecture, the padding to the west and north are pre-calculated values and inserted as the initial values of the sums in stage-1 and stage-2, respectively. The east and south padding are handled differently since the padding has to be inserted as extra pixels in the data stream, the east padding in between the rows of I and the south padding after the last pixel has been processed from I. Figure 3 shows an example of both the pre-calculated padding and the inserted extra bits for each corresponding side of I when B consists of seven rows and five columns of ONES.

Using this architecture, each pixel in I is used once to update the sum stored in the flip-flop in stage-1, that records the number of consecutive ONES to the left of the currently processed pixel. When the input is ONE, the sum is increased, otherwise it is reset to ZERO. Each time the sum plus the input equals the width of B_1, stage-1 outputs ONE to stage-2 and the previous sum



CC	in	ff	out$_1$	m$_{row}$	out$_2$
1	0	0	0	----	-
2	0	0	0	0---	-
3	1	1	0	00--	-
4	1	2	0	000-	-
5	1	2	1	0002	-
6	1	2	0	N.U.	-
7	1	2	1	1002	0
8	1	2	1	1102	0
9	0	0	0	1102	0
10	1	1	0	1100	0
11	1	2	0	N.U.	-
12	1	2	1	2100	0
13	1	2	1	2200	0
14	0	0	0	2200	0
15	1	1	0	2200	0
16	1	2	0	N.U.	-
17	1	2	1	2200	1
18	1	2	1	2200	1
19	1	2	1	2210	0
20	1	2	1	2211	0

Figure 4: A Clock Cycle (CC) true example of an erosion using a 3×3 SE. ff shows the content of the flip-flop in stage-1, m$_{\text{row}}$ the row memory in stage-2, and out$_1$ and out$_2$ show the output from each respective stage. *N.U.* indicates that the row memory is not updated in that clock cycle; '-' represents invalid data.

is kept. The same principle is used in stage-2, but instead of a flip-flop, a row memory is used to store the number of ONES from stage-1 in the vertical direction for each column in I. In addition, a controller is required to handle the padding and to determine the operation to be performed, i.e. ε or δ. An example of the values in the main blocks in the architecture after each clock cycle when performing an erosion is shown in Figure 4. The input image is padded in the same manner, shown in Figure 4.7(b) and all signals can be found in Figure 2. Since an erosion is performed, stage-0 and stage-3 are only bypassing the input and output signals.

The input and output of this architecture is binary; hence the WL in stage-0 and stage-3 only has to be one bit. However, in stage-1 and stage-2 sums are recorded and the WL has to be wide enough to hold the maximum values. In stage-1 the maximum sum is equal to $B_w - 1$ and the corresponding WL to $\lceil \log_2(B_w - 1) \rceil$. In stage-2 the maximum sum depends on the height of B and WL= $\lceil \log_2(B_h - 1) \rceil$.

When padding is inserted in the data stream, this means that the input data have to be stalled for the duration of the padding. The effect of this is twofold: additional memory is required and the operating frequency of the data-path has to be higher than the input frequency. Hence, an asynchronous FIFO,

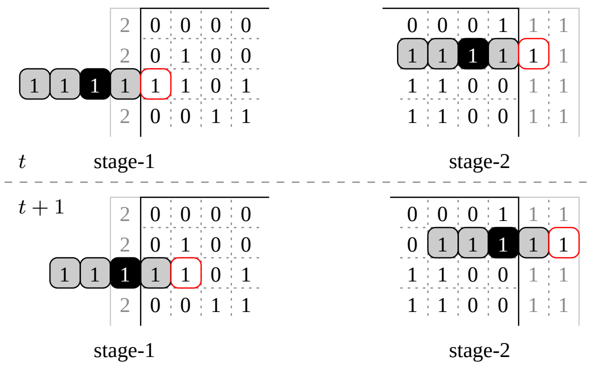


Figure 5: An example of parallel processing of the padding in stage-1 and stage-2 at time t and $t+1$ if the B is of size 1×5. The two east padding pixels from the previous row are processed in parallel with the first two pixels in the current row.

located at the input in Figure 2, is needed to store input data and separate the two different clock domains.

2.3 Stall-Free Low Complexity Architecture

In order to improve the memory requirements, an extension of the architecture described in Section 2.2 is proposed. The result is an architecture that shares the same principles and explores the same morphological properties, achieving the same computational complexity. However, the major difference lies in how the padding is addressed. Adding hardware support for processing padding in parallel instead of serially, e.g. the east and west padding, omits the need to stall the input. Hence, no FIFO is required at the input and the memory requirements are reduced even further.

Assuming that the input is streaming back-to-back images, two cases of independent consecutive pixels can be recognized: the transition from one row to the next and the transition from one image to the next. In the case of the row-to-row transition, the last pixels in a row are predefined padding pixels and the first pixels in the next row can only increase the stored sum in stage-1; they do not produce an output to stage-2. Hence, a modified version of stage-1, which only handles the padding, can be added to process the last pixels in a row corresponding to the east padding, thus freeing the regular stage-1 to start processing the first input pixels of the next row concurrently. The procedure is illustrated in Figure 5, which shows an example of which padding pixels are processed in parallel during a transition between two consecutive rows in a

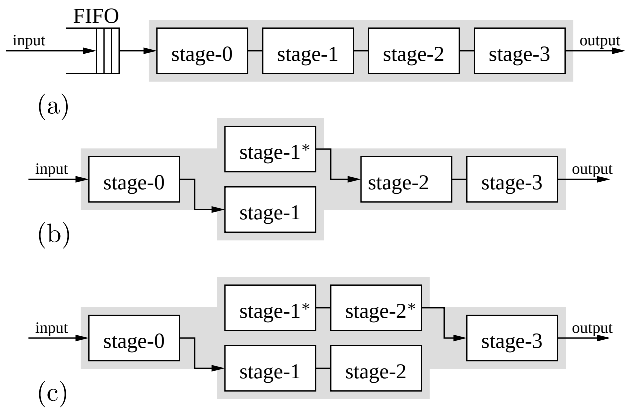


Figure 6: (a) A block diagram of the low complexity algorithm. (b)–(c) Block diagrams of the dataflow during a row-break and an image-break, respectively, in the stall-free algorithm. * indicates modified blocks.

frame at time t and $t + 1$. The block diagram of the regular low complexity and the stall-free architecture during a row-break is shown Figures 6(a) and (b), respectively. In the image-to-image case, the same principle can be used, since the last pixels in an image are the south padding and the first pixels of the next image cannot produce an output from stage-2. The only difference is that a second stage-2 is added and the dataflow is as shown in Figure 6(c).

A continuous data stream constituted of back-to-back images is the worst-case scenario. This type of input pattern characteristic can be found when the input source is burst read from memory, e.g. when processing images from a video sequence. However, when using a sensor in the image acquisition step in a real-time environment, the timing model can be relaxed somewhat. This is because for most sensors, there are typically tens of extra cycles in between the rows and images in the sensor output pattern. These extra cycles can be utilized to perform the east padding found in between rows and thereby reduce the size requirement of the input FIFO needed in the low complexity architecture. However, the number of extra cycles in the output pattern will, in most cases, not exceed the number of the required stall cycles during the south padding, which is mainly proportional to $\lfloor B_h/2 \rfloor \cdot I_w$, where I_w is the width of the input image. Therefore, in applications where the input source is a stream of multiple images requiring an $B_h > 1$, the FIFO is still needed in the low complexity architecture, making the stall-free architecture superior to the others in terms of memory requirements.

With these modifications, streaming back-to-back images can be processed without stalling input data. Even though the amount of hardware is increased inside the data-path, Section 4 shows that this amount is far less than the FIFO requirement. Another benefit derived from this property is that only one clock domain is required, i.e. the architecture can run at the same speed as the incoming data, which facilitates incorporating the unit in an embedded system environment.

2.4 Extended Morphological Operations

Due to its low complexity, the stall-free architecture allows several units to be connected to form extended morphological operations, which increases the flexibility and thereby the applicability of the architecture. As an example, contour extraction is performed by subtracting $I - \varepsilon(I, B = 3 \times 3)$, which is accomplished with an adder and a FIFO to compensate for the latency imposed by the architecture [30]. A boundary extraction unit using the proposed architecture together with examples of input and output is shown in Figure 7(a).

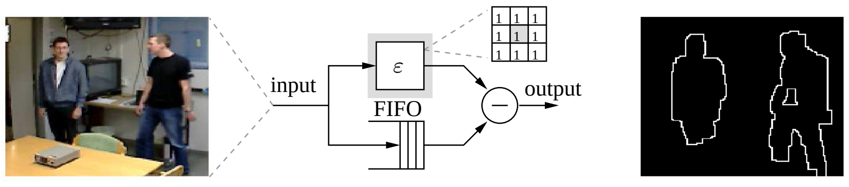
Granulometry based on parallel openings is a morphological operation which is used to estimate cluster sizes in images [44]. This is an example of an advanced operation in which the benefits of the proposed architecture are substantial both in terms of speed and memory requirements. The operation is based on the difference between the remaining number of pixels after parallel openings with an increasing B size, i.e. a square with a side $N_B \in \{1, 3, 5, \ldots\}$. Let A_i be the sum of the remaining pixels p equal to ONE after the ith opening, calculated as

$$A_i = \sum_{\forall p=1} I \circ B_i,$$

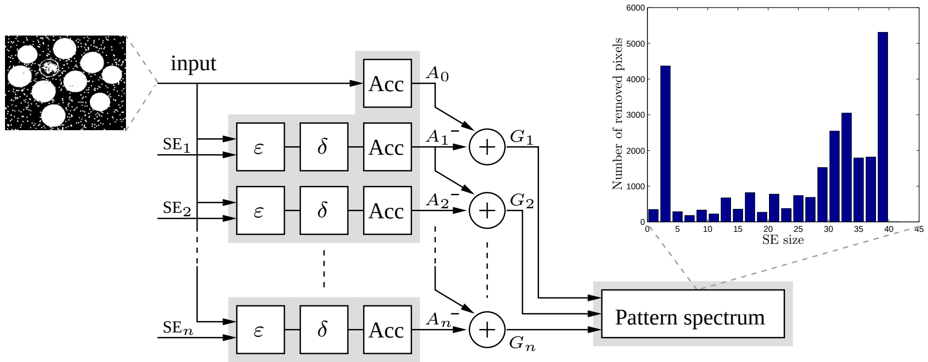
referred to as the size distribution. The difference between adjacent size distributions is defined as the granulometric function and is calculated according to

$$G_i = A_{i-1} - A_i, \text{ where } i = \{1, 2, .., n\},$$

which is also referred to as the pattern spectrum. The granulometric function is sensitive to changes in the number of removed pixels, which means that an impulse in the pattern spectrum at a certain B size indicates that this is a typical cluster size. The hardware unit used to calculate the pattern spectrum based on the proposed architecture is illustrated in Figure 7(b), where Acc are accumulators that sums and store the number of remaining foreground pixels.



(a)



(b)

Figure 7: Examples of extended morphological operations based on the low complexity architecture. (a) A boundary extraction unit with the required B (shaded origin) is shown together with an example of input and output. Note that the input is unsegmented. (b) A hardware unit for pattern spectrum extraction. The pattern spectrum has three main peaks indicating the size of the clusters, i.e. approximately $3 \times 3 = 9$, $33 \times 33 = 1089$ and $39 \times 39 = 1521$ pixels.

When all openings are finished, G_1 to G_n are calculated as the difference between the sums A_0 to A_n. To determine the number of opening branches, some *a priori* knowledge of the image content is required. This is application-specific and depends on the relation between the size of B and the resolution. However, even with a large number of branches, e.g. a quarter of the image height, the memory requirement of the unit is still low due to the use of the proposed architecture, i.e. it is mainly proportional to $I_h/4(\lceil \log_2(B_h) \rceil I_w)$, where I_h is the height of the input image. Furthermore, since the unit preserves the raster scan order, it can run at the same speed as the incoming pixels, but with a latency proportional to the largest B size. Examples of applications where granulometry can be useful include process monitoring and medical applications [65].

Table 1: Synthesis results in the UMC 0.13μm CMOS process, using an image resolution of 640×480 and supporting a maximum B size of 63×63 pixels.

Design	**Delay-line**	**Low complexity**	**Stall-free**
Memory [kB]	4.97	3.08	0.96
mem$_{fifo}$ **area** [mm^2]	0.20	0.18	0
mem$_{dp}$ **area** [mm^2]	0.10	0.04	0.08
Total area [mm^2]	0.69	0.22	0.09
Memory area	43%	98%	88%
Normalized area	7.7	2.5	1
Gate count [k]	135	43	18
Max speed [MHz]	333	190	333

To be able to perform other important and more computationally expensive multi-pass morphological operations such as the hit-and-miss, skeletonization, and convex hull transformation [66], additional intermediate storage as well as an extension to the definition of B is required.

3 Implementation

The architectures have been implemented in VHDL and synthesized for the UMC 0.13 μm CMOS process, supporting an image resolution of 640×480 and a maximum B size of 63×63 pixels (not limited by the architecture). All three architectures can perform either ε or δ, controlled by a single bit, and support changing the B size in between images during run-time, i.e. height and width. Table 1 compiles the most important resource requirements and characteristics of the architectures. Memory area is divided into mem$_{\text{fifo}}$ and mem$_{dp}$; the former is the amount of memory used to stall or align input data and the latter the required memory to calculate the output.

Table 1 shows that memory is a significant part of all three implementations. The delay-line architecture has a lower memory-area percentage than the other two, since this architecture has a more complex controller which handles the padding. For both the low complexity and the stall-free architecture, memory is equal to or greater than 88% of the total area when the row memories are implemented as single-port high-density SRAMs, as further discussed in Section 3.1. In order to distinguish the area requirement relationship between the designs, the normalized area is inferred. This figure shows that the delay-

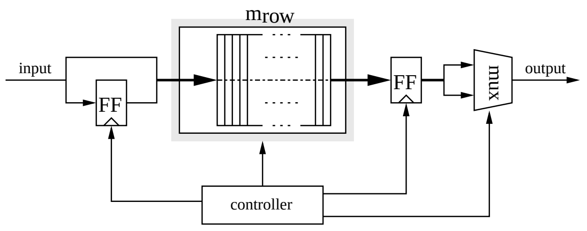


Figure 8: A row memory implemented with one double-width single-port memory, which performs read and write every other clock cycle.

line and low complexity architecture requires a factor of 7.7 and 2.5 more area than the stall-free architecture. Furthermore, it can be noticed that the low complexity architecture has a lower operating speed compared to the others. This is due to the fact that the asynchronous FIFO located at the input of this architecture is replaced by a dual-port memory. The gate count is based on a 2-input NAND-gate (5.12 μm^2) and includes all memory blocks.

3.1 **Memory Architecture**

In a low complexity and stall-free architecture, memory is by far the single largest component, putting constraints on maximum operating speed as well as a lower limit on the area. Therefore, optimizing the memory requirement is of particular concern. Ideally, a value in the row memory (m_{row}) should be read, updated, and written back to the memory in a single cycle. This requires a simultaneous read- and write-operation that normally is implemented using a dual-port memory. However, this type of memory introduces an area overhead mainly due to the dual address decoders, which especially large if the memory is small. Another observation is that the row memories have the memory content access pattern of a FIFO, resulting in the trivialization of the address generation; it can be implemented as a simple modulo-counter. Based on these facts, all row memories can be advantageously implemented using a single-port, double width memory that reads and writes two samples every other clock cycle. The memory architecture is illustrated in Figure 8 along with the simple controller that manages the memory, FFs, and the multiplexer. As an example, using a memory with a depth and width of 320×12 bits and two 12 bit flip-flops, the memory area may be reduced by approximately 30% compared to when a standard dual-port memory is used for this particular process (UMC 0.13μm).

4 Results and Performance

This section discusses and compares the performance of each architecture. The comparison is performed in terms of computational complexity, execution time, and memory requirements.

4.1 Computational Complexity

Computational complexity for the presented architectures is measured in the number of operations per output, i.e. the number of times an input sample is used. Typically, the delay-line architecture uses each input as many times as the number of elements in B in order to support arbitrary-shaped Bs, but can be reduced to $2\lceil \log(N_B) \rceil$ when using a rectangular B (discussed in Section 1.1). Both the low complexity and the stall-free architecture have a constant computational complexity of 4 operations per pixel, i.e. each operation is accomplished with only two summations and two comparisons and each input is used only once, independent of the B size. This is due to the fact that they are based on the same principle which trades the freedom of choosing an arbitrary B shape for reduced complexity.

4.2 Execution Time

In a typical morphological operation, the execution time T_{exe} is the time between the processing of the first input and the production of the last output. It consists of two contributions: pixel processing time, T_{pp}, and padding time, T_{padd}. T_{pp} is the time it takes for the architecture to process all the pixels in the input image and is thus proportional to the resolution and, in some cases, to the B size, depending on wether time multiplexing is used in the implementation. T_{padd} includes all extra clock cycles due to padding and is hence dependent on both the resolution and the size of B, as shown in Figure 4.7.

The execution times, measured in clock cycles, of the delay-line and stall-free architectures are equal to the image resolution, since no padding is inserted into the data stream, and can be written as

$$T_{\text{exe}} = T_{\text{pp}} = I_h \cdot I_w \text{ clock cycles.} \tag{1}$$

The low complexity architecture, on the other hand, needs to insert padding on two sides, resulting in an execution time of

$$T_{\text{exe}} = T_{\text{pp}} + T_{\text{padd}} = I_h \cdot I_w + \lfloor \frac{B_w}{2} \rfloor I_h + \lfloor \frac{B_h}{2} \rfloor (I_w + \lfloor \frac{B_w}{2} \rfloor) \text{ clock cycles,} \tag{2}$$

where the second and third terms correspond to the time it takes to insert the east and south padding.

Comparing (1) and (2) for an input image of 640×480 and a $B = 63 \times 63$, it is found that the low complexity architecture requires approximately an 11% longer execution time due to the inserted padding. With an increasing resolution compared to the B size, this penalty will become smaller and eventually insignificant. However, this architecture still requires multiple clock domains, one for the pixel stream and one for the operating frequency of the architecture.

4.3 Memory Requirement

The required amount of memory for the delay-line architecture can be seen in Figure 1(b) and is calculated as

$$\text{mem}_{dl} = (B_h - 1)(I_w - B_w + 1) + B_h(B_w - 1) \text{ bits}, \tag{3}$$

where the first term accounts for the FIFOs and the second term for the flip-flops used to store the pixels currently covered by the SE.

The memory requirement for the low complexity architecture is proportional to the word-length in each stage, illustrated in Figure 2. The word-lengths in stage-1 and stage-2 depend on the maximum supported B size and are equal to $\lceil \log_2(B_w - 1) \rceil$ and $\lceil \log_2(B_h - 1) \rceil$, respectively. In addition, a FIFO is required at the input since the incoming pixel stream needs to be stalled during the processing of the padding pixels. Thus, the total amount of required memory is

$$\text{mem}_{lc} = \text{FIFO} + \lceil \log_2(B_w) \rceil + \lceil \log_2(B_h) \rceil I_w \text{ bits}, \tag{4}$$

where the second and third term corresponds to the flip-flop in stage-1 and to the row memory in stage-2, respectively. The size of the FIFO not only depends on the padding and resolution but also on the operating and input frequency, f_{op} and f_{in}; the higher f_{op} is in comparison to f_{in}, the smaller the FIFO. If f_{op} is lowered as much as possible while still supporting back-to-back images, i.e. $f_{\text{op}} = \frac{N^2 + N_p}{N^2} f_{\text{in}}$, the size of the FIFO can be approximated as

$$\text{FIFO} \approx N_{\text{sp}} \frac{f_{\text{in}}}{f_{\text{op}}} = N_{\text{sp}} \frac{N^2}{N^2 + N_p} \approx N_{\text{sp}} \text{ bits}, \tag{5}$$

where N_{sp} is the size of the south padding pixels, N_{p} is the sum of all padding pixels, and N^2 is the input image in pixels. This means that since the sum of

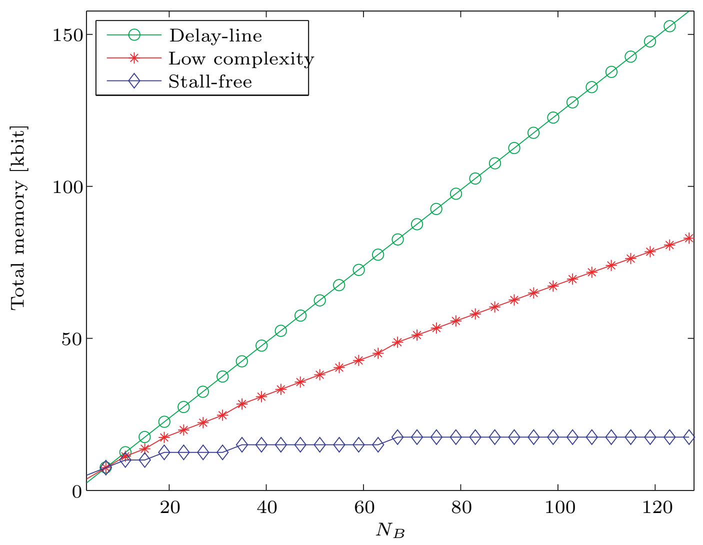


Figure 9: The vertical axis shows the total memory requirement in kbit as a function of N_B for each implementation. The image resolution is 1280×1024 and ${N_B}^2$ is the size in pixels of a quadratic B.

all padding pixels is small compared to the resolution, i.e. $N_\mathrm{p} \ll N^2$, the input FIFO must be able to store as many pixels as the number of south padding pixels, when operating at the lowest possible frequency.

The memory requirements of the stall-free architecture follows the principles of (4) but without the FIFO. The resulting total memory requirement can be written as

$$\mathrm{mem}_{sf} = 2(\lceil \log_2(B_w) \rceil + \lceil \log_2(B_h) \rceil I_w) \text{ bits}, \tag{6}$$

where factor 2 is due to the parallel processing during padding. However, removing the FIFO at the input still has a significant impact on the memory requirements, as shown in Table 1. Equation (6) indicates that the memory area of the data-path in the stall-free architecture should be twice the size of the low complexity memory, but this is not the case when comparing mem_{dp} in Table 1. The explanation is that instead of using two separate row memories, one single row memory of double width is used, as discussed in Section 3.1.

Hardware requirements for implementations supporting higher resolutions and Bs can be estimated accurately using the required memory size since this is the main source, as shown in Table 1. The memory requirements for the three

Table 2: The most important properties of the architectures, where N^2 and ${N_B}^2$ is the size in pixels of a quadratic input image and B.

Design	**Delay-line**	**Low complexity**	**Stall-free**
Complexity	$2\lceil \log_2(N_B) \rceil$	4	4
mem accesses	$2N_B$	$2+\alpha$	$2+\alpha$
mem [bits]	$N\,N_B$	$N\,\log_2(N_B)$	$N\,\log_2(N_B)$
$\mathbf{T}_{exe}$	N^2	$N^2{+}N_B\,N$	N^2
SE support	Arbitrary	Rectangular	Rectangular

architectures as a function of B size using an image resolution of 1280×1024 is shown in Figure 9. As an example, the total memory requirement of a stall-free architecture supporting a maximum B size of 63×63 is about 15 kbits. With the same settings, the delay-line implementations would require approximately 79 kbits of memory, which is more than 5 times as much. Table 2 summarizes the most important properties of the different architectures as functions of image resolution and B size. The table clearly indicates that for applications in which a rectangular B is sufficient to fulfill the specifications, the stall-free architecture reduces computational complexity and memory requirements without sacrificing execution time.

From a power perspective, it is advantageous to have a low arithmetic complexity and to minimize the number of memory accesses since both contribute to the dynamic power budget. The actual number of memory accesses per pixel for the delay-line implementation is mainly proportional to $2 \cdot (B_h - 1)$, since each value is shifted downward one row in Figure 1 each time it is to be used in a calculation (the factor two is for reading and writing). For the low complexity and the stall-free architectures, this number is reduced to $2 + \alpha$, where α is the additional $2\lfloor \frac{B_h}{2} \rfloor (I_w)$ accesses required during the south padding (neglecting the input FIFO read- and write-operations required in the low complexity architecture). As an example, using a resolution of 640×480 and supporting a maximum B size of 63×63, these additional memory accesses only constitute $\approx 6.5\%$ of the total number memory access per frame or an additional $\alpha = 0.13$ accesses per pixel. The conclusion is that both the low complexity and the stall-free architectures mainly require one read and one write operation per pixel. Furthermore, since static power consumption is becoming increasingly important in modern CMOS technologies due to leakage, it is beneficial to reduce the overall area [6]. For the designs in this article, area mainly constitutes of memory. A reduced memory requirement will therefore not only have a large impact on the static but also the dynamic power since accessing smaller memories requires less power than larger ones. Based on these facts and the results in Table 2, it is seen that stall-free architecture has the lowest complexity and the lowest memory requirements both in terms of bits and accesses, and hence has better dynamic and static power dissipation properties than the other designs.

5 Conclusion

An evaluation of three architectures for binary erosion and dilation intended to be used as hardware accelerators in real-time applications is presented. In particular, the architecture of a fast, stall-free, low complexity architecture based on B decomposition is proposed. The most important features and properties are that it requires no extra clock cycles due to padding and has a memory requirement proportional to B height and input image width. The architecture supports flat arbitrary sized rectangular Bs and the number of operations and memory accesses per pixel is constant, independent of both B and image size. Furthermore, due to its low complexity and memory requirements, multiple units can be connected without any intermediate storage to perform other morphological operations. In order to verify and evaluate the results, the architecture has been implemented in VHDL and synthesized for a UMC $0.13\,\mu$m CMOS process using a resolution of 640×480 and supporting a maximum B of 63×63. In comparison to implementations of the delay-line and the low complexity architecture using the same parameter setting, the area is decreased by a factor of 7.7 and 2.5, respectively, while achieving the same or an improved execution time.

Part II

Binary Morphology with Locally Adaptive Structuring Elements: Algorithm and Architecture

Abstract

Mathematical morphology is parameterized using the structuring element which is usually constant throughout the processing of the entire image. However, allowing locally adaptive structuring elements is advantageous in many situations whenever one can let the structuring element locally adapt to certain high-level information, e.g. apparent size of the objects, granularity, texture, or direction. In an effort to achieve this performance increase, this part presents a novel algorithm for binary morphological erosion and dilation with a flexible structuring element, together with a corresponding hardware architecture. The algorithm supports resizable rectangular structuring elements, and has a linear computational complexity and memory requirement. In order to achieve high throughput, the proposed architecture maintains the important raster-scan pixel processing order, and requires no intermediate storage for the image data. This part concludes with implementation results of the architecture when targeted for both FPGA and ASIC.

Based on: H. Hedberg, P. Dokladal, and V. Öwall, "Binary Morphology with Locally Adaptive Structuring Elements: Algorithm and Architecture," first round of revision for publication in *IEEE Transactions on Image Processing.*

1 Introduction

Although a structuring element with fixed shape and size may be sufficient in some cases, locally adaptive, flexible-shaped structuring elements, called amoebas, may outperform the static ones in many applications. Within the framework of mathematical morphology, they have been proposed for noise filtering in [67]. The amoebas, which take into account the image contour variations to adapt their shape, outperform classical morphological operations with a fixed, space-invariant structuring element. Other applications where locally adaptive structuring elements are useful are those using a zoom camera or where one has to take into account the perspective deformations. As an example of an application where noise filtering with a flexible structuring element outperforms a static one - a typical road surveillance - is shown in Figure 1(a) with the corresponding segmentation result shown in b. Using the depth information in the scene to control the structuring element size (increasing towards the lower part of the image) can significantly reduce over-segmentation. The difference can be distinguished when comparing the output from when using a static and a flexible structuring element, shown in Figure 1(c) and (d), respectively. The performance improvement can be seen as correctly separated and more homogenous objects.

Structuring elements are also used for object detection. Objects deformed by the perspective - having different sizes in different parts of the image - have to be detected in several passes with correspondingly scaled structuring elements. Using flexible structuring elements allow detection of such objects in only one pass.

The conclusion is that most real-time image processing based system with non-stationary or constant scene content requiring MM can benefit from using a flexible structuring element.

This work represents the first step towards arbitrary-shaped flexible structuring elements in efficient hardware implementations. The main contribution of this part is threefold:

1) A new algorithm is presented supporting flexible rectangular structuring elements for binary mathematical morphology with low computational complexity and memory requirement. These properties make the algorithm applicable also for software implementations. Note that such a software implementation can run *in-place*, meaning that the result can overwrite the input data.

2) A corresponding hardware architecture of the algorithm is proposed suitable for embedded or mobile applications. The architecture has several important properties from a hardware perspective, i.e. sequential pixel processing, low computational complexity, and low memory requirement.

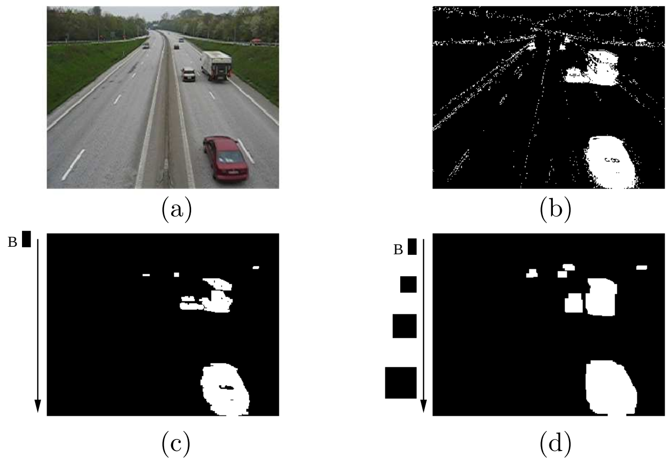


Figure 1: (a) An example of a typical road surveillance application where using a flexible structuring element is useful. (b) Binary motion mask from the segmentation algorithm. (c) Filtered motion mask using a constant structuring element. (d) Filtered motion mask using a flexible structuring element, increasing in size towards the lower part of the image where vehicles appear larger.

3) Implementation results of the proposed architecture are presented in terms of resource utilization when targeted for both FPGA and ASIC.

1.1 Previous Research

In direct mapped implementations of mathematical morphology operations, to output a pixel value one needs to examine the pixel's entire neighborhood defined by the structuring element in the input image, e.g. the delay-line architecture discussed in Part I. As a consequence, using large neighborhoods becomes particularly computationally intensive and efforts have been made to make these operations more efficient.

The implementations by Van Herk [68] and by Lemonier and Klein [69] support large linear structuring elements but need two scans to complete each operation, thus requiring intermediate storage. Furthermore, the computational complexity is proportional to N_B and memory requirement $N_B N$ and they are more targeted for gray scale applications, where N_B is the structuring element width and N the side in a square image.

Van Droogenbroeck and Talbot [70] propose an algorithm based on a histogram. The histogram is updated as the structuring element slides over the image. The respective value for the needed rank filter (dilation, erosion or median) is taken from the histogram. However, this algorithm is not suitable for adaptive structuring elements, and also computing the histogram requires additional resources.

Another approach is found in [71], where algorithmic support for structuring elements with a fixed shape but with a locally adaptable size is achieved by using the distance transform [72]. However, the distance transform is a multi-scan operation with a high memory requirement since it requires a large intermediate storage for partial results in between the scans. In [72], no hardware architecture or implementation results are presented.

The proposed architecture is a development from the ones presented in Part I, which is from now on referenced as the Low Complexity architecture (LC). The new architecture allows for changing the size of the rectangle within an image from pixel to pixel, and can thereby locally adapt its size. Although having mainly the same memory requirement, the structuring element flexibility comes at the cost: the computational complexity is increased from being constant to being proportional to N_B.

2 Algorithmic Issues

The structuring element defines which pixel values in the input image I to include in the calculation of the output value located at the current position of the origin, as discussed in Section 4.1. Whereas for a static structuring element, the geometric shape is constant throughout the input image, the shape of a flexible structuring element can change from pixel to pixel within the frame.

The local shape adaptation of the structuring element replaces in (4.2) and (4.4), the fixed set B translations $x+b$ given by all $b \in B$ by a flexible set given by $B : D \to \mathcal{P}(D)$ with $D = \text{supp}\{I\}$, and $\mathcal{P}$ denoting the set of subsets. This means that $B(i,j) \subset D$, becomes a function $B : \mathbb{Z}^2 \to \mathcal{C}$, where for every pair of coordinates i, j, the function $B(i,j)$ chooses an element from the class $\mathcal{C}$ of allowed shapes.

In order to avoid unexpected morphological results and maintain the raster scan order discussed in Section 3.1.4, both morphological and algorithmic specific restrictions have to be inferred, each addressed in consecutive order:

1) **Morphological restrictions** concern the continuity of the function describing the shape of structuring elements used at different coordinates.

For the sake of simplicity, consider first a binary object X in the 1-D space $\mathbb{R}$, i.e. $X : \mathbb{R}^+ \to \{0,1\}$, as illustrated in Figure 2(a). In the same space, consider also a one-sided line structuring element $B(x) \subset \mathbb{R}^+$, $\forall x \in \mathbb{R}^+$, equipped with the origin at x, encoded by its length $\beta : \mathbb{R} \to \mathbb{R}^+$. Figure 2(b) shows an

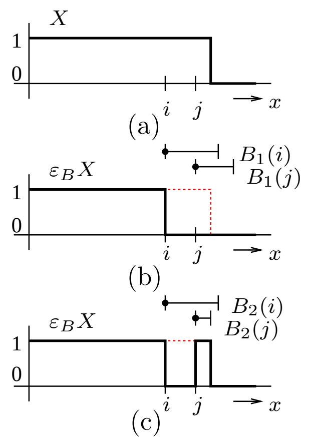


Figure 2: (a) A 1-D binary object X. (b) Erosion $\varepsilon_B X$ with a flexible structuring element $B(x)$. (c) Example of a discontinuous result $\varepsilon_B X$ with B varying "too fast".

example of an erosion of a 1-D object X by a function B_1, illustrated at i and j (the origin of $B_1(i)$ is given by the black dot above i). The eroded object $\varepsilon_{B_1} X(x) = 0$, for $x = i, j$, note that $B_1(j)$ may be different from $B_1(i)$. Figure 2(c) shows the same X eroded by a different function B_2. The eroded object $\varepsilon_{B_2} X(i) = 0$ but $\varepsilon_{B_2} X(j) = 1$ since $B_2(j)$ does not extend outside X as does $B_1(j)$ in Figure 2(b). Eroding by B_2 has created a hole in X at i instead of 'eroding' X on the surface.

In the following, only the case illustrated in Figure 2(b) is considered, where the function $B(x)$ preserves the continuity of X, by eroding only on its surface, i.e. $X(i) = X(j) = 1$, and $i < j$, then $\varepsilon_B X(i) = 0$ implies $\varepsilon_B X(j) = 0$. The continuity of $\varepsilon_B X$ is ensured if $j - i \geq \beta(j) - \beta(i)$, or $|(\beta(j) - \beta(i))/(j - i)| \leq 1$, or if β is differentiable,

$$\left| \frac{\partial \beta(x)}{\partial x} \right| \leq 1. \tag{1}$$

Recall that $\beta(i)$ is the length of $B(i)$. The conclusion is that erosion by B does not create holes if the size of the structuring element $B(x)$ does not change "too fast".

In $\mathbb{Z}^2$, ∂ is obviously replaced by Δ, and the continuous distance $\beta(x)$ is replaced by the quantized offset N_u, N_l, N_d and N_r in directions discretized at

90°, 180°, 270° and 360° for *up*, *left*, *down* and *right*, noted by:

$$\left|\frac{\Delta N_q}{\Delta i, \Delta j}\right| \leq 1, \text{ for } q = u, l, d, \text{ and } r. \quad (2)$$

2) **Algorithmic restrictions**: In order to preserve the raster scan order and still support arbitrary-shaped structuring elements in low complexity and memory requirement architectures with no intermediate storage, some restrictions have to be inferred. Since the input pixels arrive in a stream, pixels above and to the left from the origin are already known, while the pixels below and to the right are to be read. However, one cannot output the current pixel (i, j) until its entire neighborhood has been processed. Indeed, the latency L between the input and the output stream for any pixel (i, j) is given by

$$L(i, j) = N_d(i, j)I_w + N_r(i, j), \quad (3)$$

where I_w is the width of the image I, N_d and N_r are the coordinates of the origin offset from the bottom-right corner (d=down and r=right), illustrated in Figure 3.

When using a static structuring element for the entire image, i.e. $B(i, j) = B$, $\forall i, j$, the latency is constant; the raster scan order is maintained and the algorithm can be implemented without intermediate storage. However, if the structuring element changes from pixel to pixel, the latency varies. Consequently, the output pixels can arrive in a different order and there is a need to store them in an intermediate memory to retain raster scan order.

The algorithmic restrictions on the flexibility of B introduced above (see (2)), impose that $|\Delta N_i| \leq 1$ for all four directions $i = u, l, d$ and r (up, left, down and right). From (3), we have $\Delta L/\Delta N_r = 1$, and $\Delta L/\Delta N_u = \Delta L/\Delta N_l = 0$, meaning that increasing/decreasing the size of the structuring element by one pixel to the right will increase/decrease the latency by one. Adding above and to the left has no impact on the latency since these pixels have already been read. Hence, unitary changes of L from pixel to pixel, i.e. $|\Delta L/\Delta i, j| = \pm 1$, may be handled with no additional memory by stalling the input or output. Indeed, if N_r increases/decreases, the latency L increases/decreases, and the output/input is stalled. Stalling the output means that two input pixels are read before the next output value is calculated, whereas stalling the input means that two pixels are produced before the next input pixel is read.

Henceforth, for the rest of the paper, we put a restriction on the class $\mathcal{C}$ of allowed shapes, to be a class of rectangles, not necessarily symmetric around the origin. Therefore, $B(i, j)$ becomes a function $B : \mathbb{Z}^{+2} \rightarrow \mathbb{Z}^{+4}$, i.e. for every pair of coordinates (i, j), the function $B(i, j)$ yields a quadruple (N_u, N_l, N_d,

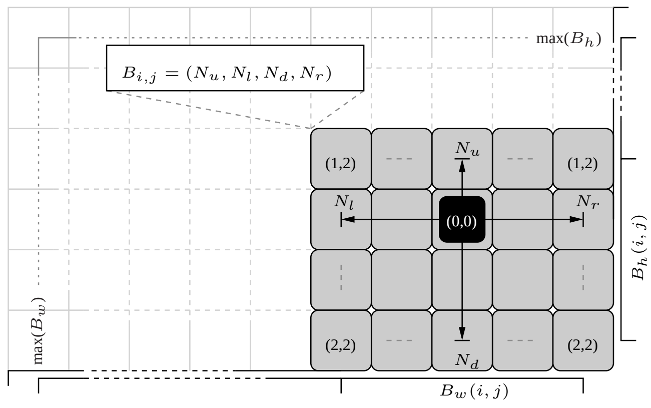


Figure 3: Example of a 4×5 $(B_h \times B_w)$ rectangular structuring element with $B(i,j) = (N_u,\, N_l,\, N_d,\, N_r) = (1,2,2,2)$.

N_r) defining the position of the origin with respect to the upper, left, down and right edge of the rectangle. These parameters are tied to its width and height by $B_w = N_l + N_r + 1$, and $B_h = N_u + N_d + 1$. The maximum width and height found in the collection of $B(i,j)$, $\forall i, j$, are denoted $\max(B_w)$ and $\max(B_h)$, respectively. Figure 3 shows an example of a structuring element $B(i,j) = (1,2,2,2)$, being a 4 by 5 rectangle with the origin offset by 2 rows and 2 columns from the lower-right corner.

From (3), $\Delta L/\Delta N_d = I_w$ means that increasing the size of the structuring element by adding one bottom row to it will increase the latency L by the entire width I_w of the image. This substantial change of latency cannot be handled without using additional buffer memory. This means that from pixel to pixel, the rectangle can grow/diminish by one at all sides, except when adding/deleting one bottom row, authorized only between image rows. The conditions given by (2) become stronger:

$$\left|\frac{\Delta N_{u,l,r}}{\Delta i, \Delta j}\right| \leq 1, \quad \left|\frac{\Delta N_d}{\Delta i}\right| \leq 1 \text{ and } \left|\frac{\Delta N_d}{\Delta j}\right| = 0. \tag{4}$$

An example of synthetic test data (640×480 pixels) is illustrated in Figure 4(a). The image contains a set of black spots (on a uniform grid 60 pixels). A dilation applied to this input image will enlarge the black spots. Figure 4(b) gives a dilation obtained with a rectangular structuring element progressively increasing in size from left-up right-downwards of the image: $B(i,j) = (N_u,$ $N_l,\, N_d,\, N_r) =$ (i/20, j/20, i/20, j/20).

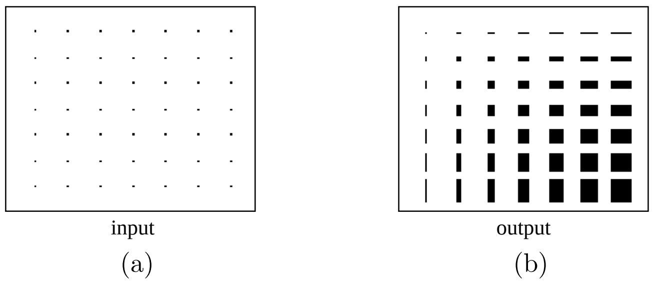


Figure 4: (a) A synthetic test image containing black dots on a grid, corresponding to the foreground. (b) Dilation obtained with similar structuring element as in Figure 1, i.e. a rectangle increasing in size from the top left corner of the image downwards to the right.

3 Algorithm Description

The algorithm reads the input image I and writes the output image O sequentially in the raster scan order. Let (k, l) denote the current reading and (i, j) current writing position. Figure 5(a) gives a synthetic example image $I(M, N)$ containing one object - a car. The object contains 1, and the background 0. Obviously, by causality, reading position (k, l) proceeds writing position (i, j). The latency L between reading and writing the data depends of the size of the currently used structuring element $B(i, j)$ (3). Since $B(i, j)$ differs for different coordinates, also the latency L varies.

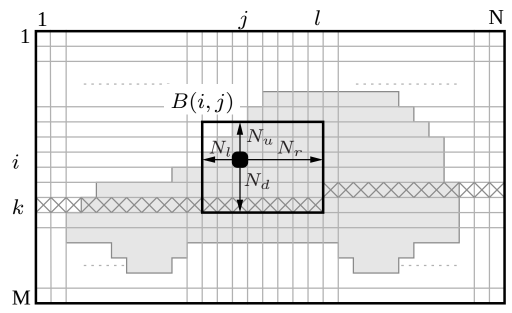
The structuring element shape function B is a parameter of the morphological operation and is also read in raster scan order at the same rate and position as the output image O.

Reading (k, l) and writing (i, j) positions are bound by

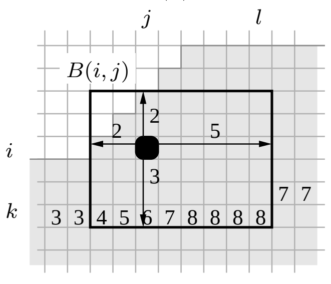
$$\begin{aligned} i &= \max\left(1, \min(k + N_d(i,j), M)\right) \text{ and} \\ j &= \max\left(1, \min(l + N_r(i,j), N)\right). \end{aligned} \tag{5}$$

Suppose the currently processed pixel is located at coordinates (i, j). The corresponding structuring element $B(i, j)$ - placed by its origin at (i, j) - has just been read. Recall its current size $B(i, j)$ is coded by (N_u, N_l, N_d, N_r), equal in the Figure 5 example to (2, 2, 3, 5). The input data need to be read up to the bottom right position of $B(i, j)$, indicated as (k, l).

The algorithm proceeds by decomposing the erosion into columns. In each column $1, \ldots, N$ of the input image I, the algorithm keeps track of the distance $d(1, \ldots, N)$ from the currently processed line to the closest upwards zero



(a)



(b)

Figure 5: (a) A synthetic input image when processing a pixel at location (k,l) with $B(k,l)$. (b) A zoom in of when processing the same pixel using $B(k,l) = (2,2,3,5)$ as structuring element. The numbers in bottom row of $B(k,l)$ show the current distance values, which saturates at the value 8.

(background). For each column, the distance is updated as I is sequentially read row by row:

$$d(l) = \begin{cases} 0 : & d(l) = 0 \\ 1 : & d(l) = d(l) + 1 \end{cases} \tag{6}$$

Figure 5(a) indicates by $\times$ where the known distances are currently located. Notice that for the currently processed pixel $O(i,j)$, situated on the row i, the distances are calculated on a different row k. The corresponding distance values for this particular example are shown in Figure 5(b).

These distances are then compared, column by column, to the height of the currently used structuring element $B(i,j)$, given by $N_u + N_d + 1$. This evaluation, at position $O(i,j)$ in the output image, can be formalized as if the comparison

$$d(l) \geq N_u + N_d + 1, \tag{7}$$

yields TRUE, for all $l \in [\max(1, j{-}N_l), \min(j{+}N_r, N)]$, then at position $O(i,j)$ write 1, else write 0. The whole algorithm can be written as:

Algorithm:

```
for i=1 ..M
  for j=1..N
    read B(i,j)
    read I up to (k,l)                 (Eq.6)
    update d up to l                   (Eq.7)
    O(i,j)=AND(d(l) $\geq$ N$_u$+N$_d$+1)      (Eq.8)
    write O(i,j)
  end
end,
```

where AND means 1 if comparisons for all used l yield TRUE, else 0 (see 7). In the example shown in Figure 5, for the pixel (i,j), the distances $d(j-2) = 4$ and $d(j-1) = 5$ are smaller than the height $N_u + N_d + 1 = 6$, therefore the output written at $O(i,j)$ is 0.

The distance calculation is an independent process of the morphological operation being performed; the memory content is unrelated to the dimensions and origin of $B(i,j)$. It is this algorithmic property that allows an adaptable structuring element different for each individual pixel, since no information about a former B propagates through the architecture.

A block diagram of the algorithm is illustrated in Figure 6. A controller is needed to stall the input and output depending on how the parameters for the structuring element change. Based on these control signals, the distances stored in the row memory in the update stage are either updated with a new pixel value (if B is the same as for the previous pixel or increased) or left unchanged (if B is decreased). However, the output value from the update stage is always equal to the last calculated distance for the column of the current pixel. This distance is used as input to both the compare stage and to the serially connected Flip-Flops (FF-chain), in order to let the distances propagate to be used in multiple calculations. The distances stored in the FF-chain (for the previous columns) are all used as inputs to the compare stage. The controller selects which of these distances to include in the calculation, which are compared to the height given by the current B; if all are greater or equal to this height, output 1 else 0. The result from the compare stage is sent together with the generated control signals for the current B to an output

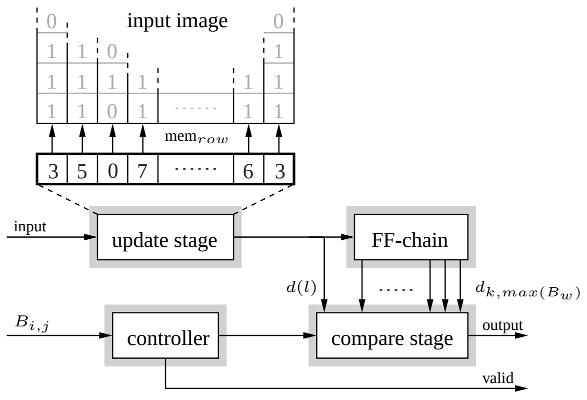


Figure 6: Block diagram of the proposed algorithm.

controller unit which synchronizes the output with a valid signal.

4 Architecture

A more detailed hardware architecture of the proposed algorithm is illustrated in Figure 7. As in Figure 6, the architecture is divided into three stages: update, FF-chain, and compare. In the update stage, the row memory (mem$_{row}$) stores the distances for each column in the input image, which are updated according to (6). This is implemented as an incrementer and a multiplexer (placed in the middle of this stage in the figure), where the input from the FIFO is the control signal to the multiplexer, which outputs the reset or the increased sum for further processing. If the distance is equal to the maximum supported structuring element height $\max(B_h)$, the sum saturates at this value, which also is the initial value in order to leave the result unaffected at the image borders.

The FF-chain stage contains the delay elements to stall the distance $d_{k,\max(B_w)}$ to d_l which are included in the current calculation, i.e. to be evaluated against the columns in the current $B_{i,j}$. This is implemented as a series of FFs. The block also includes multiplexers for initialization when encountering a new row in the input image.

The compare stage compares the stored distances to the height of the structuring element. The number of comparators equals the maximum supported structuring element width, $\max(B_w)$. The results from the comparators serve as input to the logic AND-operation. Notice that the fan-in to this unit increases linearly with $\max(B_w)$, thus affects the critical path and is the major

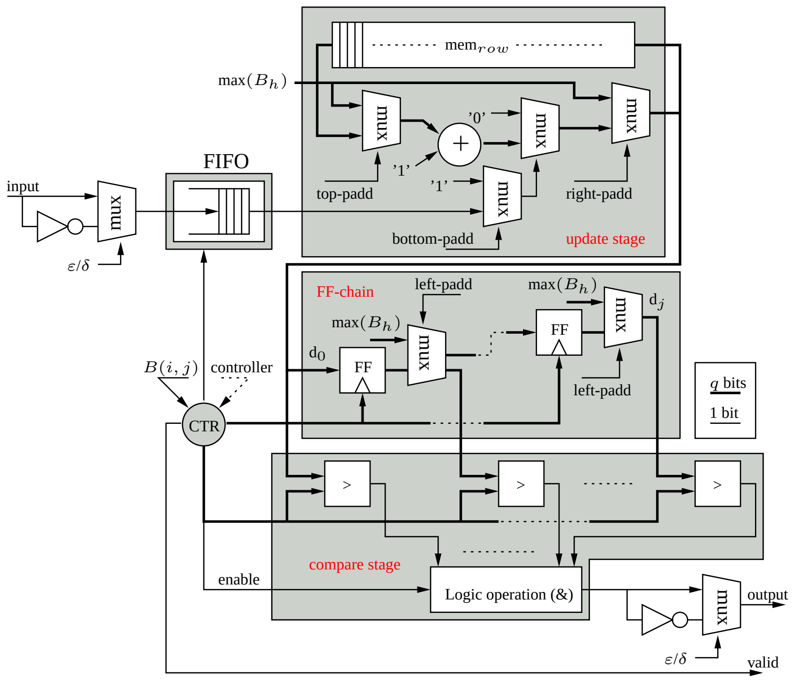


Figure 7: Overview of the implemented architecture. Note that the data valid signal is not included in figure but is generated by the CTR block.

bottleneck of the architecture. Hence, for large structuring elements or high speed applications, a pipeline can be inferred. Using pipelining, one or several additional delays are required to synchronize the output with the data valid signal.

The CTR block in Figure 7 is implemented as an Finite State Machine (FSM) and manages all control signals in the architecture based on $B(i, j)$. This includes not only padding but also the enable signal to decide the number of active comparators (enable), and which operation to perform (operation), i.e. ε or δ. By default, the architecture performs a logic AND-operation (minimum) on the compared distances, i.e. $d_{k,\max(B_w)}$ to d_l, which in mathematical morphology corresponds to an erosion. Due to the duality principle discussed in Section 4.5, to perform a dilation by default, simply calculate the distances to closest upward one for each column and perform a logical OR-operation (maximum). However, another way to obtain a dilation and still use the default operation is to simply invert the input and the output, accomplished in

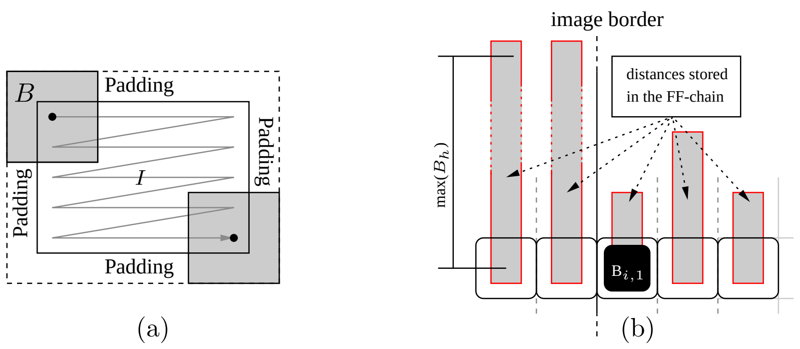


Figure 8: (a) An illustration of where the structuring element may stretch outside the image border and where padding may be required. (b) An illustration of the left padding for a linear structuring element with a centered origin.

HW by placing a multiplexer and an inverter at the input and the output of the architecture, shown in Figure 7.

4.1 Handling the Borders

The architecture needs to address the boundary problem, as discussed in Section 4.6 and illustrated in Figure 8(a). Supporting a locally adaptive structuring element requires the possibility to stall the input data stream since the latency can vary from one side of the image to the other (3), resulting in the need for two separate clock domains and a FIFO located at the input. Therefore, handling the padding pixels on a controller level is not feasible. Instead, the padding pixels are handled by the architecture in one of two ways: precalculated initial values (top and left padding) or pixels inserted into the data stream (right and bottom padding). The result is a less complex controller but with the drawbacks of requiring two clock domains and an input FIFO. Padding control is included in CTR in Figure 7 with corresponding control signals, i.e. top-, left-, right, and bottom-padding.

Processing the first row in an image, initial values equal to the maximum supported structuring element height max(B_h), are inserted into the adders through the leftmost multiplexer in the update stage in Figure 7. This is due to the fact that the distance to the closest upward zero for this preceding row is assumed to be infinite and should not affect the calculation. The initial values are updated with the pixel value in the input image and the result is sent to both the compare stage and written into the memory.

Assuming a linear structuring element, e.g. $B_{i,j} = 0, 2, 0, 2$, calculating the first output pixel of a new row is an example requiring left padding, Figure 8(b). When starting at a new row i, each distance to the left of the first pixel located at $(i, 1)$ are set to the initial value $\max(B_h)$ simultaneously by using the multiplexers in the FF-chain stage. This procedure causes the distances located beyond the image borders not to affect the calculation. When reaching and extending the structuring element beyond the right image border, the same initial value is inserted into the data stream and sent to the compare stage through the rightmost multiplexer in the update stage.

Reaching the bottom segment of the input image, the structuring element can stretch outside the bottom border. Depending on the actual height of $B_{i,j}$, additional ONES are inserted in the pixel stream, at most $\lfloor \frac{\max(B_h)}{2} \rfloor (I_w + \lfloor \frac{\max(B_w)}{2} \rfloor)$, through the lower multiplexer in the update stage. This insertion is necessary to handle the different latency that will occur in a video stream if different sizes of the structuring element are used at the end of one image to the beginning of the next. During the insertion of these extra pixels, the input data stream is stalled (requiring the FIFO on the input). Once the last pixel has been processed, the erosion operation is complete and starts over with the next frame.

4.2 Coding the Structuring Element Size

The structuring element size is controlled by the function B for each pixel (i, j) through the parameters N_u, N_l, N_d, N_r, defined in Section 2. The parameters are generated outside the architecture and are sent as input in parallel with the input pixel stream to the controller in Figure 7. Formally, $B(i, j)$ becomes $B(i, j, t)$ with $I(t)$ for video sequences and it is the user's responsibility to design the application-dependent $B(i, j, t)$ generation process, which must fulfill the conditions in (4). In order to reduce the bandwidth of the generation process, an efficient coding of B is required.

Given these conditions, instead of coding the size $N_{u,l,d,r}(i, j)$ directly, one can code the difference $\Delta N_{u,l,d,r}$ between two adjacent pixels. Limiting the difference to $|\Delta N_{u,l,r}/(\Delta i, \Delta j)| \leq 1$, the coding can be represented by using two bits, i.e. increase ("01"), decrease ("11"), no-change ("00"), and reset ("10"), corresponding to a simple Δ-code (or difference-code). The reset value can be used to restore B to an initial setting at the beginning of each row. Thus, coding $B(i, j, t)$ will require $3 \times 2 = 6$ bits between two adjacent pixels in the same row (since N_d is not allowed to change in the middle of a row), and 2 more bits in between two consecutive lines to represent N_d, ending up with a total number of 8 bits to code $B(i, j, t)$.

4.3 Memory Requirements

The row memory located in the update stage stores the distances for each column and is the largest single internal component in the architecture (excluding the input FIFO). The requirement is linearly proportional to the resolution according to

$$\text{mem}_{row} = \lceil \log_2(\max(B_h)) \rceil \cdot I_w \text{ bits}, \tag{8}$$

where $\max(B_h)$ is the maximum supported structuring element height which determines the number of bits per stored value according to $q = \lceil \log_2(\max(B_h)) \rceil$. Additional registers in the FF-chain are needed to delay the stored distances (mem_{row} content) serving as input to the comparators, Figure 7. The number of registers is proportional to the maximum allowed structuring element width. Since their content should be compared to the maximum structuring element height, the number of bits in these registers is

$$\text{FF}_{chain} = q \cdot \max(B_w) \text{ bits}. \tag{9}$$

Combining (8) and (9), the total memory requirement for the algorithm is of $O(\max(B_w))$ and is equal to

$$\text{mem}_{tot} = \text{mem}_{row} + \text{FF}_{chain} = k(I_w + \max(B_w)) \text{ bits}. \tag{10}$$

As for the row memory in the low complexity architecture discussed in Part I, Section 3.1, a value in mem_{row} in the update stage, should be read, updated, and written back to this memory in a single cycle. Using a resolution of 640×480 together with supporting a maximum structuring element of size 63×63, a memory of size a 6×64 with dual-port functionality is required according to (8). However, due to the memory access pattern, the same memory architecture can be used as the LC-architecture, i.e. implemented using a single-port memory of double width (12×320), two registers, a multiplexer, and a controller, running on the same clock domain as the input data, that reads and writes two samples every other clock cycle.

Due to the stall cycles discussed in Section 4.1, the architecture requires an asynchronous FIFO at the input. The size of this FIFO is a trade-off between operating frequency and memory resources and is set by many parameters, e.g. input data speed, operating speed, resolution, and maximum supported structuring element; the higher operating frequency, the smaller FIFO. Let f_{in} be the speed of the incoming pixels, the critical time for the architecture to process a frame can be written as $t_{in} = \frac{1}{f_{in}} \cdot I_h \cdot I_w$, which includes the padding cycles. The number of additional clock cycles is determined by two factors: the size of the structuring element and the location of the origin. In the worst case, the origin is located in the top left corner of a large structuring element requiring $\max(B_h)(I_w + \max(B_w))$ additional cycles since none of the distance values located outside the image border can be precalculated (left and top padding). Using a centered origin, the number of padding cycles is reduced to $\lfloor \frac{\max(B_h)}{2} \rfloor (I_w + \frac{\max(B_w)}{2})$ since half of the values can be inserted as initial values in the FF-chain, recall Section 4.1. However, assuming the worst case, the total time t_{op}, it takes for the architecture to process complete frame is equal to $t_{op} = \frac{1}{f_{op}}(I_h + \max(B_h))(I_w + \max(B_w))$, which includes all extra cycles due to padding. Using these two expressions, a timing constraint for the architecture can be written as $t_{in} \geq t_{op}$, or expressed in the operating frequency as

$$f_{op} \geq f_{in} \frac{(I_h + \max(B_h))(I_w + \max(B_w))}{I_h \cdot I_w} \text{ MHz.} \tag{11}$$

Assuming that the maximum supported structuring element is small compared to the resolution, i.e. $\max(B_h) \ll I_h$ and $\max(B_w) \ll I_w$, then $f_{op} \approx f_{in}$. Using this approximation and assuming a centered origin, the architecture must at most stall $\lfloor \frac{\max(B_h)}{2} \rfloor (I_w + \lfloor \frac{\max(B_w)}{2} \rfloor)$ pixels during padding at the lower boundary of the image, as discussed in Section 4.1. With f_{op}=10 MHz, a resolution of 640×480, a maximum supported structuring element of 63×63, and a frame rate of 25 *fps* (f_{in}=7.68 MHz), the required FIFO capacity is 21 kb. With these settings and using (10), the total amount of memory for the complete architecture is $\approx$ 25 kb. Calculating the memory requirement when increasing f_{op} to 100 MHz, the architecture requires a FIFO with a capacity of $\approx$2 kb, reducing the total amount of memory to $\approx$ 6 kb.

Determining the FIFO size is not a trivial problem since it impacts both the dynamic power consumption according to $P_{dyn} \propto f_{op}$ [6], and the static power dissipation (area dependent). In practice, if minimizing the dynamic power is of high priority, this means operating at the lowest possible speed (for a given supply voltage), i.e. minimizing f_{op}, resulting in a large FIFO. To summarize, the memory requirement is dependent on the operating speed f_{op} and memory resources can be traded for low power properties.

A possibility to reduce the memory requirements on an architectural level is to connect two architectures in parallel, multiplexing the processing of consecutive frames between the two. The FIFO can be substantially smaller or even removed, reducing the memory requirements to approximately 8 kb. However, this is highly application dependent, putting additional constraints on the SE flexibility when processing the two frames in parallel.

Table 1: Implementation resource utilization characteristics of a Xilinx Virtex II-PRO FPGA and in the UMC 0.13 μm CMOS process using a resolution of 640×480 and supporting a flexible SE up to 63×63.

FPGA	used	available	**ASIC**	used
Slices	807	13696	**Area** [mm^2]	0.24
Block RAM	3	136	$\mathbf{Mem}_{tot}$ [kb]	24.6
LUTs	1365	27392	**Gate count** [k]	47
Speed [MHz]	80	—	**Speed** [MHz]	260

5 Implementation Results and Performance

The architecture has been implemented in VHDL using a resolution of 640×480 and supporting a flexible structuring element of up to 63×63. This choice is obviously application specific. Indeed, in order to correctly filter the largest objects found in the image, we have chosen the largest B to be approximately 1/10 of the image width. In general, there are no algorithmic restrictions on the largest supported structuring element size, but q in (10) will increase accordingly.

The implementation has been targeted for both FPGA and ASIC: verified on a Xilinx Virtex-II PRO FPGA (XC2VP30-7FF896) and synthesized for the UMC 0.13 μm CMOS process, respectively. The most important implementation results and properties for both technologies are compiled in Table 1, where the area is reported with all memory blocks. This includes an asynchronous input FIFO of 21 kb, as discussed in Section 4.3 (replaced by a dual-port memory of the same size in the ASIC implementation to support multiple clock domains), resulting in that memory constitutes 86% of the total area in this particular implementation. The gate count is based on a 2-input NAND-gate (5.12 μm^2).

As mentioned in Section 4, the combinatorial critical path passes through the logical operation performed in the compare stage. Pipelining this operation will not necessarily increase the speed figures found in Table 1 since the bandwidth to the memory is the limiting factor.

In order to compare this work to the PRR and LC architecture discussed in Section 1.1 and Part I, important properties are compiled in Table 2 as a function of the resolution and the maximum supported structuring element. SE support refers to the class of supported structuring elements and SE flexibility to the ability to change it between two adjacent pixels. Naturally, this should be distinguished from the ability to change the structuring element in between

Table 2: Important properties of various architectures, where N and N_B is the side in pixels of a quadratic input image and structuring element, respectively.

Design	**PRR** [59]	**LC** [62]	**This work**
SE support	Arbitrary	Rectangular	Rectangular
SE flexibility	No	No	Yes
Complexity	$2\lceil \log_2(N_B) \rceil$	4	N_B
mem [bits]	$(N_B - 1)\, N$	$N \log_2(N_B)$	$N \log_2(N_B) + N_B$
$\mathbf{T}_{exe}$ [CC]	N^2	$N^2 + N_B\, N$	N^2

frames which is supported by most architectures. The complexity refers to the number of operations per pixel (calculation), e.g. in the case of PRR, number of comparators; and in the case of LC, two summations and two additions. The memory requirement is basically the same as for the LC architecture but for the additional N_B delay elements found in the FF-chain. T_{exe} is reported in the number of clock cycles to process a complete frame but does not include the latency, which is present in all architectures. Table 2 indicates that while still maintaining a low memory requirement, the ability to support locally adaptive structuring elements comes at the cost of the complexity increase from 4 to N_B, found in the compare stage as an increased number of comparators, and multiplexers, making the architecture proportional to the maximum supported structuring element width $\max(B_w)$.

6 Conclusion

A novel algorithm for binary ε and δ supporting a locally adaptive structuring element is presented. The memory data is decoupled from the structuring element size, i.e. no data based on the geometrical shape of a former B is stored in memory, which is the property that enables the structuring element flexibility. In order to preserve the raster scan property and avoid unwanted results, algorithmic restrictions on the flexibility is applied.

This part also presents a corresponding hardware architecture, intended to be used as an accelerator in embedded systems. The memory requirement of the architecture is mainly proportional to the I_w while the computational complexity is proportional to the maximum supported structuring element width. The image data is processed in raster scan order without storing the image in memory, especially useful for processing large images. The architecture has been successfully verified on a Xilinx Virtex-II PRO FPGA and implemented as an ASIC in the UMC 0.13 μm CMOS process using a resolution of 640×480 and supporting a maximum structuring element of 63×63. Due to its linear behavior when processing large images, the algorithm is very useful in software applications.

Part III

An Architecture for Calculation of the Distance Transform Based on Mathematical Morphology

Abstract

This part presents a hardware architecture for calculating the city-block and chessboard distance transform on binary images. It is based on applying parallel morphological erosions and adding the results, enabling preservation of the raster pixel scan order and having a well defined execution time. Furthermore, the distance metric to be calculated is controlled by the shape of the structuring element, i.e. diamonds for the city-block and squares for the chessboard. These properties together with a low memory requirement make the architecture applicable in any streaming data real-time embedded system environment with hard timing constraints, e.g. set by the frame rate. Depending on the application, if a priori knowledge of the image content is known, i.e. the maximum size of the clusters, this information can be explored reducing execution time and memory requirement even further. An implementation of the architecture has been verified on an FPGA in an embedded system environment with an image resolution of 320×240 at a frame rate of 25 fps running at 100 MHz. Implementation results when targeted for an ASIC are also included.

Based on: H. Hedberg, and V. Öwall, “An Architecture for Calculation of the Distance Transform Based on Mathematical Morphology,” to be submitted for publication, 2008.

1 Introduction

A typical binary image consists of sets of connected pixels corresponding to foreground and background, i.e. the objects (clusters) of interest and the rest of the image. Taking such a binary image as input, the Distance Transform (DT) calculates the distance from every foreground pixel equal to ONE to the closest background pixel (equal to ZERO), thus transforming the binary image into a gray level image, which can be seen as a topographical surface.

Calculating the DT using morphology was first introduced in [73] and is accomplished by adding the result of multiple erosions and the input image. To calculate the chessboard DT, a flat quadratic B of size 3×3 is used with a binary image I as input, such as the one shown in Figure 1(a). By subtracting the result of an ε using B from the input image, i.e., $I - \varepsilon(I, B)$, the contour of the objects is extracted, as described in Part I, Section 2.4. The clusters' contours correspond to the pixels that are removed in each ε-operation, and an example is illustrated in Figure 1(b). Hence, applying multiple ε-operations using the same B and taking the output from one operation as input to the next according to $\varepsilon(\ldots\varepsilon(\varepsilon(I, B), B)\ldots, B)$, will iteratively remove contour pixels until there are no pixels left. As an example, the input image in Figure 1(a) requires 20 erosions for all pixels to be removed. By adding the original image I with the partial result from each operation, the DT can be calculated as

$$\mathrm{DT} = I + \varepsilon(I, B) + \varepsilon(\varepsilon(I, B), B) + \ldots + \varepsilon(\ldots\varepsilon(\varepsilon(I, B), B)\ldots, B). \qquad (1)$$

By using this procedure, a grayscale image $I_g \in \mathbb{Z}^2$ is created where only the foreground pixels have contributed to the intensity, which directly corresponds to the chessboard DT. For the example in (a), the resulting chessboard DT is illustrated in 2-D in Figure 1(c) and in 3-D in Figure 1(d), where the peaks correspond to pixels with the longest distance to the background. Notice the non-solid object in the middle section that mainly consists of relatively thin parts; this will result in low DT values and hence low intensity values in Figure 1(c) and (d). In order to calculate the city-block metric, simply use diamonds as the shape of the Bs instead of squares in (1).

The DT is present and plays an important role in many digital image processing applications, such as robotics and automation [74]. It can also be used in segmentation applications when calculating the watershed algorithm [75]. The watershed algorithm gets its name from its analogy to a basin after topographic flooding. First, a binary image is created, for instance by thresholding, for which the DT is calculated. The DT result serves as input to the watershed algorithm marking starting points of the flooding, meaning, where the inverted DT has local minimums. When flooding this image, small isolated basins will

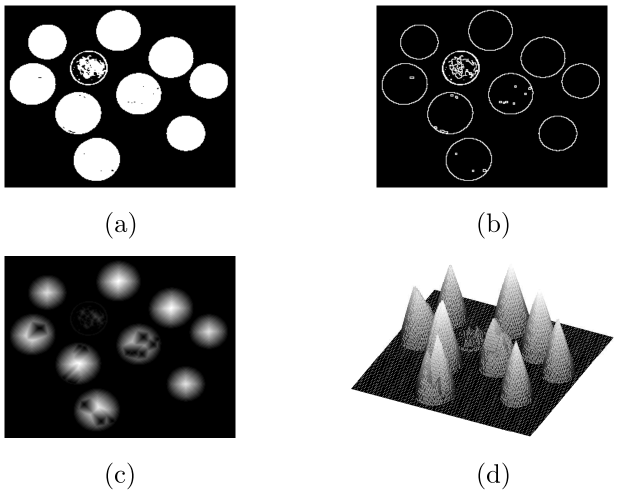


Figure 1: (a) A typical binary image I in which the foreground is represented in white and the background in black. (b) An example of extracted contour pixels when applying a single ε-operation. (c) The chessboard DT for I shown in 2-D. (d) The same DT shown in 3-D slightly rotated clockwise.

form that are spread out over the image; these will eventually merge and become one large basin covering the complete image. The watershed algorithm detects when two or more basins merge and it marks these as regions, creating a regional mask image. Superimposing this mask image on the original binary image, an accurate segmentation result is achieved, which is especially effective when segmenting partially overlapping objects.

The actual distance is measured in units defined by the geometry metric, e.g. Euclidian, chessboard, and city-block, which will be addressed further in Section 2. The choice of distance metric can affect the result and in [75], the impact of using the different metrics when calculating the watershed algorithm is investigated. The conclusion is that the chessboard metric outperforms both the Euclidean and the city-block metric, and has the least erroneous segmentation result. Therefore, the need for efficient algorithms with corresponding hardware architectures calculating the chessboard DT with high-throughput, low memory requirement, and thereby low power properties becomes evident.

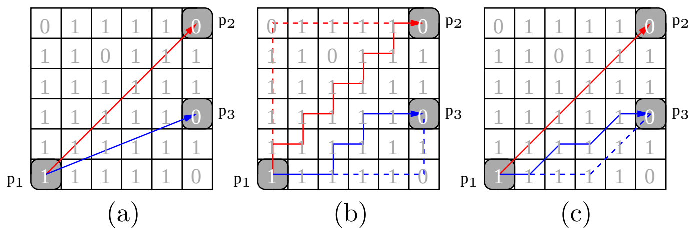


Figure 2: Examples of different geometric distance metrics between two pixel pairs p_1, p_2 and p_1, p_3 all $\in Z^2$. (a) Euclidian, (b) city-block, (c) and chessboard. Notice the dashed lines, which are alternative paths of the same length.

2 Geometry Metrics

For every pair of pixels $p_1(i_1, j_1), p_2(i_2, j_2) \in I$, there exists at least one shortest path. Naturally, the length of the path depends on the level of freedom of allowed directions when moving from one pixel to another, and in any case, the length of this path is equal to the sum of the individual parts. A denominator for all the sets of allowed transitions is that they are geometry metrics. There are basically three geometric metrics (or approximations of these):

- *Euclidian*,
- *City-block*, and
- *Chessboard*.

The Euclidian distance metric is the most intuitive to humans of the three since it corresponds to the distance you measure using a ruler. Mathematically, the Euclidian distance between two pixels $p_1, p_2 \in Z^2$, can be expressed as

$$d_E = \sqrt{(i_1 - i_2)^2 + (j_1 - j_2)^2}. \quad (2)$$

The city-block metric measures the distance as the shortest path between two points through a square lattice in which only orthogonal path transitions are allowed, i.e. no diagonal transitions (4-connected). The path consists of one or several orthogonal line segments which correspond to its projection onto the lattice. Mathematically, the city-block distance between p_1 and p_2 can be expressed as

$$d_4 = |i_1 - i_2| + |j_1 - j_2|. \quad (3)$$

As for the city-block metric, the chessboard distance is measured as the sum of the line segments of a projected path with the addition that diagonal transitions are allowed (8-connected, counted as 1). The chessboard distance between p_1 and p_2 can be expressed as

$$d_8 = \max(|i_1 - i_2|, |j_1 - j_2|). \tag{4}$$

An illustration of how the three metrics maps onto a square grid is shown in Figure 2.

2.1 Previous Research

Assuming raster scan order, the city-block and chessboard DT can be accomplished in two image scans based on Sequential Local Operations (SLO) [76]. A partial result is first propagated in a forward direction (towards the lower right corner of the image), and then in the opposite direction. The final result is obtained by taking the minimum of the forward and backward scans. During the forward scan, the partial result needs to be stored resulting in a large intermediate storage, which is equal to $I_w \times I_h$ times the number of bits required to represent the maximum supported object size in hardware.

The Euclidean DT (EDT) is essentially a global operation, which means that an extensive search has to be performed for every pair of pixels in the image, for every calculated distance. One way to overcome this obstacle and thereby avoid computational complexity is to calculate a less expensive approximation. This is often accomplished by propagating local distances using a sliding window of a particular size, which increases with the accuracy. This procedure is referred to as the chamfer distance and is optimized in [77]. Several hardware implementations of such an approximation can be found in literature [78] [79]. In [80], Takala *et al.* presents a hardware architecture for calculating a variant of such approximation. Due to the well-defined execution time, the architecture is applicable in embedded system environments with raster scan order.

3 Architecture

An illustration of an architecture supporting a single arbitrary B is shown in Figure 3 [81], equivalent to the delay-line architecture discussed in Part I Section 2.1. The architecture mainly consists of four parts: pixel kernel, delay-lines, calculation unit, and a controller. The pixel kernel gives parallel access to the pixels to be included in the current calculation (thus supporting an arbitrary B), implemented with as many flip-flops (ff) as elements in the B. As the B slides over the input image, the pixels are stored waiting to be reused,

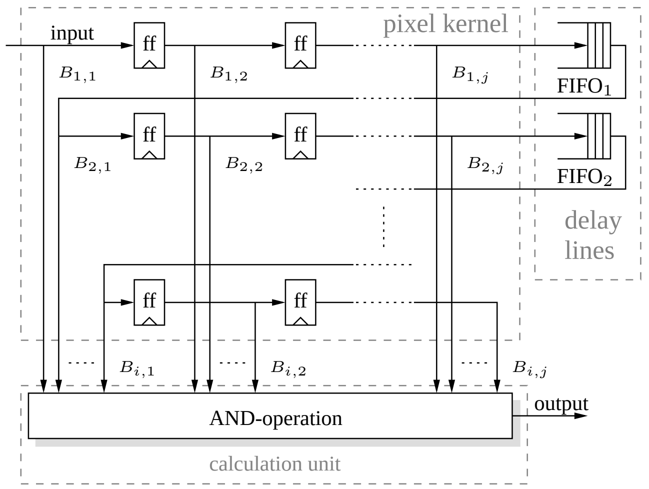


Figure 3: The proposed architecture when supporting a single B. The controller with corresponding signals is omitted for clarity.

accomplished by using delay lines (FIFOs). A logic `AND`-operation is performed on the pixels in the calculation unit, corresponding to an ε, which generates the actual result (to perform a δ, perform a logic `OR`-operation). A controller is required to manage the pixel kernel and the FIFOs as well as the padding; for instance, the controller decides which pixels to exclude from the calculation, as discussed in Section 4.6.

The primary formula to use for calculating the DT is intuitive (discussed in Section 1): apply multiple parallel morphological ε-operations and add the result. This will generate the DT and still preserve the raster scan order. In order to accomplish this, the architecture described in Figure 3 can be utilized but with an extended calculation unit, which requires additional arithmetic, i.e. adders. This is based on the observation that the result of a smaller B_1 is contained in a larger B_2, i.e., if $B_1 \in B_2$, thus partial results can be reused. As an example, the result when using a quadratic B of size 3×3 is contained in the result when using a quadratic B of size 5×5, and can be logically `AND`:ed with the result of the contour pixels of the larger B. Furthermore, since the pixel kernel extracts a certain maximum quadratic pixel neighborhood, all B permutations contained in this neighborhood can be calculated simultaneously. Therefore, the result is the same as having multiple stand-alone units in parallel. The extended calculation unit is shown in Figure 4 where the partial result from each SE size is accumulated in the adders. The depth of the calculation unit

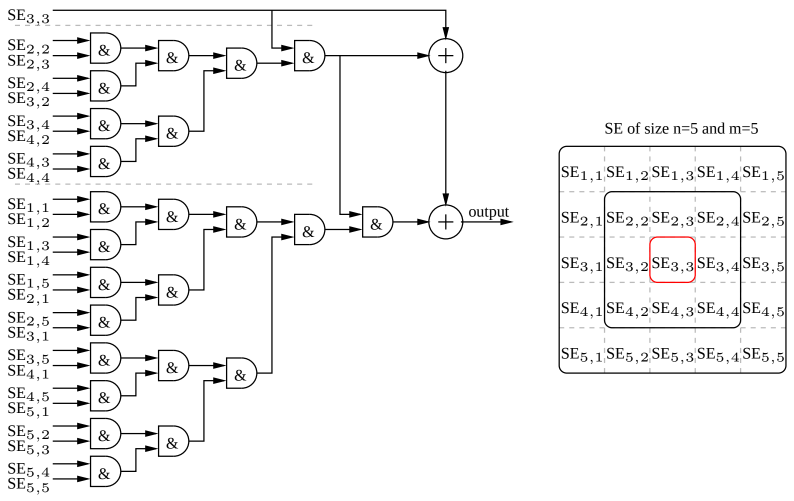


Figure 4: An example of the extended calculation unit when supporting a maximum B of 5×5.

is proportional to the size of the B, i.e. the critical path. In order to avoid a long critical path, pipelining can be explored inferring only latency. Since the logic operations are performed in parallel, a high-throughput is achieved.

Calculating the DT based on the city-block or the chessboard metric is controlled by the shape of the B, as discussed in Section 1. The result is produced directly at the output of the calculation unit, and no further processing is required. However, to produce an approximation of the Euclidean distance metric, the result from the chessboard and the city-block DT can be combined [73], but this operation requires additional arithmetic.

3.1 Execution Time

In the proposed architecture, since the padding is managed by the controller by excluding pixels from the calculation, no stall cycles are required, the result of which is that the architecture operates on the same frequency as the incoming pixels and only latency is inferred. Since each ε operation is calculated in parallel when the entire neighborhood of the maximum supported B has been extracted, the latency is proportional to $\lfloor \frac{N_B}{2} \rfloor$. This is a major improvement over the SLO-based architecture, which has an execution time of two times the resolution due to the forward and backward scans.

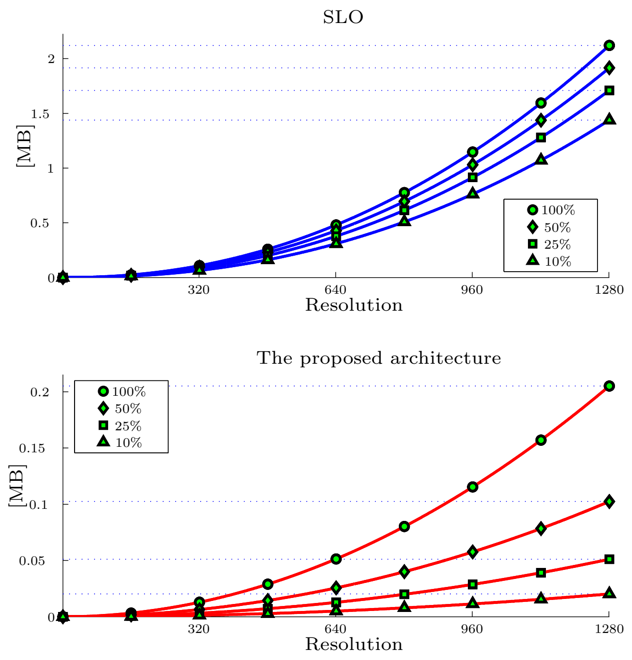


Figure 5: Memory requirement versus resolution (a square) for the SLO and the proposed architecture.

3.2 **Memory Requirement**

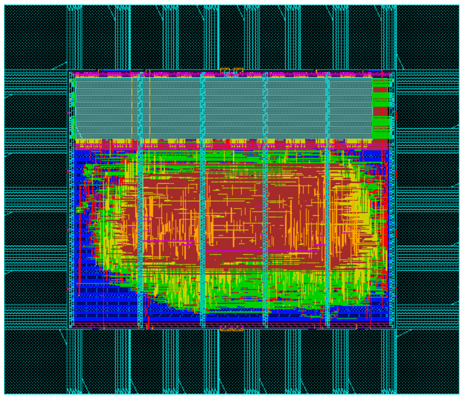
The memory requirement for storing the pixels that are to be reused in future calculations, i.e. the delay lines, is proportional to both the resolution and the B according to

$$\text{mem}_{dl} = (B_h - 1)(I_w - B_w). \tag{5}$$

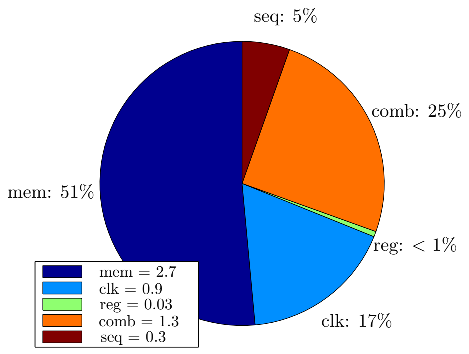
The number of flip-flops in the pixel kernel that holds the current values is proportional dependent of the B size according to

$$\text{mem}_{pk} = B_h \cdot B_w. \tag{6}$$

Therefore, by combining (5) and (6), the total memory requirement for the architecture can be written as $\text{mem}_{tot} = \text{mem}_{dl} + \text{mem}_{pk}$.



(a)



(b)

Figure 6: (a) A screen shot of the final layout. (b) The power distribution in the different parts of the design when running at maximum speed, i.e. $\approx$ 450 MHz. The total power dissipated in the design is 5.2 mW.

The worst case input is an object covering the entire image, the result of which is that the maximum supported B must be equal to the $\max(I_h, I_w)$. However, in many applications a priori knowledge of the image content is known, which can be utilized to reduce the memory requirement. Figure 5 shows the memory requirement versus resolution for the SLO and the proposed architecture. Taken from the graph, using worst case scenario input, the proposed architecture requires $\approx$ 10% of the memory required by the SLO (since only binary values need to be stored). However, allowing a maximum object size of 50%, 25% and 10% of the image resolution, the memory requirement is reduced even further, which is also illustrated in the graph.

4 Implementation Results and Performance

The proposed architecture has been implemented in VHDL and successfully verified on a Xilinx Virtex II-pro FPGA, using a resolution of 320×240, a frame rate of 25 fps, and supporting a maximum B size of 31×31, which is $\approx \frac{I_w}{10}$. If the input image contains objects larger than $\frac{I_w}{10}$, the distance values become saturated. The architecture has also been implemented in the UMC 0.13 ASIC technology and results in the form of resource utilization characteristics for both technologies are compiled in Table 1, where the area is reported excluding IO-pads (Input Output), and the gate count is based

Table 1: Implementation resource utilization characteristics of a Xilinx Virtex II-PRO FPGA and in the UMC 0.13 μm CMOS process using a resolution of 320×240 and supporting a flexible SE up to 31×31.

FPGA	used	available	**ASIC**	used
Slices	2150	13696	**Area** [mm^2]	0.20
LUTs	2706	27392	**Gate count** [k]	39
Block RAM	1	136	**mem**$_{tot}$ [kb]	9.63
Speed [MHz]	100	—	**Speed** [MHz]	454

on a 2-input NAND-gate (5.12 μm^2) and includes all memory blocks. The final layout of the architecture is illustrated in Figure 6(a). Note that the primary use of the architecture is not as a stand-alone ASIC but rather as a hardware accelerator in an embedded real-time image processing system.

The architectures in [80] [81], report a similar hardware resource utilization in terms of memory requirement, since they both use delay lines to extract the neighborhood. However, [80] is not based on morphology and calculates an approximation of the EDT, and [81] only supports calculation of a single erosion per extracted neighborhood.

The result of a post-layout power simulation, using Primetime provided by Synopsys [82], is shown in Figure 6(b). The total power P_{tot}, defined in Section 2.4, dissipated in the architecture is 5.2 mW, when running at maximum speed. As can be seen in the power distribution chart, the major part is consumed in the memory followed by the part that is dissipated in the combinatorial logic, mainly corresponding to the calculation unit.

5 Conclusion

A flexible HW architecture for calculating the DT based on mathematical morphology is presented. The geometry metric to be calculated, i.e. city-block or chessboard, is controlled by using either diamonds or squares as structuring elements, thus still preserving the important raster scan order. Comparing the architecture with a reference design (two-scan SLO), a speed increase of a factor of two is gained together with a reduced memory requirement making the architecture applicable in any real-time streaming data environment. Implementation results of the architecture in the form of resource utilization characteristics when targeted for both FPGA and ASIC are also included.

Part IV

Implementation of Labeling Algorithm Based on Contour Tracing with Feature Extraction

Abstract

An evaluation from a hardware perspective of the two types of connected-component labeling algorithms introduced in Chapter 5 is presented. Based on this evaluation, the CT-based algorithm was found most suitable in the target application and therefore chosen for implementation. The implementation is intended as a hardware accelerator in a self-contained, real-time digital surveillance system. The algorithm has lower memory requirements as compared to other labeling techniques and can guarantee labeling of a predefined number of clusters independent of their shape. In addition, features especially important for this particular application are extracted during the contour tracing with little increase in hardware complexity. The implementation is verified on an FPGA in an embedded system environment with an image resolution of 320×240 at a frame rate of 25 fps. The implementation supports labeling of 61 independent clusters, extracting their location, size, and center of gravity.

Based on: H. Hedberg, F. Kristensen, and V. Öwall, "Implementation of Labeling Algorithm Based on Contour Tracing with Feature Extraction," in *Proc. ISCAS 2007*, New Orleans, USA, May 28-31 2007.

1 Introduction

In an automated surveillance system, the aim is to track and classify objects present in the scene, and to do so without human interaction. One of the first steps in such a system is to detect the objects of interest, which is performed by segmentation [3]. After segmentation, a binary frame is produced containing clusters of pixels that represent different objects present in the scene. These pixels are referred to as foreground and the remaining are background. Assuming that noise has been removed by some pre-processing step (morphological filter), the frame only contains clusters of interest that are to be tracked and classified. However, before any tracking and classification are possible, the system has to be able to separate and extract features of the clusters, which is accomplished by labeling. The goal is to label the clusters and to separate them in memory for further processing. This procedure gives the possibility to tie features to each cluster (label). A feature is a figure that reveals a property of the cluster on which tracking and classification are based. Thus, labeling can be seen as the link between the clusters and their features. During the labeling process, as many cluster features as possible are extracted based on the binary representation of the objects, e.g. position (coordinates) and size. In addition, the labeled mask is used to extract color features. By superimposing the mask on the color video stream, valid pixels can be cut out and used to calculate the color features, such as color mean value, and color histogram. The extracted features are then used by the system to monitor many object properties revelent in surveillance applications: object trajectories, appearances, and disappearances.

2 Hardware Aspects of Labeling Algorithms

To choose one of the two types of algorithms introduced in Chapter 5 for implementation, i.e. SLO- and CT-based algorithms, important properties relevant to our application needs to be compared and evaluated; these properties are complexity, throughput, memory requirements, and extracted features. The evaluation of the two types of algorithms is compiled in the subsequent sections.

2.1 Complexity

Regarding the algorithmic complexity of SLO-based and CT-based algorithms, neither of them require any advanced arithmetic operations to perform the actual labeling. The main part of the complexity in both algorithms is due to memory handling. In the case of two-scan SLO, the controller extracting the pixels covered by the scan mask is simple due to the regular memory ac-

Table 1: Simulations on three independent sequences showing the number of clusters and corresponding label collisions.

Sequence	**Seq. 1**	**Seq. 2**	**Seq. 3**
Mean clusters per frame	4.19	6.98	6.56
Mean label$_{col}$ per frame	13.1	6.72	14.3
Max clusters in a frame ($c_{\max}$)	14	20	20
Max label$_{col}$ in a frame($l_{\mathrm{c,max}}$)	27	21	50
Resolution	320×240	352×288	768×576
Nbr. of frames in the seq.	700	900	2500

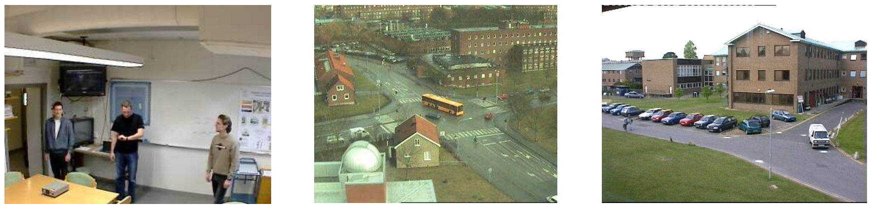
cess pattern. However, a more complex controller is required to manage label ambiguities due to label collisions. The controller needs to support a merge and search operation, which may be implemented using the union-find algorithm [50]. In CT-based algorithms, the most complex block controls the CT phase which manages the search process for contour pixels. No major advantage in terms of computational complexity can be seen in any of the two algorithms.

2.2 Memory Requirements

Both types of algorithms need a memory to store the labeled image result mem$_{label}$. Due to the physical limitations of this memory, an upper bound is placed on the number of clusters that can be labeled in a frame $c_{\max}$. In SLO-based algorithms, each label collision will occupy a temporary label during the initial scan. Thus, the memory size is determined by a combination of $c_{\max}$ and the maximum number of label collisions $l_{\mathrm{c,max}}$. This results in a memory overhead in the form of additional required bits per stored pixel. Assuming that the memory required to manage the label collisions is small compared to mem$_{label}$ and can therefore be neglected, the memory requirement for the SLO-based algorithms may be approximated as

$$\mathrm{mem}_{SLO} = \lceil \log_2(c_{\max} + l_{\mathrm{c,max}} + 1) \rceil \cdot N^2 \text{ bits}, \tag{1}$$

where $N^2 = (I_h \times I_w)$ is the number of pixels in an image and the +1 comes from the fact that 0 is a preoccupied label representing the space in between clusters or holes inside clusters. In CT-based algorithms, the memory size is directly proportional to the image resolution and $c_{\max}$ since no label collisions occur. Therefore, the memory requirement can be written as



(a) (b) (c)

Figure 1: (a) A frame from sequence 1, taken in our research laboratory, (b) a frame from sequence 2, covering a traffic crossing, (c) and a frame from sequence 3, taken from the PETS database [83], covering a parking lot.

$$\text{mem}_{CT} = \lceil \log_2(c_{\max} + 3) \rceil \cdot N^2 \text{ bits,} \tag{2}$$

where +3 comes from the fact that 0, 1, and 2 are preoccupied labels; 0 is used to represent the space in between clusters or holes inside clusters, 1 is used to represent the pixels within a cluster (intermediate), and 2 is used for the reserved label l_r.

To decide how much memory to assign to the algorithm for a certain application, a good strategy is to run a simulation on typical input, e.g. a sequence from the surveilled scene. A simulation reveals important cluster data, e.g. mean/maximum number of clusters and label collisions per frame. This data may be used to support the decision of setting $c_{\max}$ and $l_{\mathrm{c,max}}$ to actual values. Table 1 compiles three simulations of such sequences that show the number of clusters with corresponding label collisions per frame. Sequence 1 is captured in our research laboratory, Sequence 2 is captured outdoors covering a traffic crossing, and sequence 3 is taken from the PETS database [83]. A typical frame from each of the three sequences is shown in Figure 1. The figures in Table 1 show that SLO-based algorithms would require 6, 6, 7 bits per pixel compared to CT-based algorithms, which would require 5, 5, 5 to handle the worst case scenario in all three sequences. Having identified and set these values, the memory requirement for both types of algorithms can be compared using (1) and (2). This shows that SLO-based algorithms requires at least additional N^2 bits of memory for every extra bit per pixel compared to the CT-based algorithm. As an example, with a resolution of 320×240 and supporting sequence 2 in Table 1, $c_{\max}$ is set to ≈ 64. "Approximately" is used instead of "equal" due to preoccupied labels as well as the width of the label memory being adjusted to the closest power of two. Using these settings, CT-based algorithms requires $\approx 14\%$ less memory than the SLO-based algorithm

Table 2: Simulation results of the number of memory accesses required by an SLO-based algorithm.

Sequence	**Seq. 1**	**Seq. 2**	**Seq. 3**
Mean $\sum size_i$ per frame	349	363	802
Max $\sum size_i$ clusters in a frame	2959	2325	5403
Mean nbr of mem$_{access,SLO}$ per frame [k]	154	203	886
Max nbr of mem$_{access,SLO}$ in a frame [k]	157	205	890
Nbr. of frames in the seq.	700	900	2500

according to

$$\frac{\text{mem}_{CT}}{\text{mem}_{SLO}} = 1 - \frac{\lceil log_2(c_{\max}+3)\rceil}{\lceil log_2(c_{\max}+l_{c,\max}+1)\rceil)} = 1 - \frac{6}{7} \approx 0.14.$$

In general, as shown in the example sequences in Table 1, CT-based algorithms require less memory as compared to SLO-based algorithms to be able to label the same number of clusters.

2.3 Memory Accesses

The required number of memory accesses for the two-scan SLO-based algorithm to complete the labeling procedure consists of several parts. Assuming raster scan order, the input data is first written into mem$_{label}$ and assigned the preliminary labels during the initial scan, resulting in N^2 write operations. In addition, possible label ambiguities have to be written into an equivalence table. However, the memory operations needed to maintain the equivalence table may be neglected since they are $\ll N^2$. The initial scan is followed by a second read scan with an additional N^2 read operations and possible write operations to resolve the label ambiguities. Let $size_i$ be the remainder of a cluster in pixels that need to be relabeled. This results in additional $size_i$ write operations for this particular cluster. The total number of memory accesses for a frame containing K clusters, each with $size_i$ pixels that needs to be relabeled, can be written as

$$\text{mem}_{access,SLO} = 2 \cdot N^2 + \sum_{i=1}^{K} size_i. \tag{3}$$

Simulation results are compiled in Table 2, which shows the number of memory accesses for each of the previously described sequences. As can be seen from

Table 3: Simulation results of the required number of memory accesses for a CT-based algorithm.

Sequence	**Seq. 1**	**Seq. 2**	**Seq. 3**
Mean $\sum p_i$ per frame	754	372	694
Max $\sum p_i$ clusters in a frame	1364	939	1479
Mean $\sum l_j$ per frame	574	204	404
Max $\sum l_j$ clusters in a frame	1026	570	949
Mean nbr of mem$_{access,CT}$ per frame [k]	156	204	887
Max nbr of mem$_{access,CT}$ in a frame [k]	158	205	889
Nbr. of frames in the seq.	700	900	2500

these figures, the total number of pixels that needs to relabeled $\sum size_i$ is small compared to the number of memory accesses due to the two global scans. This is typically the case since the objects present in the scene are usually small compared to the resolution (otherwise the camera is too close). Note that each sequence is captured in a different resolution.

Concerning the number of required memory accesses for the CT-based algorithm: first, the input stream is written into mem$_{label}$ resulting in N^2 write operations. This is followed by a second scan during which the CT takes place, which has an upper limit of N^2 read operations. For every cluster, tracing the contour and writing the reserved label on both sides of the cluster requires additional read and write operations. If a frame contains K clusters, each with p_i contour pixels and the need to write l_j reserved labels along each side, the total number of memory accesses can be written as

$$\text{mem}_{access,CT} = 2 \cdot N^2 + \alpha \sum_{i=1}^{K} p_i + \sum_{j=1}^{K} l_j. \tag{4}$$

where α is the average number of memory accesses needed in the CT phase to find the next contour pixel and $\sum p_i$ and $\sum l_j$ are the total number of contour pixels and reserved labels in a frame. Table 3 compiles simulation results of the CT-based algorithm for each of the three sequences. Using these figures and $\alpha \approx 2.2$ (simulated value), the mean and maximum number of accesses per frame can be calculated using (4). Taken from the figures, sequence 1 contains larger objects which can be seen as a larger number of contour pixels $\sum p_i$ compared to the other two. However, as for the simulation in Table 2, the main part of the memory accesses are due to the two global scans. Note that neither the holes nor the intermediate pixels are filled or assigned the correct label in any of the simulations.

Comparing the results in Tables 2 and 3, no major performance advantage may be distinguished between the software models in terms of memory access requirements (for these three sequences). Note that the optimization marking the start and end pixels discussed in Section 5.4, is not included in either of the simulations compiled in Tables 2 and 3, but would affect the first part of (3) and (4) equally in both types of algorithms.

2.4 **Execution Time**

The total execution time t_{exe} (number of clock cycles), for both SLO- and CT-based algorithms is determined by the number of memory accesses, which in turn are proportional to the complexity of the input clusters; a complex contour results in more read and write operations, i.e. affects $size_i$ in (3) and both p_i and l_j in (4). To be able to determine an upper limit on t_{exe}, clusters with worst case contours have to be constructed, which can be hard to prove theoretically. However, for SLO-based algorithms, one of the worst case scenarios is a cluster covering the complete frame with a label collision in the upper left corner. This means that almost the complete cluster needs to be relabeled, resulting in an upper limit of the latter part of (3), i.e. $\sum_{k=1}^{K} size_i \leq N^2$. An upper limit on the total amount of required memory accesses per frame can therefore be written as $\leq 3 \cdot N^2$. Creating an image causing the maximum number of memory accesses in the CT is even harder and is most likely to be artificial, e.g. elongated horizontal clusters covering the whole frame. Instead of evaluating such synthetic input frames, the latter part of (4) is left with the assumption that the maximum number of memory accesses needed by the CT phase cannot exceed the image size, i.e. $\sum_{i=i}^{K} p_i + \sum_{j=1}^{K} l_j \leq N^2$. Hence, when comparing the upper limit on the total execution time for the two types of algorithms, both can be written as $t_{exe} \leq 3 \cdot N^2$.

2.5 **Feature Extraction**

From a system perspective, it is desirable to extract features where they have low requirements in terms of execution time and hardware complexity, since extracting them during post-processing can strain the timing budget. There are several features of binary clusters that are advantageously extracted in a labeling unit since all foreground pixels are visited at least once and since it has information pertaining to which cluster each pixel belongs to. Furthermore, many of these features can be extracted in a similar manner for both SLO- and CT-based labeling algorithms, e.g. maximum / minimum coordinates and area, which can be done by updating or sequentially incrementing registers.

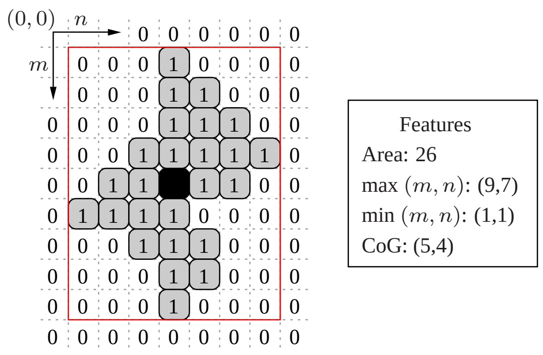


Figure 2: An example of an arbitrary cluster for which cluster features have been extracted. The center of gravity is marked in black and a bounding box encapsulates the cluster base on the maximum and minimum coordinates.

For example, the maximum and minimum coordinates can be used to draw a bounding box around each object, which is often used in surveillance applications, as illustrated in Figure 2. To extract more advanced features such as Center of Gravity (CoG), which is an important feature in a surveillance application, used by the tracking unit to match clusters in consecutive frames and to handle occlusion, more complex arithmetic is required, such as multipliers and dividers. In SLO-based algorithms, a straightforward method to extract CoG for a cluster C_k, is to store both the total number of pixels in this cluster $C_{k,a}$, and the sum of the coordinates in each direction, i.e. $C_{k,m}$ and $C_{k,n}$. If a cluster consists of several clusters (contains label collisions), simply merge each respective sum. Using this notation, calculating CoG in each direction, i.e. $m_{k,CoG}$ and $n_{k,CoG}$, can be expressed as

$$m_{k,CoG} = \frac{C_{k,m}}{C_{k,a}}, \text{ and } n_{k,CoG} = \frac{C_{k,n}}{C_{k,a}}.$$

An example is illustrated in Figure 2, in which $C_{k,m} = 129$ and $C_{k,n} = 108$, resulting in a CoG located in $(m_{k,CoG}, n_{k,CoG}) = (5, 4)$.

CT-based algorithms offer a more refined way to calculate CoG since they have the possibility to add discrete Green's theorem to the CT phase. Green's theorem gives the relationship between a closed contour (curve) and a double integral over this cluster (plane region), enabling calculation of the moments [84]. The moments can in turn be used to calculate the CoG (and be an alternative way to calculate the area). In order to calculate moments, background to foreground transitions ($p_i = 1$, $p_j = 0$, $p_j \in N_8(p_i)$ have to be determined for

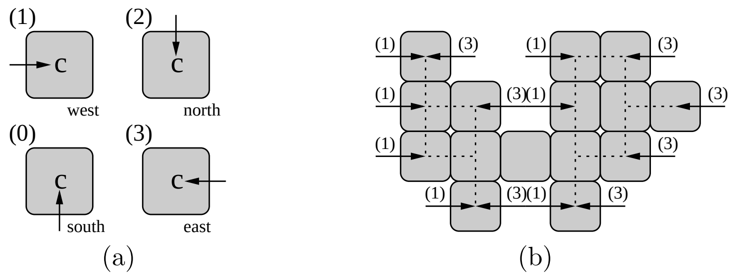


Figure 3: (a) The four possible pixel-to-cluster transitions, (b) and an example of a cluster with corresponding west and east background-to-foreground transitions. Note that vertical transitions entering pixels from the top and bottom are omitted from the figure since they do not contribute in (5).

each cluster. There are four possible background to foreground transitions to a contour pixel: south, west, north, and east, as illustrated in Figure 3(a). Each transition is associated with a triplet $(m, n, \Delta m)$, where (m, n) are the contour pixel coordinates and Δm gives directional information, and is defined as -1 and 1 for the west and east transition, and 0 for the remaining two [84]. The triplets are used to compute the monomial in the sum

$$u_{i,j} = \sum_{contour} n^{i+1} m^j \Delta m, \tag{5}$$

in order to calculate u_{00}, u_{10}, and u_{01}, which are updated for every contour pixel. Note that u_{11} is omitted since it is not required in any further calculations. Since $\Delta m = 0$ for both the south and north transitions, they do not contribute in (5), and the transitions are reduced to only west and east, as illustrated in figure Figure 3(b). The sums are used to calculate the moments according to

$$m_{00} = u_{00},\ m_{10} = \frac{u_{00}}{2} + \frac{u_{10}}{2},\ m_{01} = u_{01}. \tag{6}$$

After the moments have been calculated, they are transformed into area and CoG according to

$$C_{k,a} = m_{00},\ n_{k,CoG} = \frac{m_{10}}{m_{00}},\ m_{k,CoG} = \frac{m_{01}}{m_{00}}. \tag{7}$$

A division is needed to calculate CoG, which is the most complex arithmetic operation in this unit. However, this is not a major disadvantage since a divider is required for SLO-based algorithms to calculate CoG as well; refer to (5).

Applying (5) on the example illustrated in Figure 2, starting on the left side of the top pixel and tracing the contour clockwise generates a transition sequence of 133333333311111111, using the definition in Figure 3. This sequence is translated into a Δ sequence equal to $(-1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, -1, -1)$. The Δ sequence is used together with the coordinates for each Δ in each direction in (5), generating $u_{00} = 26$, $u_{10} = 180$, and $u_{01} = 130$. Using this result in (6), gives the moments $m_{00} = 26$ $m_{10} = 103$ and $m_{01} = 130$, which when applied to (7), generates a CoG equal to $(m_{k,CoG}, m_{k,CoG}) = (5, 4)$.

Higher order moments can be used to extract the axis of orientation. However, this feature is currently not used by our tracking algorithm and is therefore not considered.

3 Algorithm Evaluation

The preceding evaluation can be compiled into the following:

Memory requirement: The CT-based algorithms can guarantee the labeling of a specific number of clusters $c_{\max}$, and therefore they require less memory.

Memory accesses and Execution time: In their original forms, both type of algorithms have the same upper bound on the total execution time, i.e. $t_{exe} \leq 3 \cdot N^2$. However, modifying the read-out scheme for SLO-based algorithms, this type of algorithm will have an advantage over CT-based algorithms.

Features: Both types of algorithms can extract the same features, e.g., maximum and minimum coordinates, area, and CoG.

As it becomes evident in the list, both types of algorithms have pros and cons. To summarize: SLO-based algorithms are fast, and the regular data access patterns permits burst reads from memory but have the drawback of label collisions making the number of labeled clusters per frame undefined. A major advantage of CT-based algorithms is that they can guarantee labeling of a specific number of clusters but have the drawback of random memory accesses.

Some applications do not require unique labels to be assigned to each cluster but rather have the restriction that each cluster may be visited only once; for instance, when counting the clusters. In other applications it can be sufficient to extract correct binary feature extraction for each cluster assuming that all pixels within the same bounding box are part of the same object. This can be explored in an automated surveillance system and is discussed further in Part V of this thesis. In such applications, the CT-based algorithms are superior in

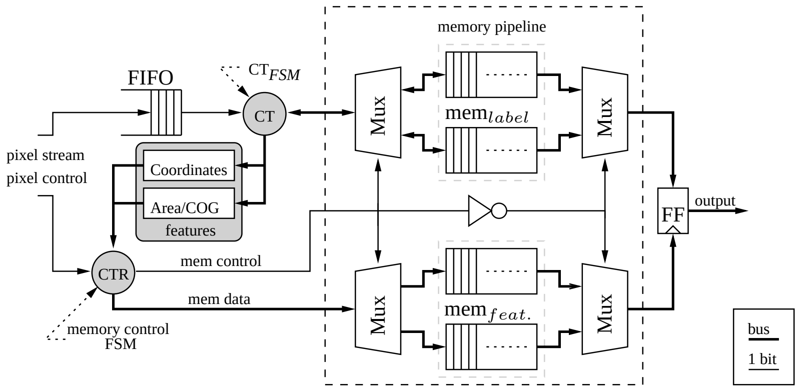


Figure 4: Overview of the implemented architecture.

terms of memory requirement over SLO-based algorithms. This comes from the fact that in CT-based algorithms, the label memory can be reduced to 2 bits per pixel, still being able to mark each cluster as visited and correctly extract their binary features and place them in separate entries in the feature memory. On the other hand, in SLO-based algorithms, the memory cannot be reduced since this type of algorithm still faces the problem of label collisions.

4 Implementation

Based on the Due to the lower memory requirements, the CT-based algorithm was chosen for implementation and an overview of the architecture is shown in Figure 4. A FIFO is needed at the input as the data stream is stalled when a frame is being labeled. The CT_{FSM} is the main block in which the actual contour tracing takes place, and its functionality is illustrated as a block diagram in Figure 5.5. Assuming labeling of 8-connected clusters, for the CT_{FSM} to be able to find the next contour pixel, a matrix is inferred marking the direction from a pixel p to its eight neighbors, $N_8(p)$, as shown in Figure 5(a), similar to what is used in a *chain code* [2]. A variable d gives the direction from the last visited to the current contour pixel. Since the global memory scan is performed in raster scan order, the default direction is right, i.e. 1 in the matrix. d is used as input to a LUT, containing the direction to start the search for the next consecutive contour pixel; hence it is referred to as the initial search LUT and is illustrated in Figure 5(b). The second LUT contains the number to add to the current coordinates to reach a pixel for each direction; this is referred to as the address LUT and is illustrated in Figure 5(c). If the next contour

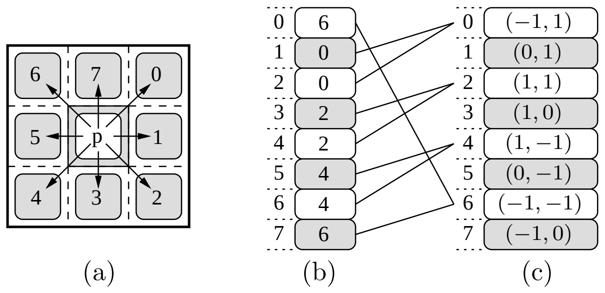


Figure 5: (a) Definition of the direction matrix, (b) the LUT containing the start direction to search for the next contour pixel, (c) and the LUT containing the number to add to the current coordinates (address) for each direction in (a). The connectors between (b) and (c) indicate that the output from the initial search LUT is used as an index to the address LUT.

pixel is found, the current label is written into mem$_{label}$ at this coordinate, d is updated, and the search starts over in the direction given by the initial search LUT with d as the index. If there is a miss, the direction is incremented clockwise and then used as the index in the address LUT. The output from the address LUT is added to the current coordinates, which now point to the next possible contour pixel, which repeats until the next contour pixel is reached. When the local starting pixel is reached a second time, it means that the contour of this cluster is completely traced and the global scan continues. Note that the index to the address LUT is increased in a *modulo*-8 fashion, which means that when the index reaches 8, it starts over at zero.

An illustration of the contour tracing of a small cluster with corresponding search directions is shown in Figure 6(a). In (b), the pixel is marked as the local start pixel and assigned the current label, l_1. Since the default direction is 1, a lookup is made in the initial search LUT, which outputs a 0. This 0 is used as the index in the address LUT, which outputs $(-1, 1)$, which is added to the current address to start searching for the next contour pixel. Since the next contour pixel is not located at this address, the index to the address LUT is increased by 1, and the next number to add to the current address is $(0, 1)$, which is now pointing to the left of the current pixel. Since the next contour pixel is found on this address, the current address is updated to this pixel together with that it is assigned the appropriate label. To start the search for the next consecutive contour pixel, since the current pixel was found in a direction equal to 1, a new lookup in the initial search LUT is made, again receiving a 0 which is used to make a new lookup in the address LUT. If a

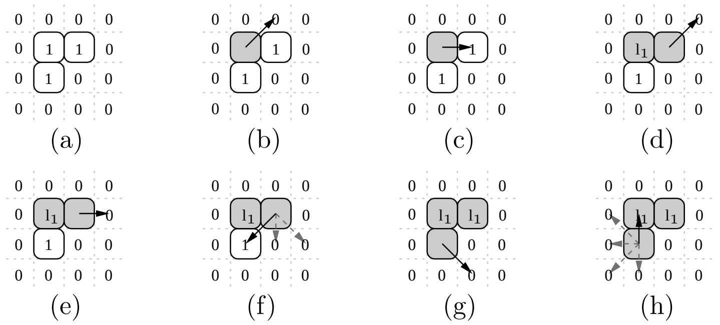


Figure 6: (a) A small cluster that is to be contour traced, (b) the start pixel is marked and assigned a label, l_1. The arrow indicates the initial search direction for the next contour pixel which is found in (c). (d) The current address is updated and the arrow indicates the new initial search direction for the next contour pixel. (e) The search direction is increased, until a new contour pixel is found in (f). (g) Again, the address is updated, the pixel is labeled, and the search starts in the direction indicated by the arrow. (h) The search direction is increased until the last contour pixel is found, ending the contour tracing of this cluster. Note that the gray dashed arrows indicate search misses.

new contour pixel is not found on the current search address, the index to the address LUT is increased by 1. This process repeats until the start pixel is reached, indicating that all contour pixels of this particular cluster have been labeled and the image scan continues searching for the next unlabeled cluster.

During the CT phase, the CT$_{FSM}$ produces a triplet of control signals for every contour pixel, i.e. $(m, n, \Delta m)$. The triplet is sent as input to the feature extraction blocks which calculate maximum / minimum coordinates, size, and CoG, for every cluster. In turn, the features are sent to the control block (CTR) which aligns, concatenates, and writes them into mem$_{feat.}$, in which each address corresponds to a specific label. The output of the unit is the memory content, and to maximize the time the embedded tracking SW can access this result, a memory pipeline is inferred. The memory pipeline is implemented with multiplexors and duplicated memories as shown in Figure 4. Hence, as the algorithm is labeling one frame, i.e. accessing one memory pair, the unit gives access to the other through an HW-interface consisting of SW addressable registers from the embedded Power PC.

Table 4: Implementation and resource utilization characteristics.

Resolution	320×240	**Nbr. clusters**	61
$\mathbf{Mem}_{tot}$[kbit]	446	**Frame rate**	25
LUTs	2229 (8.1%)	**Speed** [MHz]	67

5 Results and Performance

Based on the simulations in Section 2 and due to memory limitations in the FPGA, $c_{\max}$ is set to 61 (plus three preoccupied labels), resulting in 6 bits per pixel in mem_{label}. Using (2) and a resolution of 320×240, the total size of a single mem_{label} is 461 kb. Furthermore, $c_{\max}$, the resolution, and number of cluster features determines the size of $\text{mem}_{feat.}$, since $c_{\max}$ determines the depth and each word consists of the maximum and minimum coordinates, cluster area, and CoG. Using these settings, the total size of the cluster memory can be written as

$$\begin{aligned}\text{mem}_{feat.} = (c_{\max} + 3) \cdot (&\lceil log_2(I_h)\rceil + \lceil log_2(I_w)\rceil + \lceil log_2(I_h)\rceil + \\ &+ \lceil log_2(I_w)\rceil + \lceil log_2(I_h)\rceil + \lceil log_2(I_w)\rceil + \lceil log_2(I_h)\rceil \\ &+ \lceil log_2(I_w)\rceil) \approx 4.3 \text{ kb.}\end{aligned} \tag{8}$$

Based on (2) and the fact that the input FIFO is set to be ≈ 76.8 kb to store a complete frame at the current frame rate, the total memory requirement is

$$\text{mem}_{tot} = FIFO + 2 \cdot (\text{mem}_{label} + \text{mem}_{feat.}) \approx 1.01 \text{ Mb},$$

where the factor 2 is the memory pipeline. Note that the actual size of input FIFO is set to ≈ 130 kb, due to limited size support in the EDA tool, i.e. Core Generator provided by Xilinx [85].

As mentioned in Section 3, the target application of this unit does not require unique labels to be assigned to the clusters but rather only that the clusters have separate entries in $\text{mem}_{feat.}$. This means that a single label may be used without system performance degradation, since this does not affect cluster or color feature extraction; cluster features are extracted during contour tracing, and color features are extracted on single objects even before the BB of another object overlap. In either case, label ambiguities are avoided. The result is that the label memory only uses two bits per pixel, and the complete label unit requires 446 kbit instead of 1.01 Mbit.

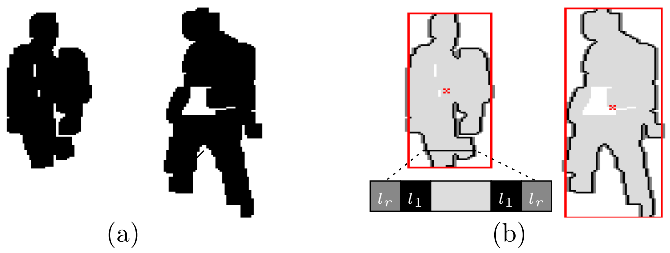


Figure 7: (a) A fragment of a typical binary input frame to the label unit, (b) corresponding labeled output. Notice the gray pixels on each side of the contour which corresponds to the reserved label, and the hole in the middle cluster.

The architecture is implemented in VHDL on a Xilinx Virtex-II pro FPGA (XC2VP30-7FF896), and the most important implementation results are summarized in Table 4. An example of a typical binary input frame taken from sequence 1 in Table 1 containing two human silhouettes is shown in Figure 7(a), with corresponding labeled output from the implemented architecture shown in (b).

6 Conclusion

An evaluation of SLO-based and CT-based algorithms for connected-cluster labeling in binary images from a hardware perspective is presented. Due to the lower memory requirements in our target application, the CT-based type of algorithm is chosen for implementation. Therefore, implementation results of such a hardware accelerator to be used in an automated surveillance system are also included. The unit labels 61 clusters, and extracts their coordinates, size, and CoG. The algorithm introduces no memory overhead, as opposed to SLO-based approaches, and can guarantee labeling of a specific number of clusters. The memory requirement is 446 kbit, which is less than SLO-based algorithms and is proportional to the image size and the maximum number of individual clusters that can be labeled per frame. The design has been successfully implemented and verified on an FPGA running at 67 MHz in a system environment with an image resolution of 320×240 and a frame rate of 25 fps.

Part V

An Embedded Real-Time Surveillance System: Implementation and Evaluation

Abstract

This part presents the design of an embedded automated digital video surveillance system with real-time performance. Hardware accelerators for video segmentation, morphological operations, labeling and feature extraction are required to achieve real-time performance, while tracking will be handled in software running on an embedded processor. By implementing a complete embedded system, bottlenecks in computational complexity and memory requirements can be identified and addressed. Accordingly, a memory reduction scheme for the video segmentation unit, reducing bandwidth by more than 70% and a low complexity morphology architecture that only requires memory proportional to the input image width, have been developed. On a system level, it is shown that a labeling unit based on a contour tracing technique does not require unique labels, resulting in a memory reduction greater than 50%. The hardware accelerators provide the tracking software with image objects properties, i.e. features, thereby decoupling the tracking algorithm from the image stream. A prototype of the embedded system is running in real-time, 25 fps, on an FPGA development board. Furthermore, the system scalability for higher image resolution is evaluated.

Based on: F. Kristensen, H. Hedberg, H. Jiang, P. Nilsson, and V. Öwall, "An Embedded Real-Time Surveillance System: Implementation and Evaluation," accepted for publication in *Journal of VLSI Signal Processing Systems*, Springer.

1 Introduction

The demands on video surveillance systems are rapidly increasing regarding parameters such as frame rate and resolution. Furthermore, with an ever increasing data rate and number of video streams, an automated process for extracting relevant information is required. Due to the large amount of input data and the computational complexity of the algorithms, software implementations are not sufficient to sustain real-time performance for reasonable resolution. In this part, an automated digital surveillance system running on an embedded platform in real-time is presented. Algorithms that are well suited for hardware implementation with streamlined dataflow are chosen and dedicated hardware accelerators have been developed. The presented hardware platform has been developed with the goal of presenting a proof of concept for the surveillance system and to identify computational and memory bottlenecks. Furthermore, when proposing modifications to the original algorithms, extensive simulations are needed, especially if long-term effects in the video sequences can be envisioned. Utilizing a reconfigurable platform based on an FPGA reduces the simulation and development time considerably.

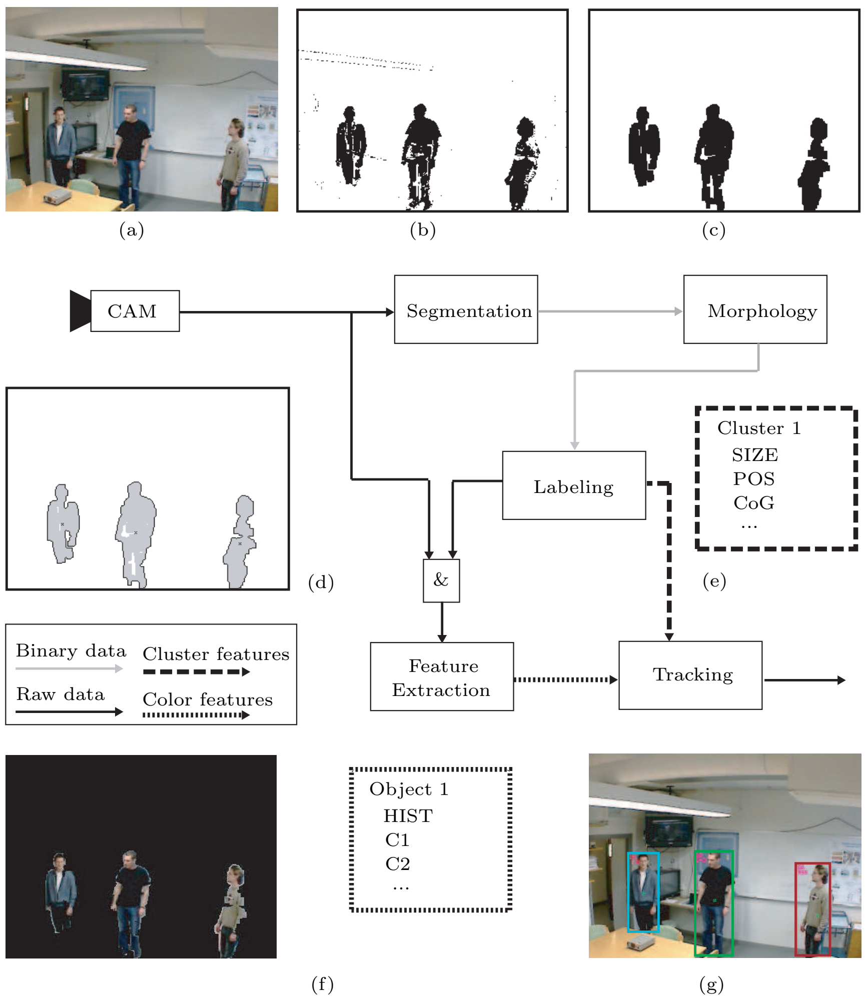


Figure 1: Surveillance system, (a) original image, (b) binary motion mask, (c) morphologically filtered motion mask, (d)-(e) Labeled clusters and cluster features, (f) detected objects and color features, and (g) tracking results. All tracked objects are marked with a uniquely colored frame as long as the objects are present in the scene.

A conceptual overview of the surveillance system is shown in Figure 1. The camera feeds the image processing system with a real-time image stream of 25 fps, Figure 1(a). A segmentation algorithm, in this case based on a Gaussian Mixture background Model (GMM), preprocesses the image stream and produces a binary mask in which background and foreground are separated, Figure 1(b). In theory, only the moving parts of an image should be distinguished as independent objects in the binary mask. However, in reality the mask will be distorted with noise and single objects are shattered. In order to remove noise and reconnect split objects, morphological operations are performed on the mask, Figure 1(c). These morphological operations will produce a frame of connected clusters which have to be identified, i.e. labeled. The labeled clusters together with extracted cluster features, e.g. size and position, are seen in Figure 1(d) and (e). Foreground objects, which have been cut out from the original frame, with corresponding color features are shown in Figure 1(f). In the final image, tracked objects are identified by uniquely colored bounding boxes, Figure 1(g).

The main bottleneck in image processing algorithms is the high memory requirements that are imposed on the hardware system, both in terms of size and bandwidth. In this work, bandwidth reduction has primarily been addressed in the segmentation unit, using wordlength reduction and by identifying and removing redundant information. The goal with the morphological unit has been to create a data path unit that does not require any intermediate storage of the image. Both decomposition and simple structuring elements have been explored to reach this goal. In the labeling unit, the main issue has been to decrease the amount of data stored on-chip. Here, carefully choosing the algorithm and contemplating system level considerations have resulted in a reduced memory size. Finally, the dependency between image resolution and memory requirements for all parts of the system has been investigated in order to identify the constraints of a future higher resolution system.

Sections 2 to 5 present the individual blocks of the system, as outlined in Figure 1. Each block has been implemented as a stand-alone block, but has been verified using a software simulation model of the complete system. Section 6 discusses how the individual blocks have been integrated on an FPGA board. Hardware utilization, system optimizations and system bottlenecks are also discussed in this section. Finally, our conclusions are drawn in Section 7.

1.1 Systems of Today

Intelligent surveillance is an expansive field as can be seen from the increasing number of products commercially available on the market today. Both surveillance cameras and larger systems with advanced image analysis capabilities are emerging. Three of the largest actors in the market are AXIS Communications, Sony and IBM.

AXIS Communications is one of the global market leaders in network video products who have specialized in professional network video solutions for remote monitoring and security surveillance [5]. Features of AXIS surveillance cameras include built-in motion detectors and Wireless Local Area Network (WLAN) modules. Several subsections of a scene can be specified for motion detection, each with an individual sensitivity level. However, the detection is as for most embedded video motion detection algorithms, very basic and based on frame difference.

One of the most advanced surveillance cameras on the market today is Sony's SNC-CS50 [86]. According to the specification, the camera can perform both advanced motion detection and unattended object detection, but not simultaneously. The unattended object detector reacts if an object is left in one place for more than a specified duration and the motion detection is based on the last fifteen frames in order to reduce noise sensitivity. However, a live demonstration showed that the camera reacts slowly to motion and is sensitive to light changes.

IBM has recently released the Smart Surveillance System (S3) release-1 to end customers on a pilot basis. Compared to the previously mentioned products, S3 is by far the most advanced. However, S3 is not designed to be used in an embedded camera but as a separate software system to which several cameras are connected. According to the website [87], the system is capable of object detection that is insensible to light, weather changes, and camera vibrations. Detected objects can be both tracked and classified. Typical classification labels include, person, group, and vehicle. In addition to real-time tracking and classification, all detected events are stored alongside the original data stream for fast event-based searching in the captured videos. Since no live demonstrator is available and the current release is limited to a small number of test users, it is not possible to evaluate the claimed capability of the system.

From the above overview, it is seen that there is a huge gap between the capabilities of the embedded surveillance cameras produced by AXIS and Sony, and IBM's large scale surveillance system. A similar trend can be seen in academia. Either research about large systems implemented in software or research about isolated algorithms implemented in dedicated hardware is published. For example, W4 [88] is a system that, in addition to motion detection and tracking of multiple people on monocular gray-scale video, tries to detect actions such as persons carrying objects and different body postures. Other surveillance systems that both track and classify objects are found in [89] and [90]. The former focus on classifying events such as people and cars arriving and leaving through a co-occurrence matrix and the latter both describe an attempt to monitor a complex area using a distributed network of cameras. A more recent system that tracks multiple humans in complex situations is [91], were people are tracked in 3D using an ellipsoid shape model. In addition, motion modes, e.g. walking, running, and standing, and body posture are estimated. For a more extensive survey of visual surveillance we refer to [92]. Common to all of these systems is that they are, or need to be, executed on one or more general purpose computers in order to reach real-time performance with an image resolution of 320×240 or greater. Most published hardware implementations deal with smaller parts of a surveillance system, e.g. implementation of motion segmentation, image filtering, or video codec. Some examples are [93] and [94] which describe the implementation of a motion segmentation algorithm and a high speed median filter, respectively. FPGA implementations of video codecs for MPEG-4 and H.264 are found in [95] and [96].

The proposed system, tries to bridge this gap by taking some of the functionality from the software system and putting it in the camera. To have the functionality inside the camera instead of running it on a separate computer has some obvious benefits. Most importantly, the amount of data that has to be transmitted over the network can be reduced, especially important if a wireless scenario is considered. For larger installations this could be critical, e.g. at airports where hundreds of cameras are installed and the aggregated bandwidth becomes substantial. The output from each of these cameras has to be routed to a security central. A reduced bandwidth could then be the difference between using the existing network or installing a completely new one. To move all functionality of a stand-alone software system into the camera will probably never be feasible. However, if some of the functionality is moved, the software system could be redesigned to use the output from the smart cameras instead of the raw image stream that is used today. In larger security systems all cameras would then be connected through a system backbone to a central unit with a coordinating functionality, whereas in smaller systems it could be sufficient to install only smart cameras. Recently, another embedded image system has been presented by Philips Research labs. The system is based on two processors, one for low level image operations and one for higher level applications, connected through a dual-port memory [97] [98] [99]. However, surveillance applications have yet to be demonstrated on it and the amount of available memory limits the possibility of processing scolor images.

The proposed system is an early attempt to move a complete hardware accelerated surveillance system onto a stand-alone embedded system, consisting of an image sensor, an FPGA with an embedded processor, and some external memory.

2 Segmentation

Over the years, various video segmentation algorithms have been proposed, e.g. frame difference, median filters [100] and linear predictive filters [101]. However, to achieve robustness in multi-modal background scenarios, an algorithm based on the GMM proposed in [102] [103] is chosen. A GMM is required for modeling repetitive background object motion, e.g. swaying trees, reflections on a lake surface or a flickering monitor. A pixel located in the region where repetitive motion occurs will generally consist of two or more background colors, i.e. the RGB value of a specific pixel toggles over time. This would result in false foreground object detection with most other adaptive background estimation approaches.

The advantage of the GMM is achieved by using several Gaussian distributions for each pixel. The drawback is the imposed computational complexity and high memory bandwidth that prohibit real-time performance using a general purpose computer. In our simulations, a frame rate of only 4-6 fps is achieved for video sequences with a 320×240 resolution, on an AMD 4400+ dual core processor. For a real-time video surveillance system with higher resolution, hardware acceleration is required. The rest of this section will present how the GMM can be improved and efficiently implemented [104].

2.1 Algorithm Formulation

The algorithm is briefly formulated as follows: In a sequence of consecutive video frames, the values of any pixel can be regarded as a Gaussian distribution. Characterized by a mean and a variance value, the distribution represents a location centered at its mean values in the RGB color space. A pixel containing several background object colors, e.g. a swaying leaf on a tree in front of a road, can be modeled with a mixture of Gaussian distributions with different weights. The weight of each distribution indicates the probability of matching a new incoming pixel. A match is defined as the incoming pixel within a certain deviation from the center. In this work, J times the standard deviation of the distribution is used as the threshold [102]. The higher the weight, the more likely the distribution belongs to the background. Mathematically, the portion of the Gaussian distributions belonging to the background is determined by

$$B = argmin_b \left(\sum_{k=1}^{b} \omega_k > H \right), \tag{1}$$

where b is the number of Gaussian distributions per pixel, H is a predefined parameter and ω is the weight. The mean, variance and weight factors are updated frame by frame. If a match is found, the parameters of the matched distribution are updated according to:

$$\omega_{k,t} = (1-\alpha)\omega_{k,t-1} + \alpha, \qquad \mu_t = (1-\rho)\mu_{t-1} + \rho X_t \tag{2}$$

$$\sigma_t^2 = (1-\rho)\sigma_{t-1}^2 + \rho(X_t - \mu_t)^T(X_t - \mu_t), \tag{3}$$

where μ and σ^2 are the mean and variance, α and ρ are learning factors, and X_t is the pixel value. For those unmatched, the weight is updated according to

$$\omega_{k,t} = (1-\alpha)\omega_{k,t-1}, \tag{4}$$

while the mean and the variance remain the same. If none of the distributions match, the one with the lowest weight is replaced by a distribution with the incoming pixel value as its mean, a low weight and a large variance.

2.2 **Color Space Transformation**

In theory, multi-modal situations only occur when repetitive background objects are present in the scene. However, this is not always true in practice. Consider an indoor environment where the illumination comes from a fluorescence lamp. A video sequence of such an environment was taken from our lab from which 5 pixels were measured over time. Their RGB value distributions are drawn in Figure 2(a) and it can be seen that instead of 5 sphere-like pixel distributions, the shapes of the pixel clusters are rather cylindrical. Pixel values tend to jump around more in one direction than another in the presence of illumination variations caused by the fluorescence lamp and camera jitter. This should be distinguished from the situation where one sphere distribution is moving slowly towards one direction due to slight daylight changes. Such a case is handled by updating the corresponding mean values in the original background model. Without an upper bound for the variance, the sphere describing the distribution will grow until it covers nearly every pixel in the most distributed direction, thus taking up a large space such that most of it does not belong to the distribution (A in Figure 2(b)). A simple solution to work around this problem is to set an upper limit for the variance, e.g. the maximum value of the variance in the least distributed direction. The result is multi-modal distributions represented as a series of smaller spheres (B-E also in Figure 2(b)). Although a background pixel distribution is modeled more precisely by such a method, several Gaussian distributions are inferred, which are costly in terms of hardware because of the extra parameter updating and storage.

To be able to model background pixels using a single distribution without much hardware overhead, color space transformation is employed. Both HSV and $\mathrm{YC}_b\mathrm{C}_r$ space have been investigated and their corresponding distributions are shown in Figure 2(c)-(d). Transforming RGB into $\mathrm{YC}_b\mathrm{C}_r$ space results

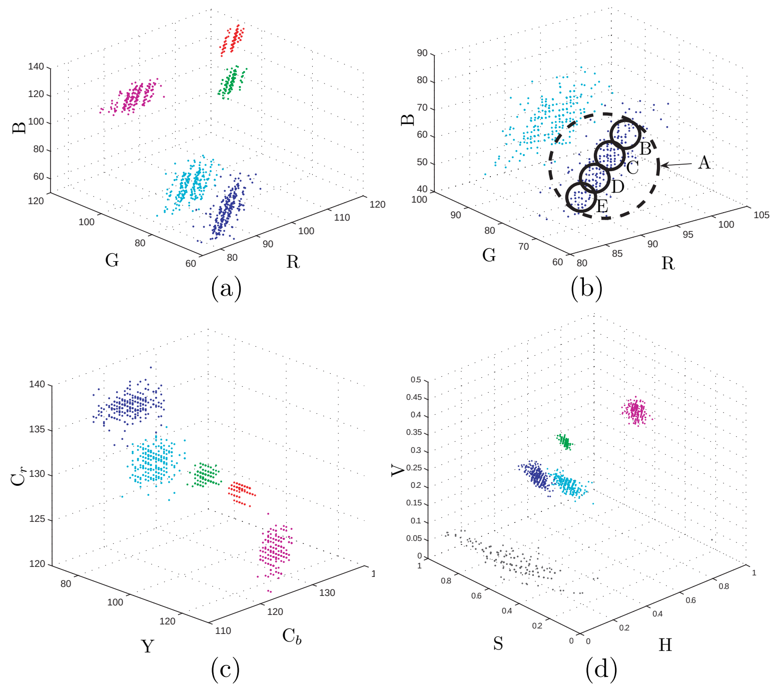


Figure 2: (a) 5 distributions in the RGB color space. (b) A closer look at the 2 Gaussian distributions on the bottom in (a). (c) Sphere distributions in the YC_bC_r space. (d) Unpredictable distributions in the HSV space.

in nearly independent color components. Accordingly, in a varying illumination environment, only the Y component (intensity) varies, leaving C_b and C_r components (chromaticity) more or less independent. In [105], this feature is utilized for shadow reduction. Consequently, values of the three independent components in the YC_bC_r color space tend to spread equally and as shown in Figure 2(c), most pixel distributions are transformed from cylinders back to spheres, capable of being modeled with a single distribution. The transformation from RGB to YC_bC_r is linear, and can be calculated with a low increase in computational complexity (see Section 6). On the other hand, HSV color space is no better than RGB color space, if not worse. Unpredictable pixel clusters appeared occasionally, which is hard to model using Gaussian distributions, as shown in Figure 2(d).

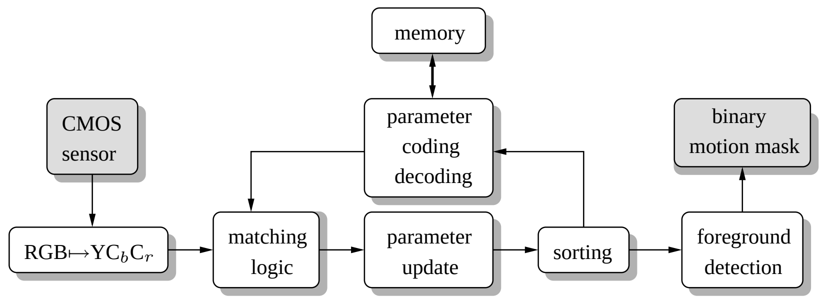


Figure 3: The architecture of the segmentation unit.

2.3 Segmentation Architecture

Maintaining a mixture of Gaussian distributions for each pixel is costly in terms of both calculation capacity and memory storage, especially at high resolution. To manage the RGB data from a video camera in real time, a dedicated hardware architecture is developed with a streaming data flow. The hardware architecture, as shown in Figure 3, is presented in [104] and briefly explained as follows: a pixel value is read into the matching logic block from the sensor together with all the parameters for the mixture of Gaussian distribution from an off-chip memory. A match is then calculated. In case an incoming pixel matches several Gaussian distributions, only the one with highest weight is selected as the matching one.

After the updated Gaussian parameters have been sorted, foreground detection is achieved by simply summing up the weights of all the Gaussian distributions that have a higher likelihood than the updated one. By comparing the sum with a predefined parameter H, a sequence of binary data indicating background and foreground is streamed out to the morphology block. The main bottleneck of the architecture is the high bandwidth to the off-chip memory, which will be addressed in the following secation.

2.4 Wordlength Reduction

Slow background updating requires a large dynamic range for each parameter in the distributions, since parameter values are changed slightly between frames but could accumulate over time. According to (2) and (3), the mean and variance of a Gaussian distribution is updated using a learning factor ρ. The difference of mean and variance between current and previous frame is derived from the equation as

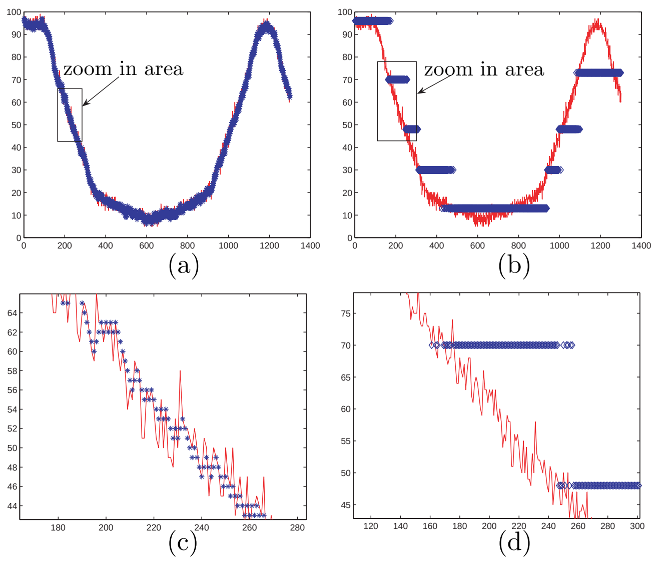


Figure 4: A comparison of parameter updating schemes in a fast light changing environment. One color value (solid line) of an RGB pixel is drawn over the frames together with the updated Gaussian RGB mean value (blue diamond line). The zoomed in area of (a) and (b) is shown in (c) and (d), respectively.

$$\begin{aligned} \Delta_\mu &= \mu_t - \mu_{t-1} = \rho(X_t - \mu_{t-1}) \quad \text{and} \\ \Delta_{\sigma^2} &= \sigma_t^2 - \sigma_{t-1}^2 = \rho((X_t - \mu_t)^T(X_t - \mu_t) - \sigma_{t-1}^2). \end{aligned} \tag{5}$$

Given a small value of ρ, e.g. 0.0001, a unit difference between the incoming pixel and the current mean value results in a value of 0.0001 for Δ_μ. To be able to record this slight change, 22 bits have to be used for the mean value, where 14 bits account for the fractional part. Empirical results have shown that the Gaussian distributions usually are spheres with a diameter less than 10 and in this study, as well as in [102], $J = 2.5$. Therefore, an upper bound for the variance is set to 16 and a maximum value of Δ_μ becomes $\rho \times J \times \sigma = 0.0001 \times 2.5 \times \sqrt{16} = 0.001$, which can be represented by 10 bits. If a wordlength lower than that was to be used, no changes would ever be recorded. In practice, the bits for the fractional parts should be somewhere between 10-14 bits and 7-14 for the mean and variance, respectively. Together with 16 bits weight and integer parts of the mean and the variance, 81-100 bits are needed for a single Gaussian distribution. To reduce this number, a wordlength reduction scheme was proposed in [104]. From (5), a small positive or negative number is derived depending on whether the incoming pixel is above or below the current mean. Instead of adding a small positive or negative fractional number to the current mean, a value of 1 or -1 is added. The overshooting caused by such coarse adjustment could be compensated for by the update in the next frame. The result is that without illumination variation, the mean value will fluctuate by a magnitude of ONE, which is negligible since the diameter of the Gaussian distribution is usually more than 10.

In a relatively fast illumination varying environment, fast adaptation to new lighting conditions is also enabled by adding or subtracting ONES in consecutive frames. Figure 4(a) shows the experimental results of the coarse updating in a room with varying lighting conditions. The parameter updating scheme specified in the original algorithm is also drawn in Figure 4(b) for comparison. A closer look at the two schemes is shown in Figures 4(c) and (d). From Figures 4(b) and (d), it is seen that parameter updating (diamond line in the figure) of the original algorithm does not work well in the presence of fast light changes. The Gaussian distribution will not keep track of the pixel value changes and Gaussian distribution replacement takes place instead of parameter updating. On the other hand, the coarse updating scheme handles such situations with only parameter updating.

With coarse updating, only integers are needed for mean specification, which effectively reduce the wordlength from 18-22 down to 8 bits. A similar approach can be applied to the variance, resulting in a wordlength of 6 bits, with 2 fractional ONES. Together with the weight, the wordlength of a single Gaussian distribution can be reduced from 81-100 to only 44 bits, resulting in a reduction greater than 45%. In addition, less hardware complexity is required since multiplication with the learning factor of ρ is no longer needed.

2.5 **Pixel Locality**

In addition to wordlength reduction, a data compression scheme for further bandwidth reduction is proposed by utilizing pixel locality for Gaussian distributions in adjacent areas. Consecutive pixels often have similar colors and hence have similar distribution. We classify “similar” Gaussian distributions in the following way: using the definition of a matching process, each Gaussian distribution can be simplified as a cube, where the center is the YC_bC_r mean value and the border of the center is specified as J times the variance. One way to measure the similarity between two distributions is to check the overlap of the two cubes. If the overlap takes up a certain percentage of both Gaussian cubes, they are regarded as “similar”. The overlap is a threshold parameter

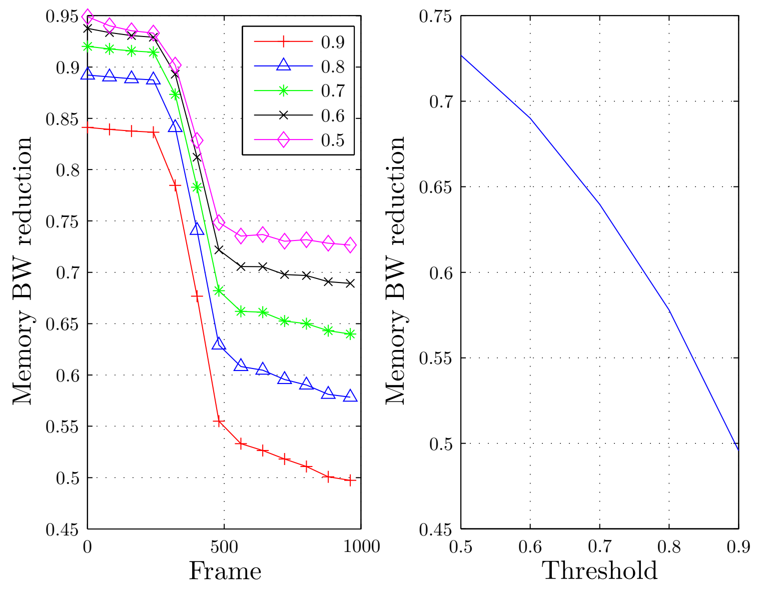


Figure 5: Memory Bandwidth (BW) reduction over frames for different thresholds is shown to the left and memory bandwidth reduction versus threshold is shown to the right.

that can be set to different values for different scenarios.

In the architecture, two similar distributions are treated as equivalent and by only saving non overlapping distributions together with the number of equivalent succeeding distributions, memory bandwidth is reduced. Various threshold values are selected to evaluate the efficiency for memory bandwidth reduction. With a low threshold value, greater savings could be achieved, but at the same time more noise is generated due to increasing mismatches. Fortunately, such noise is found to be non-accumulating and can therefore be reduced by morphological filtering, as presented in Section 3. Figure 5 shows the memory bandwidth savings over frames with various threshold values. It can be seen that memory bandwidth savings tends to stabilize (around 50% - 75% depending on threshold value) after initialization. The quality of segmentation results before and after morphology is shown in Figure 6, where it is clear that memory reduction comes at the cost of segmentation quality. Too low a threshold value results in clustered noise that would not be filtered out by morphological filtering, as shown in Figure 6(c).

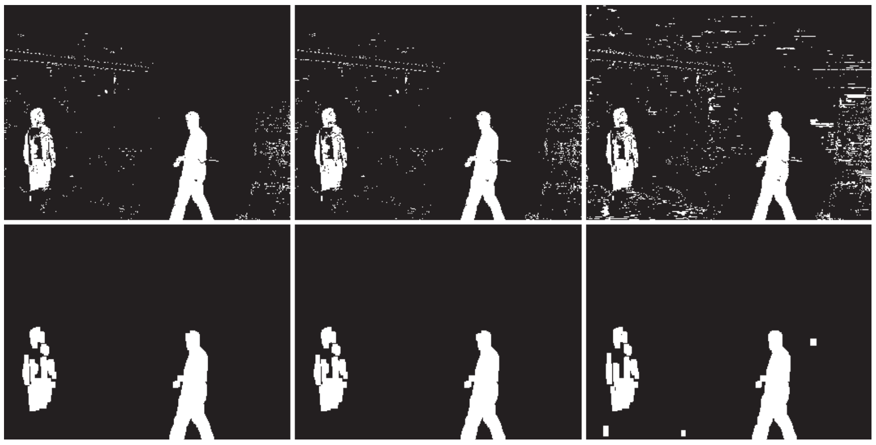

(a) (b) (c)

Figure 6: The result before and after morphological filtering for different thresholds, (a) original result, (b) with 0.8, and (c) with 0.4 threshold.

3 Morphology

As seen in the previous section, the generated binary mask needs to be filtered to reduce noise and reconnect split objects. This is accomplished by applying mathematical morphology. Erosion (ε) and dilation (δ) are the two foundations in mathematical morphology, from which many other extended operations are derived [106], e.g. opening, closing, and gradient. Mathematical morphology applies to many image representations [68], but only binary ε and δ are required in our system.

In an effort to make the binary morphological processing effective, a low complexity and low memory requirement architecture was proposed in [107]. This architecture has several properties and benefits which are of special interest for our application in order to easily incorporate the unit into the system. First, pixels are processed sequentially from first to last pixel. Since each operation is completed in a single image scan, a short execution time is ensured and no extra memory handling is invoked. This allows for several ε and δ units to be placed in series or parallel with only a small FIFO in between the blocks, to account for stall-cycles due to inserted boundary pixels (padding). Another property of the architecture is that the size of the Structuring Element (SE) can be changed for each frame during run time. With a flexible SE size comes the ability to compensate for different types of noise and to sort out certain

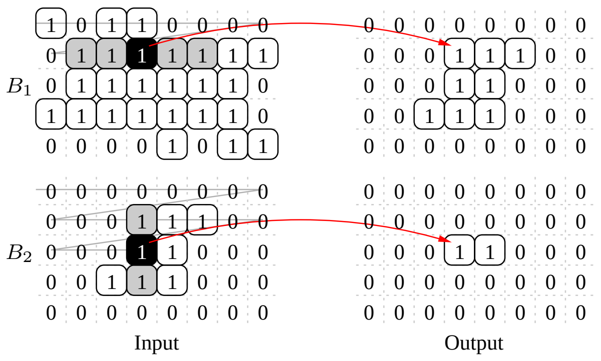


Figure 7: Input and output to an erosion were a SE of size 3×5 is decomposed into $B_1 = 1 \times 5$ and $B_2 = 3 \times 1$.

types of clusters, e.g. high and thin objects (standing humans) or wide and low objects (side view of cars).

Let I represent the binary input image and B the structuring element. If B is both reflection invariant and decomposable, i.e. $B = \widehat{B}$ and $B = B_1 \oplus B_2$, the following two equations for ε and δ can be derived

$$\varepsilon(I, B) = I \ominus (B_1 \oplus B_2) = (I \ominus B_1) \ominus B_2, \tag{6}$$

$$\delta(I, B) = (I \oplus B_1) \oplus B_2 = ((I' \ominus B_1) \ominus B_2)', \tag{7}$$

where $'$ is bit inversion. Comparing (6) and (7), it can be seen that both ε and δ can be expressed as an erosion (or as a dilation). This property is known as the duality principle. With a decomposed SE, the number of comparisons per output is decreased from the number of ones in the B to the number of ones in B_1 plus B_2. However, finding decompositions of an arbitrary SE is difficult and not always possible [45] [46]. In addition, for a SE to be reflection invariant it has to be symmetric in respect to both i and j axes, e.g. an ellipse. However, a common class of SEs that is both decomposable and reflection invariant are rectangles of ones. This type of SE is well suited for operations such as opening and closing, which are needed in this system. An example of ε with a decomposed SE is shown in Figure 7, were the SE is decomposed into B_1 and B_2, see (6). The input is first eroded by B_1 and then by B_2 and the number of comparisons per output is reduced from 15 to 8.

3.1 Morphology Architecture

By using a rectangular SE containing only ones, ε can be performed as a summation followed by a comparison. The ε is performed by keeping track of the bits in I that are currently covered by the B and are compared to its size in both the i and j direction. By decomposing the SE, the summation can be broken into two stages. The first stage compares the number of consecutive ones in I to the width B_1 and outputs a one if this condition is fulfilled. The second stage sums the result from the first stage for each column and compares it to the height of B_2. If both these conditions are fulfilled, the output at the coordinate of the SE origin is set to one, else zero.

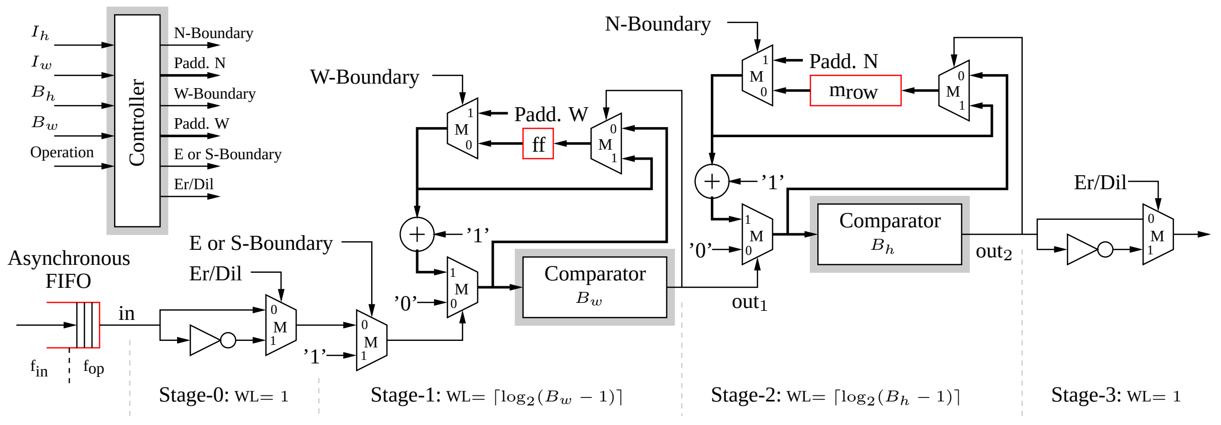


Figure 8: Architecture of the datapath in the erosion and dilation unit with corresponding wordlength in each stage.

The proposed architecture is based on the observations above and is shown in Figure 8 with corresponding wordlength in each stage. Taking advantage of the duality property, the same inner kernel is used for both δ and ε; to perform δ on a ε unit simply invert the input I and the result, performed in Stage-0 and 3. Each pixel in I is used once to update the sum. The sum is stored in the flip-flop in stage-1, which records the number of consecutive ones to the left of the currently processed pixel. When the input is one, the sum is increased, else reset to zero. Each time the sum plus the input equals the width of B_1, stage-1 outputs a one to stage-2 and the previous sum is kept. The same principle is used in stage-2 but instead of a flip-flop, a row memory is used to store the number of ones from stage-1 in the vertical direction for each column in I. In addition, omitted from the figure, a controller is required to handle padding and to determine the operation to be performed, i.e. ε or δ. How, and why, padding is inserted around the boundary of an image is discussed in [107].

The wordlength in Stage-0 and 3 is a single bit whereas the wordlengths in stage-1 and 2 are proportional to the maximum supported size of the B, i.e. $\lceil \log_2(B_w) \rceil$ and $\lceil \log_2(B_h) \rceil$, respectively. Thus, the total amount of required memory to perform ε or δ is

$$\text{mem}_{tot} = \lceil \log_2(B_w) \rceil + \lceil \log_2(B_h) \rceil I_w \text{ bits},$$

where the first part is the flip-flop in stage-1 and the second part is the row memory in stage-2. As an example, with a resolution of 320×240 and a B size of 15×15, the required amount of memory is $\lceil \log_2(15) \rceil + \lceil \log_2(15) \rceil \cdot 320 = 1.28$ kbits. The delay line implementations in [37] and [58], with the same resolution and SE size, would require $B_w + (B_h - 1)I_w = 4.50$ kb of memory, which is $\approx$ 3.5 times more.

The morphological operations used in this system are an ε followed by a δ. Due to the pipelined nature of the architecture, the two operations can be performed directly on the output stream from the segmentation by placing two units in series. This will not increase the execution time but only add a latency of a few clock cycles. Examples of filtered segmentation results are shown in Figure 6. The image is first eroded with an SE size of 5x3 and then dilated with a 7x5 SE.

4 Labeling

After the morphological operation, the binary frame contains connected clusters of pixels that represent different objects of interest which should be tracked and classified. However, the system needs to be able to separate and distinguish between these clusters. Labeling has the goal of assigning a unique label to each cluster, transforming the frame into a symbolic object mask with the possibility of tying features to each cluster. Thus, labeling can be seen as the link between the clusters and their corresponding features. Labeling algorithms dates back to the early days of image processing [48] and applies to many image representations [49]. Various algorithms have been proposed over the years and a survey can be found in [50]. The algorithms can be placed into two major categories, namely

- Sequential Local Operations (SLO), and
- Contour Tracing (CT).

The remainder of this section describes a comparison between these two types of algorithms in terms of memory requirements and which features they can extract.

In SLO-based algorithms [108], a label is assigned based upon the pixels above and to the left of the current pixel which comes naturally when working on streaming data. However, this type of algorithm have to solve possible label collisions. A typical label collision occurs if a u-shaped object is encountered. Scanning the image, the pillars will be assigned different labels since there is no momentary information advising they are part of the same cluster. Reaching the lower middle part of the u, an ambiguity about which label to assign will occur, referred to as a label collision. A common way to solve this is to write the label collisions into an equivalence table during an initial scan and resolve them during a second. The number of label collisions per frame depends on the complexity of the cluster contours.

CT-based algorithms trace and label the contour of each cluster [55]. Labeling the contour will avoid label collisions since, if a previously labeled cluster (contour) is encountered, the scan proceeds without modification. The algorithm requires a global memory scan together with additional random accesses for the CT procedure in order to label all clusters in a frame. In order to avoid pitfalls such as tracing contours of possible holes inside clusters, a reserved label is written on each side of the cluster. Based on this reserved label, the algorithm keeps track of whether it is currently inside a cluster or not. In the same manner, when reading the labeled result, pixels between two reserved labels can be considered part of the same cluster regardless of the pixel value. Thus, holes inside clusters can be filled, which is beneficial in our application.

Both types of algorithms need a memory to store the labeled image result, mem_{label}. Due to the physical limitations of this memory, an upper bound is placed on the number of clusters that can be labeled in a frame $c_{\max}$. In SLO-based algorithms, each label collision will occupy a temporary label during the initial scan. The memory size is determined by a combination of $c_{\max}$ and the maximum number of label collisions $l_{\text{c,max}}$. Thus, a memory overhead is

Table 1: Three simulations on independent sequences showing the number of clusters and corresponding label collisions.

Sequence	**Seq. 1**	**Seq. 2**	**Seq. 3**
Mean clusters per frame	4.20	6.98	6.56
Mean label$_{col}$ **per frame**	13.1	6.72	14.3
Max clusters in a frame ($c_{\max}$)	14	20	20
Max label$_{col}$ **in a frame**($l_{c,\max}$)	27	21	50
Nbr. of frames in the seq.	700	900	2500

introduced. In CT-based algorithms, the memory size is directly proportional to the image resolution and $c_{\max}$. The memory requirement for the SLO and CT-based algorithms can be written as

$$\text{mem}_{SLO} = \lceil \log 2(c_{\max} + l_{c,\max} + 1) \rceil \cdot N^2 \text{ bits}, \tag{8}$$
$$\text{mem}_{CT} = \lceil \log 2(c_{\max} + 3) \rceil \cdot N^2 \text{ bits}, \tag{9}$$

where N^2 is the number of pixels in an image and $+1$ and $+3$ comes from the number of preoccupied labels.

Table 1 compiles three simulations that show the number of clusters with corresponding label collisions per frame. Sequence 1 is captured in our research laboratory, Sequence 2 is captured outdoors covering a traffic crossing, and sequence 3 is taken from the PETS database [83]. Using (8) and (9) and figures from Table 1, SLO-based algorithms would require 6, 6, 7 bits per pixel compared to CT-based algorithms which would require 5, 5, 5 to handle the worst case scenario for sequence 1 to 3, respectively. Hence, CT-based algorithms requires less total memory compared to SLO-based algorithms to be able to label the same number of clusters.

From a system perspective, it is desirable to extract features where they have low requirements in terms of execution time and hardware complexity. Since the clusters are scanned during the labeling process, many binary features are advantageously extracted by this unit, e.g. coordinates which are used to create a bounding box around each cluster. The extraction procedure of many features is the same for both types of algorithms. However, a unique property of CT-based algorithms is that they have possibility of calculating the discrete Green's theorem during the CT phase. Green's theorem gives the relationship between a closed contour (curve) and a double integral over this cluster (plane region), enabling calculation of moments [84]. Moments can in turn be used

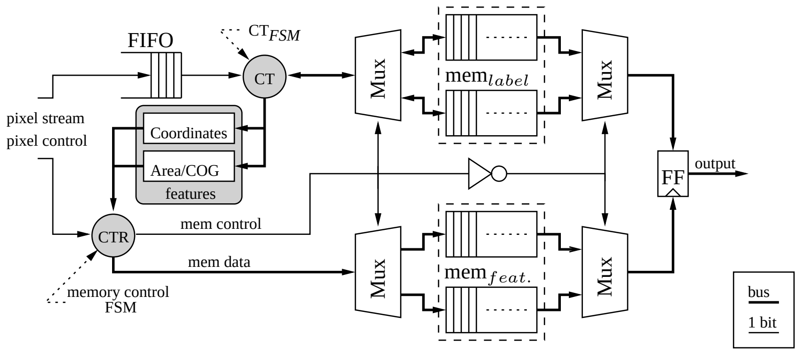


Figure 9: Overview of the implemented CT-based architecture.

to calculate area and CoG which are important properties in this particular application, e.g. used by the tracking unit to handle occlusion.

Summarizing the comparison between the two types of algorithms, a common property is that they impose a high bandwidth together with large memory requirements. Since memory issues are the major concern in our application, arithmetic complexity in the algorithms will be traded for memory resources. Therefore, the CT-based algorithm was found more suitable in our particular application and chosen for implementation, due to the following properties:

- CT-based algorithms requires less memory and can guarantee labeling of a predefined number of clusters.
- Both types of algorithms have the same upper bound on execution time, $t_{exe} \leq 3 \cdot (I_h \times I_w)$ [109].
- CT-based algorithms have the possibility of adding Green's formula and thereby extracting CoG,
- CT-based algorithms have the possibility of filling holes inside a cluster.

4.1 Labeling Architecture

An overview of the CT-based architecture implemented in [109], is illustrated in Figure 9. A FIFO is located at the input in order to stall the data stream as a frame is being labeled. The CT$_{FSM}$ first writes the complete frame into mem$_{label}$. The first and last pixel equal to 1 for this frame is marked as global start and end point respectively. After that, a second memory scan starts from the global start pixel, now also marked as local starting pixel for this

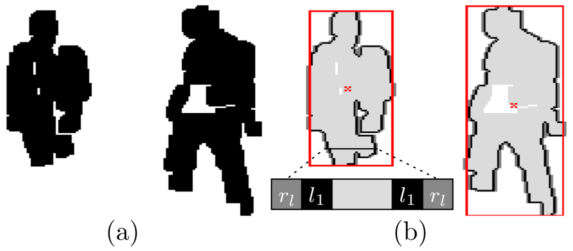


Figure 10: (a) A fragment of a typical binary input frame to the label unit, (b) corresponding labeled output. The result includes bounding boxes around each object with their CoG marked as an x. Notice r_l on each side of the cluster line segment which corresponds to the reserved label.

particular cluster. The CT$_{FSM}$ traces the contour of the cluster, and writes the label into mem$_{label}$. The contour tracing of this cluster is completed when the local starting pixel is reached a second time. The global scan then continue until a new cluster or the global end point is reached.

During the CT phase, the feature extraction blocks calculate i and j coordinates, height, width, size, and CoG, for every cluster and stores the result in the feature memory, i.e. mem$_{feat}$. To maximize the time the embedded SW (tracking algorithm) can access this result, a dual memory structure is used. Hence, as the algorithm is labeling frame f in one memory pair, the tracking algorithm has access to the result of frame $f-1$ in the other memory pair. An example of a binary frame together with corresponding labeled output from the implemented architecture can be seen in Figure 10.

Some applications do not require unique labels and binary features are sufficient. In such applications, the CT-based algorithm allows the label memory to be reduced to 2 bits per pixel, while still maintaining correct binary feature extraction, since each cluster will get a separate entry in the feature memory. This observation is further discussed in Section 6.

5 Tracking

The goal of the surveillance system is to track people while they remain in view of one stationary camera. Each person in view should be given a unique identity that should remain fixed even though people change place and/or disappears briefly from the scene. In the following text, people or things that are tracked, i.e. given a unique identity, are referred to as objects whereas objects detected

Table 2: The feature classes, features part of the class, and when they are calculated.

Class	**Features**	**Calculation**
Cluster	Size, min coordinates, Height, Width, CoG coordinates	For every cluster and frame
Color	Mean, Variance, Histogram	If occlusion is detected
Prediction	d'CoG, d'width, d'height, d'size	For every tracked object and frame

by the motion detector are referred to as clusters.

Tracking of non-rigid objects, e.g. humans, is complicated and becomes even harder when it has to be performed on an embedded system with limited resources. An initial decision must be made regarding hardware/software partitioning. Software has the benefits of flexibility and shorter design time and hardware has the advantage of high throughput. To take advantage of both these properties, the system is partitioned so that tasks that must be executed directly on the image stream are implemented in hardware, while bookkeeping and conditional tasks are performed in software. The result is that tracking is performed in software and all preprocessing and measurements on the image stream are performed in hardware. The interface between hardware and software is the features.

A feature is a property extracted from an object in the image, e.g. size, color, texture, or shape, that can separate different objects or recognize a certain object class. A good feature describes each object with a unique and compact code and does not change if the object is scaled, rotated, or enters an area with different lighting. This is necessary to be able to track an object through different environments, e.g. track a person standing under a lamp close to the camera who moves away towards a darker corner.

In this system there are three feature classes that are acquired from different parts of the system, at different times and during various conditions. First, the cluster features acquired from the binary motion mask in the label unit. These features are calculated for each labeled cluster and for each frame. Secondly, color features are calculated if an occlusion between two objects is detected. The third feature class is prediction features that are used to make an initial

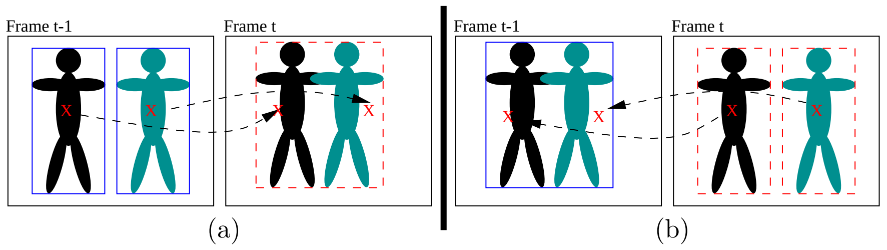


Figure 11: BBs around tracked objects are shown with solid lines and around new clusters with dashed, x marks the CoG. (a) Shows a merge event and (b) a split event.

guess about which objects from previous frames correspond to which objects in the current frame. Table 2 summarize the different feature classes.

Cluster features include minimum i and j coordinates, height and width of the cluster, the number of pixels in a cluster (size), and CoG coordinates. These features are used as initial data in the tracking algorithm, which starts with a reconstruction phase. In this phase, objects from previous frames are reconstructed from the detected clusters. This is necessary since objects can consist of more than one cluster due to imperfect segmentation and occlusions. The reconstruction is based on the predicted position of an object's CoG and size. When two or more clusters are used to reconstruct an object, new cluster features are calculated as the weighted mean of the used clusters. Cluster features are often sufficient to track non-occluded objects in the video stream.

During the reconstruction phase, merges and splits are also detected. A merge occurs when two objects touch each other and become one object, i.e. an object-object occlusion, and a split is when one object becomes two objects. Both events are detected in a similar way, based on CoG coordinates and bounding boxes (BB). The BB is defined as the minimum rectangle that completely surrounds an object or cluster and it is created with the cluster features: width, height, and minimum coordinates. A merge is detected if the CoG of two tracked objects are found inside the BB of one new cluster, and a split is detected if two cluster CoGs are found inside the BB of one object. An example is shown in Figure 11.

Color features include the mean, variance, and histogram of an object. These features have been chosen since they may be calculated from streaming data without any reordering of the pixels and they produce a minimal amount of data, i.e. minimum processing time and memory requirements. In addition, color features are size invariant and, with the right color space, also lighting

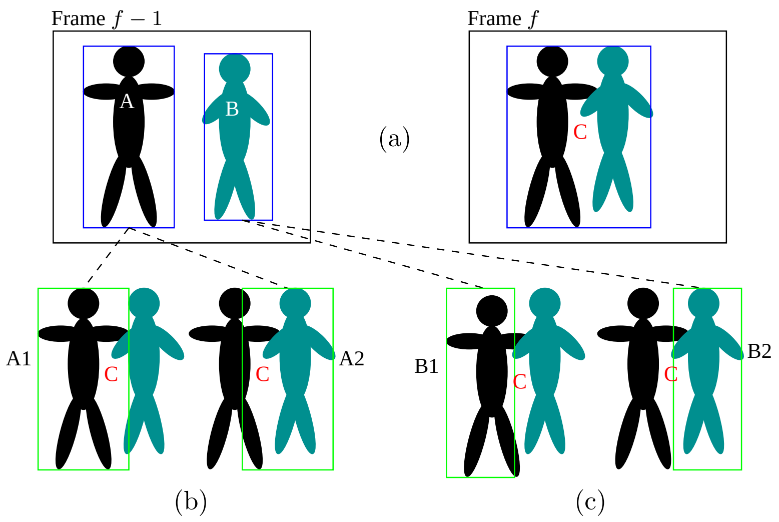


Figure 12: (a) Object A and B in frame $f-1$ and f. Object C is object A and B merged. (b) The two feature sets of A extracted from C. (c) The two feature sets of B extracted from C.

invariant [105].

If the predicted position of an object's BB in the next frame is overlapping the predicted position of another object, i.e. an occlusion is imminent, color features are extracted and stored as a reference. During the rest of the occlusion, two sets of features are extracted for each participating object. One set assumes that the object is to the right of the other object and the other set assumes that it is to the left. For example, if object A and B merge and form object C, Figure 12 shows which parts of C are used to calculate the feature sets for both object A and B. The four feature sets are then matched against the two stored reference sets and a Left-Right (LR) score is stored for each object. If an object is best matched with the right, left or no feature set, the LR score is adjusted according to

$$LR(f) = \begin{cases} LR(f-1)\alpha + K & \text{if a right match,} \\ LR(f-1)\alpha & \text{if a no match,} \\ LR(f-1)\alpha - K & \text{if a left match,} \end{cases}$$

where $\alpha < 1$ and K are constants, and f is the frame number. The larger the $|LR|$, the stronger the evidence that the object is to either the right or left side. The final decision on which object is which is not taken until a split event is

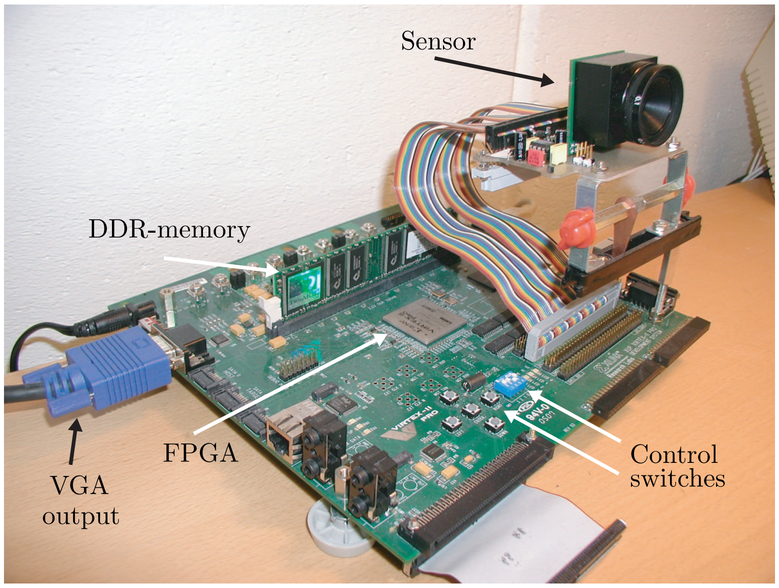


Figure 13: Xilinx XUP Virtex-II Pro FPGA development board with attached sensor. Some of the main blocks are indicated that are relevant to this application.

detected. The main advantages of this method are that no motion prediction is used to estimate the outcome and that it easily scales to more than two objects. Since no motion estimation is used, the system will not be confused if a person moves behind another person, stops, turns around and moves back the same way.

6 System Implementation and Performance

A prototype of the system is implemented on a Xilinx Virtex II pro vp30 FPGA development board, with two FPGA embedded Power PCs and a 256 MB off-chip Double Data Rate (DDR) SDRAM. A KODAK KAC-9648 CMOS sensor is attached directly onto the board and is used to capture color images at 25 fps with a resolution of 320×240. The development board is shown in Figure 13.

The architecture of the prototype is shown in Figure 14, where black indicates custom made logic, light blue is memories and red is off-the-shelf components. The architecture is modular in the sense that each block can be replaced with other algorithms without changing the overall architecture. Modularity is achieved with independent clock domains and asynchronous FIFOs in between

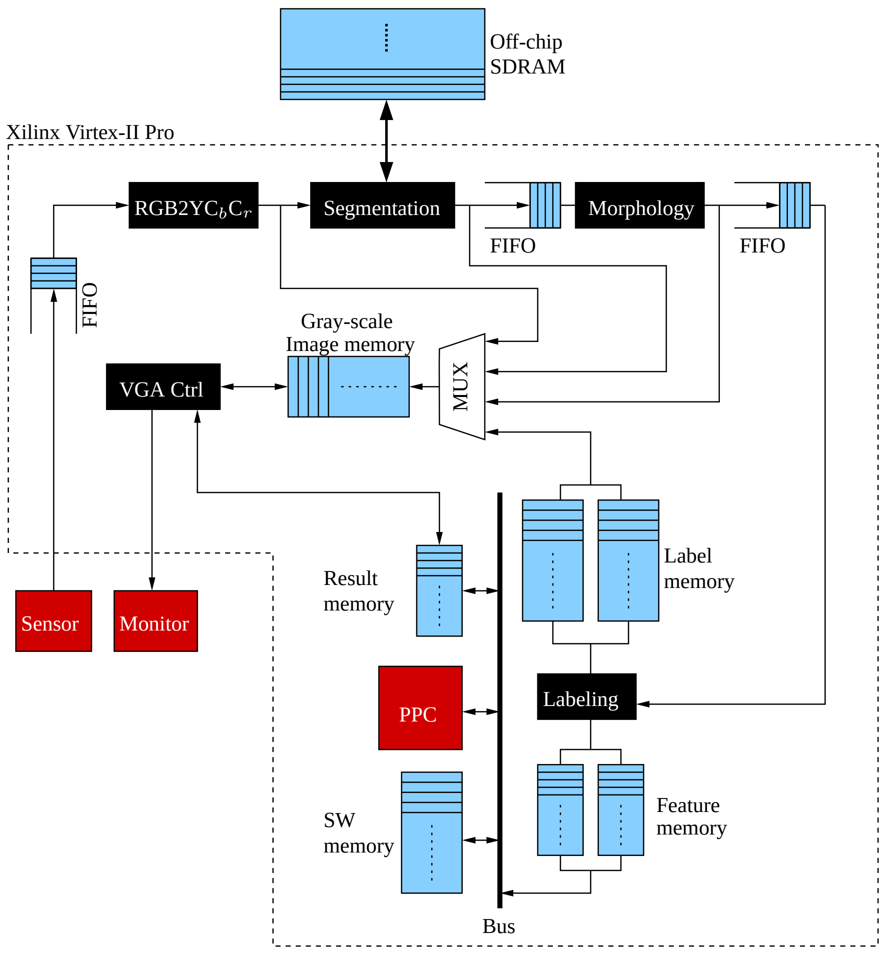


Figure 14: System architecture of the prototype, where black indicates custom made logic, light blue (light gray) is memories and red (dark gray) is off-the-shelf components.

all major blocks. Communication between feature memories and the PPC is performed with software addressable registers and is initialized with an interrupt signal from the label unit. A custom made Video Graphics Array (VGA) controller makes it possible to superimpose bounding boxes around the detected clusters on the output from any block. The output image can also be frozen in order to observe details. Typical outputs from the prototype are shown in Figure 15 and example videos can be found on the project website [110].

No color features are extracted in the current version of the prototype, since the memory is not big enough to store a color image and the software memory is not sufficient for the complete tracking code. Current tracking software reads in all cluster features of all labeled clusters in order to draw the corresponding BB. To free on-chip memory and to be able to include color feature extraction, two additional external memories will be added to the board. One memory will contain the software and the other a complete color image.

The prototype delivers 25 fps with an image resolution of 320×240 pixels. Three Gaussian distributions per pixel, stored in an off-chip SDRAM, are used to perform color image segmentation. The morphology unit performs an erosion followed by a dilation with a flexible SE that can be of any size up to 15×15. As default, the SEs are set to 3×5 and 5×7 in the erosion and dilation blocks, respectively. The labeling unit extracts cluster features from up to 61 clusters per frame. The most important parameters of the different blocks are controlled with dip-switches on the board.

The chosen maximum number of labeled clusters per frame, 61, is based on SW simulations. This number together with (9) and the system environment, would with unique labels, result in a total memory requirement of $\text{mem}_{tot} = FIFO + 2 \cdot (\text{mem}_{CT} + \text{mem}_{feat.}) \approx 1.01$ Mbit, where the factor 2 is due to the dual memory structure [109]. However, in our application a single label can be used without system performance degradation. Using one unique label will neither affect cluster nor color feature extraction. Cluster features are extracted during contour tracing and color features are extracted on single objects even before the BB of another object overlap, i.e. in either case label ambiguity is avoided. The result is that the label memory only uses two bits per pixel and the complete label unit requires 446 kbit instead of 1.01 Mbit.

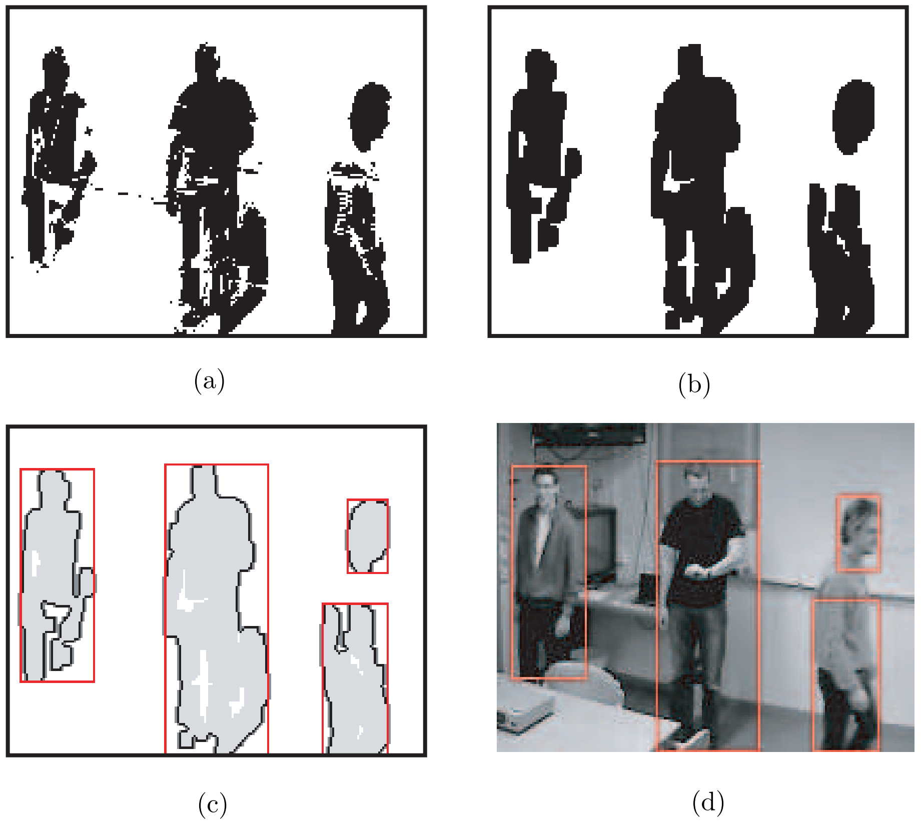


Figure 15: Typical results from the different units, (a) segmentation result, (b) the output from the morphological unit after the erosion and dilation have been performed, (c) labeled output, (d) and original video. The bounding boxes shown in (c) and (d) are generated from the PPC and can be applied to any of the outputs.

Table 3: Hardware resources and utilization of the different parts of the prototype. Figures for the segmentation block includes sensor control logic, and the VGA controller includes both result and image memory. Color feature extraction is currently not part of the prototype, but is included for comparison.

System part	**RGB to YC$_b$C$_r$**	**Segmentation**	**Morphology**	**Label**	Color feature	**Track**	**VGA Ctrl**	**Total use**	**FPGA board**
LUTs	0.8%	**12.4%**	0.6%	8.1%	14.8%	0%	1.2%	39%	27392
18x18 mult	2.2%	**5.2%**	0%	1.5%	6.6%	0%	0%	24%	136
Mem$_{int}$ [kbit]	0%	9.6%	1.5%	20%	0%	11.8%	**32%**	**74.9%**	2448
Mem$_{ext}$ [MB]	0%	**0.5%**	0%	0%	0%	0%	0%	0.5%	256
PowerPC	0%	0%	0%	0%	0%	**50%**	0%	50%	2
f$_{op}$ [MHz]	8	8	9	67	N.A.	**100**	25	—	—
f$_{max}$ [MHz]	N.A.	83	146	70	100	**300**	N.A.	—	—

There are two main purposes of the prototype apart from verifying functionality. One is to perform high speed testing of different configurations, settings and long-term effects of the individual blocks. Software simulation of long-term effects can be extremely time consuming, whereas "simulations" with the prototype are performed in real-time. To facilitate repeatability, the sensor is disconnected and input is read from a file instead. The second purpose of the prototype is to find system bottlenecks. The required hardware resources are shown in Table 3 together with the speed of all blocks. LUTs show the amount of required logic, and the 18×18 multipliers are hard macro multipliers in the FPGA. Internal and external memory refers to the on-chip block memories and the off-chip SDRAM, respectively. The speed required to reach 25 fps is shown as operating frequency and the standalone speed of a block is shown as maximum frequency. It is seen that the morphology block is very small compared to the other parts of the system and that RGB to $\mathrm{YC}_b\mathrm{C}_r$ conversion does not add a significant amount of resources to the segmentation unit. The performance of the system is to a large extent dependent on the segmentation quality, hence the great attention paid to segmentation improvements such as the right color space and reduced wordlengths and memory bandwidths. Despite the improvements, measured in LUTs, multipliers, and external memory, the segmentation unit still requires most hardware. However, none of these resources are critical on a system level and will not be critical even when the color feature block is added. On a system level, internal memory is critical. Almost 75% is used and most of it is due to the grey-scale image that is stored in the VGA-controller. To extend the system to store and display color images, off-chip memory is required.

6.1 Bottlenecks

The presented system uses a resolution of 320×240, which is rather low compared to modern digital video cameras. This resolution is used due to the limited amount of resources, especially memory, on the FPGA. However, future surveillance systems will most likely require a higher resolution. Therefore, it is of interest to study system bottlenecks and how they react to an increased resolution while maintaining real-time performance. For example, if the resolution increases to 640x480, i.e. four times as many pixels per image, and the frame rate remains at 25 fps, how will this affect the different parts of the system and what can be done to decrease the impact of an increased resolution?

The segmentation algorithm scales linearly, i.e. the critical memory bandwidth increases to 4.3 Gbit/s with the straightforward implementation and to 0.82 Gbit/s with the presented memory reduction scheme. To reduce the bandwidth further the approach presented in [111] could be used, where the distributions are not updated for every frame. The morphology unit is much less affected by a resolution increase, since the memory is only dependent on the width of the image. If the SE is increased to match the higher resolution, i.e. to 31x31 pixels, only 2.5 times more memory is required in the data path and the intermediate FIFOs are unaffected. In the label unit, both label memories increase by a factor of 4. One way to reduce this could be to keep only one label memory used by the contour tracing algorithm as it is, and compress the resulting labeled image into a smaller memory using a compression scheme, e.g. run length encoding or JBIG [112]. In terms of memory, feature extraction is unaffected by the resolution increase, since it only works on streaming data and only stores the result. However, it will require 4 times as many clock cycles to execute; this is true for all previous blocks as well. The only part totally unaffected by the resolution increase is the tracking part. Neither the number of objects nor the number of features per object is affected by a resolution increase.

7 Conclusions

An embedded automated digital surveillance system with real-time performance is presented. The system has been developed in order to identify and propose solutions to computational and memory bottlenecks. Due to the real-time processing, it also substantially reduces analysis of long-term effects due to changes in the algorithms and to parametric changes.

The main bottleneck of image processing algorithms is high memory requirements. Therefore, a new memory scheme in video segmentation using wordlength reduction and pixel locality is proposed, resulting in a memory bandwidth reduction greater than 70%. A morphological datapath unit with a memory requirement proportional to image width is presented. It is also shown that in our application, the labeling memory can be reduced by more than 50% if a contour tracing based algorithm is used. On a system level, it is shown that on-chip memory is the main bottleneck. A system prototype has been implemented and is running in 25 fps on an FPGA development board.

Conclusion

The main goal of the work presented in this thesis has been to develop architectures intended to be used as hardware accelerators in real-time embedded image processing systems. The focus when developing the architectures has been to achieve low complexity, low memory requirement, and to preserve the raster scan order. Preserving the raster scan order both permits burst read- and write-operations to and from memories, and avoids the need for additional intermediate storage. By achieving these properties, the architectures are well suited to be deployed and integrated into any streaming data environment that allows internal embedded memories in the actual accelerators.

In this thesis, it is shown that binary erosion or dilation supporting static flat rectangular structuring elements of arbitrary size can be performed with a constant number of operations per output, i.e. two summations and two comparisons. In addition, by processing the padding in parallel, the FIFO otherwise needed at the input, may be removed, hence a minimal memory requirement is achieved. Furthermore, having the same memory requirement, it is shown that extending the support to locally adaptive structuring elements increases the computational complexity from constant to being proportional to the structuring element width. Calculating the distance transform by parallel erosions makes it possible to use a single pass resulting in increased throughput and a lower memory requirement. Finally, an architecture for connected component labeling based on contour tracing has also been developed. It is shown that this type of algorithm requires less memory and can guarantee a specific number of clusters not possible in SLO-based algorithms. In applications not requiring unique labels but rather that each cluster is visited only once, the memory requirement may be reduced even further, requiring only two bits per pixel yet still extracting the cluster's coordinates, size, and CoG.

By accelerating key operations of an automated surveillance system in dedicated hardware architectures, the amount of data processed in software can be significantly reduced. This is found to be an effective method to sustain real-time performance and may be adopted in future advanced high resolution and frame rate systems.

Bibliography

[1] L. Råde and B. Westergren, *Mathematics Handbook for Science and Engineering*, 2nd ed. Lund, Sweden: Studentlitteratur, 1998.

[2] R. Gonzalez, R. Woods, and S. L. Eddins, *Digital Image Processing using Matlab*. Upper Saddle River, NJ, USA: Prentice Hall, 2004.

[3] H. Jiang, H. Ardö, and V. Öwall, "Hardware accelerator design for video segmentation with multi-modal background modelling," in *Proc. of IEEE International Symposium on Circuits and Systems*, Kobe, Japan, May 2005.

[4] F. Kristensen, P. Nilsson, and V. Öwall, "Background segmentation beyond RGB," in *Seventh biennial Asian Conference on Computer Vision*, Hyderabad, India, Jan. 2006.

[5] Official AXIS Communications website. [Online]. Available: www.axis.com, 2008.

[6] J. M. Rabaey, A. Chandrakasan, and B. Nikolić, *Digital Integrated Circuit*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2003.

[7] J. L. Hennesey and D. A. Patterson, *Computer Architecture - A quantitative approach*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann, 2003.

[8] Official Berkley Design Technology, Inc website. [Online]. Available: www.bdti.com, 2008.

[9] P. Lapsley, J. Bier, A. Shoman, and E. A. Lee, *DSP Processor Fundamentals*. New York, NY, USA: John Wiley & Sons, 1997.

[10] Official Texas Instruments website. [Online]. Available: www.ti.com, 2008.

[11] W. MacLean, "An evaluation of the suitability of fpgas for embedded vision systems," *Computer Vision and Pattern Recognition, 2005 IEEE Computer Society Conference on*, vol. 3, pp. 131–131, 2005.

[12] J. Rodrigues, T. Olsson, L. Srnmo, and V. Öwall, "Digital implementation of a wavelet-based event detector for cardiac pacemakers," *IEEE Transactions on Circuits and Systems—Part I: Fundamental Theory and Applications*, vol. 52, no. 12, pp. 2686–2698, 2005.

[13] F. Kristensen, "Design and implementation of embedded video surveillance hardware," Ph.D. dissertation, Lund University, 2007.

[14] A. N. Netravali and B. G. Haskell, *Digital Pictures : Representation, Compression, and Standards*, 2nd ed. New York, NY, USA: Plenum, 1994.

[15] Paradiso-design. [Online]. Available: www.paradiso-design.net/video standards_en.html, 2008.

[16] C. A.P. and B. R.W., "Minimizing power consumption in digital cmos circuits," *Proceedings of the IEEE*, vol. 83, no. 4, pp. 498–523, 1995.

[17] T. Tuan, S. Kao, A. Rahman, S. Das, and S. Trimberger, "A 90nm low-power fpga for battery-powered applications," in *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, Monterey, California, USA, Feb. 2006, pp. 3–11.

[18] T. Tuan and S. Trimberger, "The power of FPGA arcitectures," *Xcell journal*, no. 60, pp. 12–15, 2007.

[19] S. Gupta and J. Anderson, "Optimizing FPGA power with ISE design tools," *Xcell journal*, no. 60, pp. 16–19, 2007.

[20] Y. Zhang, J. Roivainen, and A. Mämmelä, "Clock-gating in fpgas: A novel and comparative evaluation," in *The 9th EUROMICRO Conference on Digital System Design*, Cavtat, Croatia, Aug. 2006, pp. 584–590.

[21] B. Parhami, *Computer Arithmetic, Algorithms and Hardware Designs.* New York, NY, USA: Oxford University Press, 2000.

[22] J. Jönsson, *Våglära och Optik*, 2nd ed. Lund, Sweden: Teach Support, 1995.

[23] B. E. Bayer, "Color imaging array," U.S. Patent 3,971,065, 1976.

[24] C. L. Hardin, *Color for Philosophers.* Indianapolis, IN, USA: Hackett Publishing Company, 1988.

[25] Official CANON website. [Online]. Available: www.canon.com, 2008.

[26] D. Litwiller, "CMOS vs. CCD: Maturing technologies, maturing markets," *Photonics Spectra*, Aug. 2005.

[27] ——, "CMOS vs. CCD: Facts and fiction," *Photonics Spectra*, Jan. 2001.

[28] S. Marchand-Maillet and Y. Sharaiha, *Binary Digital Image Processing.* London, UK: Academic Press, 2000.

[29] A. Rosenfeld, "Connectivity in digital pictures," *Journal of the ACM*, vol. 17, no. 1, pp. 146–160, 1970.

[30] R. Gonzalez and R. Woods, *Digital Image Processing*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2002.

[31] V. Öwall, M. Torkelson, and P. Egelberg, "A custom image convolution DSP with a sustained calculation capacity of >1 GMAC/s and low I/O bandwidth," *Journal of VLSI Signal Processing*, vol. 23, no. 2-3, pp. 335–349, Nov. 1999.

[32] H. R.M., "Mathematical morphology and computer vision," in *Signals, Systems and Computers, 1988. Twenty-Second Asilomar Conference on*, vol. 1, 1988, pp. 468–479.

[33] H. Minkowski, "Volumen und oberflshe," *Mathematische Annalen*, vol. 57, pp. 447–495, 1903.

[34] H. Hadwiger, "Minkowskische addition und subtraktion belibiger punktmengen und die theoreme von Erhard Schmidt," *Mathematische Zeitschrift*, vol. 53, pp. 210–218, 1950.

[35] E. R. Dougherty and R. A. Lotufo, *Hands-on Morphological Image Processing.* Bellingham, WA, USA: Spie Press, 2003.

[36] J. Serra, *Image Analysis and Mathematical Morpohology.* New York, NY, USA: Academic Press, 1982.

[37] S. Fejes and F. Vajda, “A data-driven algorithm and systolic architecture for image morphology,” in *Proc. of IEEE International Conference on Image Processing*, vol. 2, Austin, Texas, Nov. 13-16 1994, pp. 550–554.

[38] T. Q. Deng and H. J. A. M. Heijmans, “Grey-scale morphology based on fuzzy logic,” *Journal of Mathematical Imaging and Vision*, vol. 16, no. 2, pp. 155–171, Mar. 2002.

[39] G. Louverdis and I. Andreanis, “Design and impelementation of a fuzzy hardware structure for morphological color image processing,” *IEEE Transaction on Circuits and Systems for Video Technology*, vol. 13, no. 3, pp. 277–288, 2003.

[40] C. Stauffer and W. E. L. Grimson, “Adaptive background mixture models for real-time tracking,” in *Proc. of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Ft. Collins, USA, June 23-25 1999.

[41] B. Kisačanin and D. Schonfeld, “A fast thresholded linear convolution representation of morphological operations,” *IEEE Transactions on Image Processing*, vol. 3, no. 4, pp. 455–457, July 1994.

[42] J. Goutsias and H. J. Heijmans, “Fundamenta morphologicae mathematicae,” *Fundamenta Informaticae*, vol. 41, no. 1-2, pp. 1–31, Jan. 2000.

[43] P. Soille, “Morphological operators,” in *Handbook of Computer Vision and Applications*, vol. 2. New York, NY, USA: Academic Press, 1999, pp. 627–682.

[44] G. Matheron, *Random Sets and Integral Geometry.* New York, NY, USA: John Wiley & Sons, 1975.

[45] H. Park and R. Chin, “Decomposition of arbitrarily shaped morphological structuring elements,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 1, pp. 2–15, 1995.

[46] G. Anelli and A. Broggi, “Decomposition of arbitrarily shaped binary morphological structuring elements using genetic algorithms,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 2, pp. 217–224, 1998.

[47] R. D. Yapa and H. Koichi, “A connected component labeling algorithm for grayscale images and application of the algorithm on mammograms,” in *Proceedings of the 2007 ACM symposium on Applied computing*, Seoul, Korea, Mar. 2007, pp. 146–152.

[48] A. Rosenfeld and J. Pfaltz, "Sequential operations in digital picture processing," *Journal of the ACM*, vol. 13, no. 1, pp. 471–494, Oct. 1966.

[49] M. B. Dillencourt, H. Samet, and M. Tamminen, "A general approach to connected-component labeling for aritrary image representations," *Journal of the ACM*, vol. 39, no. 2, pp. 253–280, Apr. 1992.

[50] W. Kesheng, O. Ekow, and S. Arie, "Optimizing connected components labeling algorithms," in *SPIE Int. Symposium on Medical Imaging*, San Diego, CA, USA, Feb. 2005.

[51] R. M. Haralick, "Some neighborhood operations," in *Real-Time/Parallel Computing Image Analysis.* New York, NY, USA: Plenum, 1981, pp. 11–35.

[52] K. Suzuki, I. Horiba, and N. Sugie, "Fast connected-component labeling based on sequential local operarations in the course of forward raster scan followed by backward raster scan," in *Proc. of 15th International Conference on Pattern Recognition*, Barcelona, Spain, Sept. 3-7 2000, pp. 434–437.

[53] L. D. Stefano and A. Bulgarelli, "A simple and efficeient connected components labeling algorithm," in *Proc. of 10th International Conference on Image Analysis and Processing*, Venice, Italy, Sept. 27-29 1999.

[54] S. D. Jean, C. M. Liu, C. C. Chang, and Z. Chen, "A new algorithm and its VLSI architecture design for connected component labeling," in *Proc. of IEEE International Symposium on Circuits and Systems*, London, England, Uk, May 30-June 2 1994.

[55] F. Chang, C. J. Chen, and C. J. Lu, "A linear-time component-labeling algorithm using contour tracing technique," *Journal of CVIU*, vol. 93, pp. 206–220, Feb. 2004.

[56] F. Chang and C. J. Chen, "A component-labeling algorithm using contour tracing technique," in *Proc. of 7th International Conference on Document Analysis and Recognition*, Edinburgh, Scotland, Uk, Aug. 3-7 2003.

[57] F. Kristensen, H. Hedberg, H. Jiang, P. Nilsson, and V. Öwall, "Hardware aspects of a real-time surveillance system," in *Sixth IEEE International Workshop on Visual Surveillance at ECCV*, Graz, Austria, 2006.

[58] J. Velten and A. Kummert, "FPGA-based implementation of variable sized structuring elements for 2D binary morphological operations," in

The First IEEE International Workshop on Electronic Design, Test and Applications, Jan. 29-31 2002, pp. 309–312.

[59] S. Y. Chien, S. Y. Ma, and L. G. Chen, "Partial-result-reuse architecture and its design technique for morphological operations with flat structuring element," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 15, no. 9, pp. 344–371, Sept. 2005.

[60] A. Źarandy, A. Stoffels, T. Roska, and L. Chua, "Implementation of binary and gray-scale mathematical morphology on the cnn universal machine," *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, vol. 45, no. 2, pp. 163–168, Feb. 1998.

[61] E. N. Malamas, A. G. Malamos, and T. A. Varvarigou, "Fast implementation of binary morphological operations on hardware-efficient systolic architectures," *Journal of VLSI Signal Processing*, vol. 25, pp. 79–93, 2000.

[62] H. Hedberg, F. Kristensen, P. Nilsson, and V. Öwall, "A low complexity architecture for binary image erosion and dilation using structuring element decomposition," in *Proc. of IEEE International Symposium on Circuits and Systems*, vol. 4, Kobe, Japan, May 2005, pp. 3431–3434.

[63] J. Velten and A. Kummert, "Implementation of a high-performance hardware architecture for binary morphological image processing operations," in *Proc. of IEEE International Midwest Symposium on Circuits and Systems*, vol. 2, Hiroshima, Japan, July 25-28 2004, pp. 241–244.

[64] R. Lam and C. Li, "A fast algorithm to morphological operations with flat structuring element," *IEEE Transactions on Circuits and Systems*, vol. 45, no. 3, pp. 387–391, Mar. 1998.

[65] A. G. Dempster and C. D. Ruberto, "Using granulometries in processing images of malarial blood," in *Proc. of IEEE International Symposium on Circuits and Systems*, Sydney, Australia, May 2001.

[66] P. Soille, "From binary to grey scale convex hulls," *Fundamenta Informaticae*, vol. 41, pp. 131–146, Jan. 2000.

[67] R. Lerallut, E. Decencière, and F. Meyer, "Image filtering using morphological amoebas," *Image Vision Comput*, vol. 25, no. 4, pp. 395–404, 2007.

[68] M. van Herk, “A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels,” *Pattern Recognition Letters*, vol. 13, no. 7, pp. 517–521, 1992.

[69] F. Lemonnier and J. Klein, “Fast dilation by large 1D structuring elements,” in *IEEE International Workshop on Nonlinear Signal and Image Processing*, Halkidiki, Greece, June 1995, pp. 479–482.

[70] M. V. Droogenbroeck and H. Talbot, “Fast computation of morphological operations with arbitrary structuring elements,” *Pattern Recognition Letters*, vol. 17, no. 14, pp. 1451–1460, 1996.

[71] O. Cuisenaire, “Locally adaptable mathematical morphology using distance transformations,” *Pattern recognition, the Journal of the pattern recognition society*, vol. 39, no. 3, pp. 405–416, 2006.

[72] G. Borgefors, “Distance transformations in digital images,” *Computer Vision, Graphics and Image Processing*, vol. 34, no. 3, pp. 344–371, 1986.

[73] F.-C. Shih and O. Mitchell, “A mathematical morphology approach to euclidean distance transformation,” *IEEE Transactions on Image Processing*, Apr. 1992.

[74] D. Paglieroni, “Distance transforms: properties and machine vision applications,” *CVGIP: Graphical Models and Image Processing*, 1992.

[75] C. Qing, Y. Xiaoli, and E. Petriu, “Watershed segmentation for binary images with different distance transforms,” in *The 3rd IEEE International Workshop on Haptic, Audio and Visual Environments and Their Applications*, Ottawa, Ontario, Canada, Oct. 2004.

[76] A. Rosenfeld and J. Pfaltz, “Sequential operations in digital picture processing,” *Journal of the ACM*, Oct. 1966.

[77] G. Borgefors, “Distance transformations in digital images,” *Computer Vision, Graphics and Image Processing*, 1986.

[78] N. Sudha, “A pipelined array architecture for euclidean distance transformation and its fpga implementation,” *Microprocessors and Microsystems*, 2005.

[79] P. Kwok, “A hardware approach to distance transform,” in *The 3rd International Conference on Image Processing and its Applications*, Warwick, UK, Sept. 1989.

[80] J. Takala, J. Viitanen, and J. Saarinen, "Hardware architecture for real-time distance transform," in *IEEE International Conference on Acoustics, Speech, and Signal Processing*, Phoenix, Az, USA, Mar. 1999.

[81] S. Y. Chien, S. Y. Ma, and L. G. Chen, "Partial-result-reuse architecture and its design technique for morphological operations with flat structuring element," *IEEE Transactions on Circuits and Systems for Video Technology*, Sept. 2005.

[82] Official Synopsys website. [Online]. Available: www.synopsys.com, 2008.

[83] The PETS 2001 data set, sequence from camera 1. [Online]. Available: www.cvg.cs.rdg.ac.uk/cgi-bin/PETSMETRICS/page.cgi?dataset, 2008.

[84] L. Yang and F. Algbregtsen, "Discrete Green's theorem and its application in moment computation," in *Int. Conf. on Electronics and Information Technology*, Beijing, China, Aug. 1994.

[85] Official Xilinx website. [Online]. Available: www.xilinx.com, 2008.

[86] Official sony website. [Online]. Available: http://bssc.sel.sony.com, 2008.

[87] Official IBM website. [Online]. Available: www.research.ibm.com/peoplevision/, 2008.

[88] I. Haritaoglu, D. Harwood, and L. Davis, "W^4: real-time surveillance of people and their activities," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, pp. 809–830, 2000.

[89] C. Stauffer and W. E. L. Grimson, "Learning patterns of activity using real-time tracking." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, pp. 747–758, 2000.

[90] R. Collins, A. Lipton, H. Fujiyoshi, and T. Kanade, "Algorithms for cooperative multisensor surveillance," *Proceedings of the IEEE*, vol. 89, no. 10, pp. 1456–1477, 2001.

[91] T. Zhao and R. Nevatia, "Tracking multiple humans in complex situations." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 9, pp. 1208–1222, 2004.

[92] W. Hu, T. Tan, L. Wang, and S. Maybank, "A Survey on Visual Surveillance of Object Motion and Behaviors." *IEEE Transactions on Systems, Man, and Cybernetics—Part C: Applications and Reviews*, vol. 34, no. 3, pp. 334–353, 2004.

[93] R. Aguilar-Ponce, J. Tessier, A. Baker, C. Emmela, J. Das, J. Tecpanecatl-Xihuitl, A. Kumar, and M. Bayoumi, "VLSI architecture for an object change detector for visual sensors," in *In proceding of IEEE Workshop on Signal Processing Systems Design and Implementation, SIPS*, Athens, Greece, 2005, pp. 290–295.

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

[95] P. Schumacher, K. Denolf, A. Chilira-RUs, R. Turney, N. Fedele, K. Vissers, and J. Bormans, "A scalable, multi-stream MPEG-4 video decoder for conferencing and surveillance applications," *Image Processing, 2005. ICIP 2005. IEEE International Conference on*, vol. II, pp. 886–889, 2006.

[96] R. Kordasiewicz and S. Shirani, "ASIC and FPGA implementations of H.264 DCT and quantization blocks," *Image Processing, 2005. ICIP 2005. IEEE International Conference on*, vol. III, pp. 1020–1023, 2006.

[97] R. Kleihorst, B. Schueler, A. Danilin, and M. Heijligers, "Smart Cameras Mote with High Performance Vision System," in *Workshop on Distributed Smart Cameras (DSC 06)*, Boulder, Colorado, USA, Oct. 2006.

[98] E. Ljung, E. Simmons, and R. Kleihorst, "Distributed Vision with Multiple Uncalibrated Smart Cameras," in *Workshop on Distributed Smart Cameras (DSC 06)*, Boulder, Colorado, USA, Oct. 2006.

[99] E. Ljung, E. Simmons, A. Danilin, R. Kleihorst, and B. Schueler, "802.15.4 Powered Distributed Wireless Smart Cameras Network," in *Workshop on Distributed Smart Cameras (DSC 06)*, Boulder, Colorado, USA, Oct. 2006.

[100] R. Gonzalez and R. Woods, *Digital Image Processing*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2002.

[101] J. Toyama, B. Brumitt, and B. Meyers, "Wallflower : principles and practice of background maintenance," *In Proc. IEEE International Conference on Computer Vision and Pattern Recognition*, 1999.

[102] C. Stauffer and W. E. L. Grimson, "Adaptive background mixture models for real-time tracking," in *Proc. of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Ft. Collins, USA, June 23-25 1999.

[103] G. Russo and M. Russo, "A novel class of sorting networks," *IEEE Transactions on Circuits and Systems—Part I: Fundamental Theory and Applications*, 1996.

[104] H. Jiang, H. Ardö, and V. Öwall, "Real-time Video Segmentation with VGA Resolution and Memory Bandwidth Reduction," *In Proc. of AVSS*, 2006.

[105] F. Kristensen, P. Nilsson, and V. Öwall, "Background Segmentation Beyond RGB," in *Seventh biennial Asian Conference on Computer Vision*, Hyderabad, India, Jan. 2006.

[106] J. Serra, *Image Analysis and Mathematical Morpohology, Vol 1.* Academic Press, 1982.

[107] H. Hedberg, F. Kristensen, P. Nilsson, and V. Öwall, "A low complexity architecture for binary image erosion and dilation structuring element decomposition," in *Proc. of IEEE International Symposium on Circuits and Systems*, Kobe, Japan, May 2005.

[108] K. Suzuki, H. Isao, and S. Noboru, "Linear-time connected-component labeling based on sequential local operations," *Journal of CVIU*, vol. 89, no. 1, pp. 1–23, Jan. 2003.

[109] H. Hedberg, F. Kristensen, and V. Öwall, "Implementation of labeling algorithm based on contour tracing with feature extraction," in *Proc. of IEEE International Symposium on Circuits and Systems*, New Orleans, USA, May 23-26 2007.

[110] Project website. [Online]. Available: www.es.lth.se/Digital_Surveillance, 2008.

[111] D. Magee, "Tracking multiple vehicles using foreground, background and motion models," *Image and Vision Computing*, vol. 22, pp. 143–155, 2004.

[112] Official JBIG website. [Online]. Available: www.jpeg.org/jbig/index.html, 2008.