UNIVERSITE PARIS-EST: Ecole doctorale MSTIC

and

University of West Bohemia in Pilsen

THESIS

to obtain the Doctor of Philosophy degree of the University Paris-Est
with specialization in Computer Sciences

Hardware Architectures for Morphological Filters with Large Structuring Elements

Jan Bartovský

presented on November 14th 2012

Composition of the Examination Committee:

Michal KOZUBEK	Professor, Masaryk University Brno	Reviewer
Olivier DÉFORGES	Professor, INSA Rennes	Reviewer
Mohamed AKIL	Professor, ESIEE Paris	Director
Vjačeslav GEORGIEV	Assoc. professor, UWB Pilsen	Director
Eva DOKLÁDALOVÁ	Assoc. professor, ESIEE Paris	Examinator
Petr DOKLÁDAL	Research engineer, ARMINES Paris	Examinator
Václav MATOUŠEK	Professor, UWB Pilsen	Examinator
Michel BILODEAU	Research engineer, ARMINES Paris	Examinator

Abstract

Bartovský, J. Hardware Architectures for Morphological Filters with Large Structuring Elements. University Paris-Est, University of West Bohemia. Directors: Mohamed Akil, Vjačeslav Georgiev.

This thesis is focused on implementation of fundamental morphological filters in the dedicated hardware. The main objective of this thesis is to provide a programmable and efficient implementation of basic morphological operators using efficient dataflow algorithms considering the entire application point of view.

In the first part, we study existing algorithms for fundamental morphological operators and their implementation on different computational platforms. We are especially interested in algorithms using the queue memory because their implementation provides the sequential data access and minimal latency, the properties very beneficial for the dedicated hardware. Then we propose another queue-based arbitrary-oriented opening algorithm that allows for direct granulometric measures. Performance benchmarks of these two algorithms are discussed, too.

The second part presents hardware implementation of the efficient algorithms by means of stream processing units. We begin with a 1-D dilation unit, then thanks to the separability of dilation we build up 2-D rectangular and polygonal dilation units. The processing unit for arbitrary-oriented opening and pattern spectrum is described as well. We also introduce a method of parallel computation using a few copies of processing units in parallel, thereby speeding up the computation. All proposed processing units are experimentally assessed in hardware by means of FPGA prototypes, and the performance and FPGA occupation results are discussed.

In the third part, the proposed units are employed in two diverse applications illustrating thus their capability of addressing performance-demanding, low-power embedded applications.

The main contributions of this thesis are: 1) new algorithm for arbitrary-oriented opening and pattern spectrum, 2) programmable hardware implementation of fundamental morphological operators with large structuring elements and arbitrary orientation, 3) performance increase obtained through multi-level parallelism. Results suggest that the previously unachievable, real-time performance of these traditionally costly operators can be attained even for long concatenations and high-resolution images.

Keywords

Mathematical morphology, morphological filter, hardware implementation, algorithm, FPGA.

Résumé

Bartovský, J. Architectures matérielles pour filtres morphologiques avec des grandes éléments structurants. Université Paris-Est, L'Université de Bohême de l'Ouest. Directeurs: Mohamed Akil, Vjačeslav Georgiev.

Cette thèse se concentre sur la mise en oeuvre d'implantations matérielles dédiées des filtres morphologiques fondamentaux, basés sur des itérations d'érosions/dilatations. L'objectif principal de cette thèse est de proposer une mise en oeuvre efficace et programmable de ces opérateurs en utilisant des algorithmes en flot de données et considérant les besoins applicatifs globaux.

Dans la première partie, nous étudions les algorithmes existants pour les opérateurs morphologiques fondamentaux et leur réalisation sur des différentes plates-formes informatiques. Nous nous intéressons plus particulièrement à un algorithme basé sur la fille d'attente pour la mise en oeuvre de la dilatation car il permet de réaliser l'accès séquentiel aux données avec une latence minimale, ce qui est très favorable pour le matériel dédié. Nous proposons ensuite un autre algorithme réalisant l'ouverture morphologique, sous angle arbiraire, basé sur le même principe d'une fille d'attente, permettant d'obtenir directement des mésures de granulométrie.

La deuxième partie présente la mise en oeuvre matérielle des algorithmes efficaces au moyen d'unités de traitement à flot de données. Nous commençons par l'unité de dilatation 1-D, puis grâce à la séparabilité de la dilatation nous construisons des unités 2-D rectangulaire et polygonale. L'unité de traitement pour l'ouverture orientée et spectre modèle est ainsi décrite. Nous présentons également une méthode de parallélisation de calcul en dupliquant des unités de traitement. Toutes les unités de traitement proposées sont évaluées expérimentalement par la réalisation des prototypes à base de circuits programmables (FPGA), et les résultats en termes d'occupation de surface et de vitesse de traitement sont discutés.

Dans la troisième partie, les unités de calcul proposées sont utilisées dans deux applications différentes, illustrant ainsi leur capacité de répondre exigeances des applications embarquées a basse consommation.

Les principales contributions de cette thèse sont : i) proposition d'un nouvel algorithme d'ouverture sous angle arbitraire, ii) réalisation des architectures matérielles dédiées et programmables d'opérateurs morphologiques fondamentaux à l'élément strucurant large et sous angle arbitraire ; iii) amélioration de la performance obtenue grâce à l'exploitation de plusieurs niveaux de parallélisme. Les résultats suggèrent que les performances de temps réel précédemment irréalisable de ces opérateurs traditionnellement coûteux peuvent être atteintes même pour des longues concaténations d'opérateurs ou des images à haute résolution.

Mots clefs

Morphologie mathématique, filtre morphologique, la mise en oeuvre du matériel, algorithme, FPGA.

Anotace

Bartovský, J. Číslicové architektury morfologických filtrů s velkými strukturujícími elementy. Univerzita Paris-Est, Západočeská univerzita v Plzni. Vedoucí: Mohamed Akil, Vjačeslav Georgiev.

Tato práce se zabývá implementací základních morfologických filtrů v číslicových obvodech. Hlavním úkolem této práce je vytvořit programovatelné a efektivní číslicové implementace základních morfologických operátorů za použití výpočetně efektivních algoritmů. Důležitým hlediskem je chování celé aplikace složené z více operátorů.

V první části jsou prostudovány existující algoritmy základních morfologických operátorů a jejich realizace na vhodných výpočetních platformách. Z existujících algoritmů se pro implementaci dilatace jako nejvhodnější jeví algoritmy využívající paměť fronty. Důvodem jsou vhodné vlastnosti pro číslicové obvody, sekvenční přístup k datům a minimální latence. Posléze navrhneme a popíšeme vlastní algoritmus morfologického otevření využívající stejnou paměť fronty, který umožňuje výpočet pod libovolným úhlem a přímý výpočet granulometrie. Výkonnostní parametry obou dvou algoritmů jsou zde diskutovány.

Druhá část obsahuje popis obvodové implementace těchto algoritmů ve formě výpočetních jednotek. Napřed vytvoříme 1-D jednotku dilatace, pomocí které díky rozložitelnosti dilatace vytvoříme 2-D jednotku dilatace pomocí obdélníků a polygonů. Návrh výpočetní jednotky algoritmu orientovaného otevření a spektra vzorů je také uveden v této části. Abychom dosáhli vyššího výpočetního výkonu, použijeme metodu paralelního výpočtu, která využívá několika kopií použitých výpočetních jednotek pracujících ve stejném čase. Všechny navržené výpočetní jednotky byly experimentálně ověřeny v číslicových obvodech typu FPGA, výsledky výpočetního výkonu a potřebné plochy čipu jsou diskutovány.

Ve třetí části jsou navržené výpočetní jednotky použity ve dvou různých aplikacích, čímž ilustrují svoji využitelnost v embedded aplikacích vyžadujících velmi velký výpočetní výkon a zároveň nízkou spotřebu.

Hlavní přínosy této práce jsou následující: 1) vlastní algoritmus morfologického otevření a spektra pod libovolným úhlem, 2) číslicová implementace základních morfologických operátorů filtrů s velkými a libovolně orientovanými strukturujícími elementy, 3) zvýšení výpočetního výkonu díky víceúrovňovému paralelnímu výpočtu. Dosažené výsledky ukazují, že výpočet těchto náročných operátorů v realném čase, kterého dosud nebylo možné docílit, je nejen dosažitelný ale i udržitelný pro dlouhé zřetězené filtry a vysoké rozlišení zpracovávaných obrazů.

Klíčová slova

Matematická morfologie, morfologický filtr, obvodová implementace, algoritmus, FPGA.

Acknowledgment

I am glad to express hearty appreciation to my thesis directors Mohamed Akil and Vjačeslav Georgiev for their time, patience, consideration, and support by thoughts and motivation. Their kind and selfless help in both thought and bureaucratic crises was highly appreciated.

I am very grateful to my thesis advisors Eva Dokládalová and Petr Dokládal for their tremendously kind, insight, endless support. Without their everyday effort in enhancing my work the thesis could not have been completed in time.

I am very thankful to Michel Bilodeau who accepted me into the FREIA project, and thereby opened the doors to the Centre of Mathematical Morphology for me.

I am grateful to Michel Couprie, Thierry Grandpierre, Laurent Perroton, Hugues Talbot, Eric Llorens, Christine Auger, and the other members of the A3SI team at ESIEE Paris for friendly welcome and nice working environment.

My internship in the Centre of Mathematical Morphology was very inspiring and fruitful. The credit goes to all the staff: Fernand Meyer, Dominique Jeulin, Serge Beucher, Étienne Decenciére, Jesús Angulo, Beatriz Marcotegui, Petr Matula, Guillaume Thibault, Catherine Moysan, and the others.

I would like to thank all my fellow students and colleagues Václav Kraus, Aleš Krutina, Radek Šalom, Michael Holík, Martin Poupa, Nicolas Ngan, Imran Taj, Ramzi Mahmoudi, Rostom Kachouri, Jeoffroy Marbeaux, Oussama Feki, Xiwei Zhang, Vincent Morard, Santiago Velasco, and many others.

I should also thank the members of examination committee Olivier Déforges, Michal Kozubek, and Václav Matoušek for their time necessary to read and understand the manuscript.

The most eminent thanks goes to my parents who were unconditionally encouraging me during the entire university study to achieve the goals I can be proud of.

Glossary

List of Symbols

α	Orientation of the line SE
φ	Closing
$\complement$	Complementation operator
δ	Dilation
ε	Erosion
I, f	Discrete image
$\mathbb{Q}$	Set of rational numbers
mod	Remainder of integer division
γ	Opening
$\oplus$	Minkowski addition
PD	Parallelism Degree
PS	Pattern spectrum
$\mathbb{R}$	Set of real numbers
x, y	Element of set
X, Y	Set
$\mathbb{Z}$	Set of integer numbers
ζ	Serial concatenation of operators
AQ	Array of Queues
B	Structuring element
l	Length of the SE
$N \times M$	Width×height of the image
Q	Queue
$W \times H$	Width×height of the SE
$\mathcal{O}_{\text{image}}()$	Computational complexity against the whole image
$\mathcal{O}()$	Computation complexity

List of Abbreviations

n-D	n-Dimension(al)
ALU	Arithmetic Logic Unit
AR	Average pixel Rate
ASF$^{\lambda}$	λ-order Alternate Sequential Filter
ASIC	Application-Specific Integrated Circuit
bpp	Bits Per Pixel
BRAM	Block RAM memory
CAM	Content Addressable Memory
CCT	Connected Component Tree
CPU	Central Processing Unit
DMA	Direct Memory Access
FIFO	First In First Out

FPGA	Field-Programmable Gate Array
FPS	Frame-Per-Second ratio
FREIA	Framework for Embedded Image Applications
FSM	Finite State Machine
GPP	General-Purpose Processor
GPU	Graphics Processing Unit
HDTV	High-Definition Television
HGW	Dilation algorithm proposed by van Herk, and Gil and Werman
HW	Hardware
LUT	Look-Up Table
MMB	Mathematical Morphology Blocks
MPMC	Multi-port Memory Controller
PLB	Peripheral Local Bus
PoC	Processor-on-chip
PRR	Partial-Result Reuse
RAM	Random Access Memory
SE	Structuring Element
SIMD	Simple Instruction, Multiple Data
SoC	System-on-chip
SSE	Streaming SIMD Extension
TEMAC	Tri-Mode Ethernet Media Access Controller
UHDTV	Ultra High-Definition Television
VFBC	Video Frame Buffer Controller
VHDL	Very-high-speed integrated circuits Hardware Description Language
VLIW	Very Long Instruction Word

Contents

1 Introduction

Contents

In this thesis we focus on hardware implementation of fundamental algorithms of mathematical morphology. Mathematical morphology is a popular image processing framework providing a complete set of tools for filtering, multi-scale image analysis, or pattern recognition. It has been used in a number of applications, including biomedical and medical imaging, video surveillance, industrial control, video compression, stereology or remote sensing since its very first appearance in the late 1960's, see [Matheron 1975] [Serra 1988] [Serra 1992].

Considering the hardware implementation context, several different trends have been observed. A recent technological advance of imaging sensors stimulated the development of applications by means of high-resolution images that became a standard. This trend of increasing resolution is widely expected to carry on, such as UHDTV [ITU-R 2012], the public broadcasting of which may be ready by 2016. Needless to say large images impose challenging requirements on the computation platform in terms of both performance and memory.

On the other hand, the industrial context often induces severe real-time constraints on applications. As these demanding image-interpretation applications requires a high correct-decision liability, robust but costly multi-criteria and/or multi-scale analyses are used. Provided that slow image processing may deteriorate some industrial production, the latency and computational performance are of high interest in this context.

In embedded systems, the most important concerns are low power consumption (and consequently low heat dissipation) and small resources occupation, which allows for better embedding. All these considerations combined together infer overwhelming requirements on the architecture of polyvalent processing units addressing many different contexts.

1.1 Applications of Mathematical Morphology

Examples of the most frequent morphology operators and their applications follow below.

- *Filters:* Filters are low-level vision operators and serve either to eliminate noise that deteriorates an image by some undesired artifacts [Heijmans 1997] [Serra 1992] [Maragos 2005], or to simplify image topology in order to make further processing easier. Morphological filters are commonly used in preprocessing stage of many complex higher-level operators, such as image compression and image segmentation [Gorpas 2009].

- *Granulometry:* Granulometry (so-called size distribution) measures distribution of object size in a population of objects [Matheron 1975] [Maragos 1989]. It can be considered as an ordered set of operators—sieves—each of which allows only objects larger than a given size to pass. [Urbach 2004] used granulometry of an inner cell texture for automatic diatom cell identification and classification. [Bagdanov 2002] utilized granulometry in genre classification of printed documents, and more recently [Karas 2012b] took advantage of arbitrary-oriented granulometries for rotation detection of music sheet scans.

- *Image enhancement:* Image enhancement is a common technique to improve some visual features with respect to different criteria, e.g., contrast enhancement, toggle mapping [Serra 1989]. For another examples, [Zhang 2011] detected microaneurysms on eye fundus images, and [Wei 2007] used a multiscale top-hat transformation to locally increase contrast of orthopaedic X-ray images, which were then easier to read.

- *Classification:* In general, classification aims at identifying to which of a set of classes a new observation belongs. Mathematical morphology has been mainly used in classification of images (or sections of images) with respect to spatial features. [Moore 2007] used mathematical morphology for the classification of astronomical objects, both for star/galaxy differentiation and galaxy morphology classification.

- *Segmentation:* Segmentation is another key tool of image processing applied to a large number of problems. For example, see contouring blobs of proteins in an electrophoresis gel, separation of overlapping grains, both [Beucher 1992].

- *Statistical learning:* A selected set of morphological operators can be separately applied to an image and used as a vector of descriptors for pixelwise statistical learning, see [Vapnik 1995] for the domain introduction. [Cord 2007] devised a method for segmentation of random textures using a vector of 126 morphological descriptors.

1.1.1 Vision Application Constraints

From the applications in literature we can recognize the most common constraints shared among various vision applications as follows:

- *Real-time processing:* We understand as real-time processing the capability of processing data at a rate at least equal to the acquisition/input data rate. This constraint is closely related to performance, however, the exact value depends on a given application. For example, the common video camera formats specify the minimum frame-per-second (fps) performance to 25, 30, etc., respectively; pixel detectors take from 30 to 100 fps; and for high-end industrial applications, even more diverse values (1–1000 fps) may be encountered. Clearly, the values above reflect complexity of applications; however, the higher performance of computation platform helps us to meet the application-specific real-time constraint.

- *Latency:* The excessive latency of computation has two main implications on an application. First, the larger latency usually implies the larger memory requirements, either as an image storage (input, intermediate, output) or as a working memory. Second, the excessive latency may be limiting for certain applications that need results as soon as possible for further processing, e.g., iterative reconstruction, etc.

- *High-definition resolution:* The resolution of images varies within a large range from a small resolution 256×256 of the pixel detector, through distinct video format resolutions (nowadays standards 640×480–1920×1080, or even UHDTV 7680×4320), up to industrial sensors with resolution of tens of megapixels. The high resolution impacts computation performance, so it is more challenging to achieve real-time processing, and memory requirements. In non-destructive testing by machine vision, searching for small defects (orders of micrometers) in large-size pieces (orders of square meters), one easily encounters extreme resolution requirements.

- *Reliability, power consumption:* In certain industrial applications such as road monitoring and obstacle detection [Beucher 1995] or track autonomous following [Marion 2004], high system reliability and low power consumption are very important constraints. This constraint is typical for all embedded systems.

All these constraints combined together infer overwhelming (and even sometimes contradictory) requirements on the computational platform. We recognize three general computation platforms suitable for such vision applications: general-purpose processor (GPP, so-called CPU), graphics processing unit (GPU), and dedicated hardware. The conveniences of these platforms for vision applications are discussed in the following section.

1.2 Roles of Dedicated Hardware for Vision Applications

Probably the most popular and common computation platform is a general-purpose processor that appears in a variety of general-purpose computing devices: personal computers, multi-core workstations, or many-core clusters. Despite the wide spread, and recent technological advances, the performance of the GPP is rather low given some complex vision application as it is partly sacrificed in favor of large universality. The computation is carried out sequentially in a GPP (there are exceptions like the SIMD instruction set, superscalar CPUs etc.).

The second platform suitable for vision applications is graphics processing unit (GPU). A GPU consists of many light-weight processors interconnected in given hierarchy, each of which executes several threads of the application code. Hence, this platform provides some parallelism via many threads being computed at the same time. However, the threads are processed sequentially, and the inter-thread communication is penalizing. The GPU platforms can achieve higher performance than GPP for applications that takes advantage of thread-wise parallel computation, thus weakening universality.

The dedicated hardware (we focus chiefly on FPGAs, however, the following holds true for ASICs as well) goes even farther. It provides the designer by a large amount of logic resources (that carry out arithmetic, logic operations, etc.) and interconnection resources, and let him to decide the way how the processing architecture should be assembled. Such an approach results in a great opportunity for parallelism of different kinds (spatial, temporal) that leads to the highest performance for certain applications. These target applications are supposed to involve rather dense numerical computation per datum with less conditional jumps and context switching, such as digital filters, video coding/decoding, etc.. Obviously, most vision applications (including mathematical morphology) satisfies the first presumption. In order to comply with the latter, we have to beware of very complex applications or applications that need large diversity of operators.

Notice that for changing the computation context, like in the case of complex or general-purpose applications, the dedicated hardware must be either reconfigured, see [Hauck 2007] or [Gokhale 2010] for FPGA reconfiguration, or it must possess computation resources for every used context and programmable interconnection. Either way, it results in inefficient hardware utilization.

The dedicated hardware is a platform that can address each one of the application constraints mentioned above. On the other hand, the real power of this platform can only be exploited, above all, in single-purpose applications, which attain efficient resources utilization. We see mainly two target application groups of dedicated hardware as follows:

- *High-performance single-purpose applications:* That is applications demanding a huge computational power tailored to only one purpose. Then dedicated hardware allows for real-time processing even in the most demanding applications, for which the other platforms are not powerful enough. Dedicated hardware is definitely capable of improving the overall performance.

- *Low-power embedded applications:* The other group of dedicated hardware applications stand on the opposite side of performance scale. For low-power embedded applications the computational performance is not the main measure, but it is power consumption or resources occupation. In this field dedicated hardware is better choice than other platforms thanks to low power and area demands.

Considering the other two constraints, hardware platforms improves them as well. The hardware platform clearly allows a designer to devise an architecture with zero additional latency, or an architecture supporting large images with no penalization (unless all resources are used). On the other hand, dedicated hardware can hardly constitute a general-purpose computation platform for a complete set of vision applications. Large polyvalence of applications would gravely reduce computation performance due to inefficient utilization of hardware resources.

1.3 Contributions of the Thesis

The aim of the thesis is to propose hardware implementation of adaptable processing units for vision applications based on mathematical morphology. The thesis develops in three main stages: algorithms of mathematical morphology, hardware implementation of processing units, and applications.

First, we evaluate algorithms for low-level mathematical morphology and recognize the *Dokládal* algorithm (published in [Dokládal 2011]) to be the best choice for hardware implementation. As the first contribution, we enrich the family of supported structuring elements by inclined lines that can form regular polygons. For the inclined lines the computation of dilation proceeds along an inclined discrete line the coordinates of which are determined using Bresenham line algorithm [Bresenham 1965]. A similar approach was used by [Soille 1996] and [Morard 2011] but our solution preserves sequential access to data whatever the inclination angle, a very beneficial property of the algorithm.

After that we propose an original algorithm for arbitrary-oriented 1-D opening and pattern spectrum called *streaming peak elimination*. Even though opening is commonly obtained as a concatenation of erosion and dilation, direct computation of opening is faster and easier to implement. The proposed algorithm is targeted to hardware and GPU implementation. The performance benchmark reveals that our algorithm outperforms all other algorithms using graphics cards.

Second, we propose the dedicated hardware implementation of the *Dokládal* algorithm as a fully programmable 1-D dilation processing unit supporting different orientations, which is used as a building brick in concatenations of any length. This *inter-operator parallelism* is illustrated on the 2-D rectangular and polygonal dilation processing units. Then, we introduced a method of parallel computation that uses a few copies of processing units in parallel, each of which however processes its dataflow sequentially. This *intra-operator parallelism* almost linearly increases the performance of both rectangular and polygonal processing units. Compared to other recent architectures, our processing units outperform the others for structuring elements larger than 3×3. The difference is even more evident in the case of compound operators, for instance serial filters.

In a later part, we implemented the *streaming peak elimination* algorithm as a fully programmable 1-D opening and pattern spectrum processing unit supporting arbitrary orientation. This processing unit allows for *inter-operator parallelism* of complex morphological operators demanding multiple orientation analysis, such as oriented pattern spectra or image enhancement.

Third, we utilize the proposed processing units in two applications. We integrate the units into the FREIA (Framework for Embedded Image Applications, [FREIA 2011]) platform that is supposed to address the most computation performance demanding vision applications. From the FREIA viewpoint, the main contribution of the proposed architectures is efficient computation of large and oriented SEs. The second application classifies high-energy particles recorded by the Timepix detector. We show that a basic classification of particle shapes can be realized in a streaming manner in an embedded device using the proposed processing units.

Considering performance of the designed computing units, this scalable and programmable computing platform allows us to obtain previously unachievable, real-time performances for the traditionally costly morphological operators. Along with ability to implement large SEs without decomposition, it opens the accessibility of advanced morphological operators in industrial systems. The number of examples includes the on-line production control, aging material defectoscopy, etc. Thanks to its high degree of universality, it shall allow application developers to utilize this framework instead of an expensive ad-hoc development.

1.4 Outline

The rest of the manuscript is organized as follows. Chapter 2 recalls the basic terminology of image processing and fundamental operators of the mathematical morphology.

In Chapter 3, we present the review of the literature on advances in basic morphology algorithms, chiefly dilation and opening, and decides which algorithm has the most pleasant properties for hardware implementation. We also outline advances in morphology implementation. We discuss especially in detail existing implementations on general-purpose processors, graphics processing units, and dedicated hardware.

Chapter 4 is devoted to the thorough description of the selected morphological algorithm used in implementation later. We also propose an original algorithm for morphological opening in this chapter. Performance benchmarks of these algorithms with respect to other state-of-the-art algorithms are mentioned using different computation platforms.

In Chapter 5, we present hardware implementation of basic morphological operators using efficient algorithms that have been chosen in the previous chapters. The proposed programmable processing units can be used as basic bricks to build up more complex operators. We also shows how to speed up the performance by introducing two levels of parallelism. Chapter 6 presents experimental performance and FPGA implementation results of the proposed processing units with respect to various properties. The proposed architectures are also compared with other state-of-the-art hardware implementations in order to evaluate the contribution.

Chapter 7 contains description of two practical applications that utilizes the proposed processing units. The purpose of this work is to illustrate usability of the units both high-performance and low-power, embedded applications. Finally, chapter 8 concludes the manuscript and outlines the perspectives and the undergoing work.

2 Fundamental Operators of Mathematical Morphology

Contents

In the following sections, we review the basic image processing terminology used in the thesis. The definition of the most important morphological operations follows below.

First of all, we focus on a discrete (digital) image. The transformation of continuous image into discrete image is called digitalization and consists of sampling (i.e., discretization of spatial coordinates) and value quantization. Let X be a countable set called support, and Y a countable set of defined values. We consider a family of *discrete images* I an application of some function $f : X \to Y$ where X is usually a rectangular domain, $X \subset \mathbb{Z}^2$ or $X \subset \mathbb{Z}^3$. An element $x \in X$ is called an image point. Depending on the definition of the support X, we call an image point a *pixel* if $X \subset \mathbb{Z}^2$, or a *voxel* if $X \subset \mathbb{Z}^3$. The value y of a pixel x is defined as $y = f(x)$. We can further specify the type of image according to the set of image values Y. We call f a binary image if Y contains exactly two elements. The grey-scale images consider $Y \subset \mathbb{Z}$, or even Y being a subset of a set of floating-, fixed-point numbers (such sets are countable). In general, all image operations $\Xi : I \to I, g = \xi(f)$ can be broken down with regard to the influence scope into three basic types:

- Pixel operations. The output $g(x)$ depends only on the input at the very same position $f(x)$, it is independent of all other pixels in the image. For instance, threshold, contrast addition, subtraction, stretching are pixel operations.
- Neighborhood (local) operations. The output $g(x)$ depends on a given set of input pixels $f\left(\mathcal{P}(x)\right); \mathcal{P}(x) \subset X$ ($\mathcal{P}(X)$ is a subset of X) often surrounding x, hence called a neighborhood of x. The examples of neighborhood operations are, e.g., various filters (morphological, smoothing, Laplacian), convolution, gradient, sharpening, etc.
- Global operations. The output $g(x)$ depends on the entire input image $f(X)$. The global operations are designed to reflect some statistical information of the image, e.g., distance transformation, histogram equalization, or they

extract some hierarchical information, e.g., connected components trees, segmentation, scene parsing.

Morphological operators aim at extracting some relevant spatial information from an image, see [Serra 1982, Matheron 1975] for extensive information. Since the image is considered to be a set X, it can be achieved by probing the image with another set of a known shape. Hereafter, we call the probe *Structuring Element* (SE), in other literature sometimes called window or kernel. The shape of the SE has very significant influence on the result of any morphological operation, and therefore, the choice of the shape and size is often made according to some a priori knowledge of the image geometry.

Although a SE may be generally $n+1$-dimensional for n-dimensional images, we focus on n-dimensional SEs. These SEs are referred to as *flat* because they have only 2 dimensions in the case of 2-dimensional image, which is the most common. The $n+1$-dimensional SEs are called *volumic*, *non-flat*, or *gray-scale*, and omitted in this memory due to high computation complexity and restricted usage. Regardless the type, each SE is equipped by an origin that allows positioning of the SE at a given point of an image.

2.0.1 Erosion and Dilation

The erosion and dilation are fundamental operations of mathematical morphology; they answer to the most obvious question while probing an image (the following questions quote [Soille 2003]).

The binary dilation answers the question *"Does the structuring element hit the set?"*. The result set contains the points where the answer is affirmative. The binary dilation of a set X by a SE B is denoted by $\delta_B(X)$ and it is defined as

$$\delta_B(X) = \{x \mid \widehat{B}(x) \cap X \neq \emptyset\} \tag{2-1}$$

where the SE B is considered to be flat, i.e., $B \subset \mathbb{Z}^2$ (translation-invariant), equipped with an origin $x \in B$. The transposed SE $\widehat{B}$ is equal to the geometric reflection of B around the origin

$$\widehat{B} = \{x \mid -x \in B\}. \tag{2-2}$$

The binary dilation can be also defined by means of Minkowski set addition, such as

$$\delta_B X = X \oplus B = \bigcup_{b \in B} X_b \tag{2-3}$$

where for set X and element b, the subscript X_b denotes translation of X by b.

This latter definition allows for direct extension to gray-scale images (functions). The definition of the gray-scale dilation by a flat SE also exists in two versions. First, the extension to functions of the Minkowski set addition, such as

$$[\delta_B(f)](x) = \left[\bigvee_{b \in B} f_b\right](x) \tag{2-4}$$

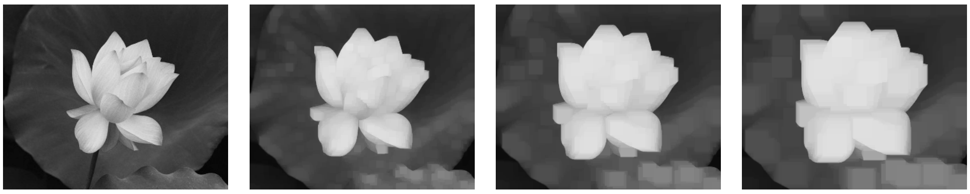
where f_b denotes translations of f by vectors $b \in B$. The second definition is obtained by extension to functions of the set intersection/inclusion given by

$$[\delta_B(f)](x) = \wedge\{v \in Y \mid \widehat{B} + v \geq f\}. \tag{2-5}$$

The implementation of the gray-scale dilation essentially consists of searching the maximum of f within the scope of B such as

$$[\delta_B(f)](x) = \max_{b \in B} [f(x+b)] \tag{2-6}$$

The example of an image dilated by SEs of various sizes is displayed in Fig. 2.1. At first sight, the light regions, which have high gray-level values, are stretched out.



(a) Input image (b) Dilation by 11×11 (c) Dilation by 21×21 (d) Dilation by 31×31

Figure 2.1: Example of images processed by dilation with various SEs. (a) input image f, (b) dilated by SE 11×11, (c) dilated by SE 21×21, (d) dilated by SE 31×31.

The output of the binary erosion is a set of points where the answer to *"Does the structuring element fit the set?"* is positive. The binary erosion of a set X by a structuring element B is denoted by $\varepsilon_B(X)$ and it is defined a

$$\varepsilon_B(X) = \{x \mid B(x) \subset X\} \tag{2-7}$$

where the SE B is considered to be flat equipped with an origin $x \in B$. The binary erosion can be also defined by means of Minkowski set addition, such as

$$\varepsilon_B X = X \ominus B = \bigcap_{b \in \widehat{B}} X_b \tag{2-8}$$

where for set X and element b, the subscript X_b denotes translation of X by b.

This latter definition allows for direct extension to gray-scale images (functions). The definition of the gray-scale erosion by a flat SE also exists in two versions. First, the extension to functions of the Minkowski set addition, such as

$$[\varepsilon_B(f)](x) = \left[\bigwedge_{b \in \widehat{B}} f_b \right](x), \tag{2-9}$$

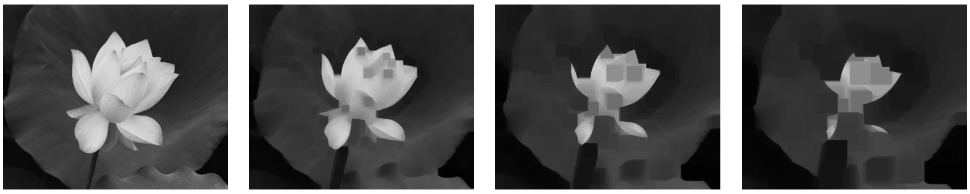
where f_b denotes translations of f by vectors $b \in B$ computed as $f(x+b)$. The second definition is obtained by extension to functions of the set intersection/inclusion given by

$$[\varepsilon_B(f)](x) = \vee\{v \in Y \mid B + v \leq f\}. \tag{2-10}$$

The implementation of the gray-scale erosion essentially consists of searching the minimum of f within the scope of B such as

$$[\varepsilon_B(f)](x) = \min_{b \in \widehat{B}} [f(x+b)] \tag{2-11}$$

The example of erosion by SEs of various sizes is displayed in Fig. 2.2. Erosion expands the dark regions as those have small gray-level value.



(a) Input image (b) Erosion by 11×11 (c) Erosion by 21×21 (d) Erosion by 31×31

Figure 2.2: Example of images processed by erosion with various SEs. (a) input image f, (b) eroded by SE 11×11, (c) eroded by SE 21×21, (d) eroded by SE 31×31.

Both dilation and erosion share some important properties. First of all, they are dual operation to each other. It means that an erosion of an image is equal to complementation of the dilation of the complemented image (and the other way around). Complementation is a basic set operator, and complementation of some image f, denoted as $\complement f$, is defined for each pixel x as the maximum value of the data type used for storing the pixel $t_{\max}$ minus the value of image f at position x, such as

$$\complement f(x) = t_{\max} - f(x). \tag{2-12}$$

From the implementation point of view, one can omit dealing with an erosion if he has a dilation and some light-weighted, efficient complementation operator $\complement$,

$$\delta_B(f) = \complement \varepsilon_{\widehat{B}} \complement(f). \tag{2-13}$$

Dilation and erosion also form an adjunction pair

$$\delta(X) \leq Y \Leftrightarrow X \leq \varepsilon(Y). \tag{2-14}$$

The adjunction is necessary to obtain properties allowing for combining dilation and erosion to form filters.

Other properties are increasingness, ordering relations, invariance to translation, distributivity, etc., the thorough description of which can found in [Soille 2003].

2.0.1.1 Composition of Structuring Elements

The last, and very important, property of erosion and dilation is SE composition (sometimes referred to as SE decomposition). The composition property claims that a sequence of dilations (or erosions) is equivalent to only one operation by the SE equal to the Minkowski addition $\oplus$ of both original SEs, see (2-15). This property is very useful because it allows us to compose more spatially complex SEs using elementary SEs that often decreases the order of computation complexity. Figure 2.3 displays the composition of rectangle, hexagon, and octagon from lines. This decomposition is often used to approximate circle SEs.

$$\delta_{B_1}\delta_{B_2}(f) = \delta_{B_1 \oplus B_2}(f) \tag{2-15}$$

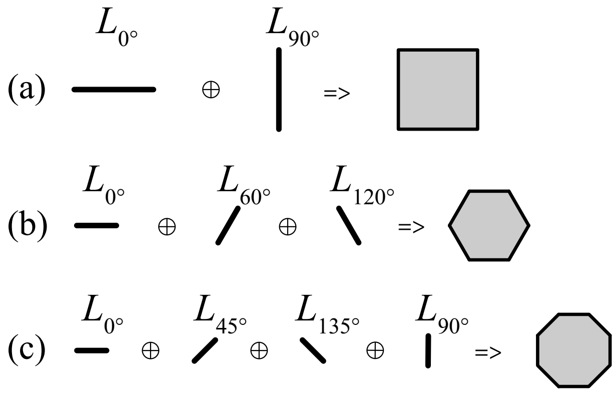
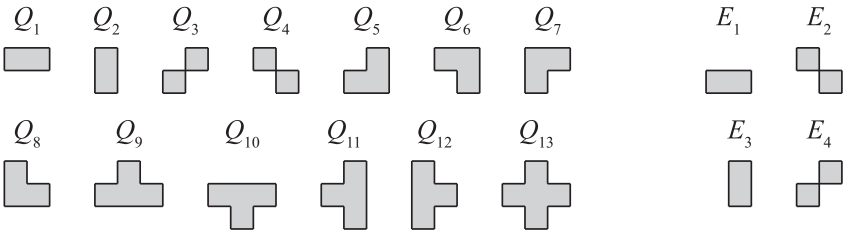


Figure 2.3: Examples of regular polygon SE composition: (a) rectangle, (b) hexagon, (c) octagon.



(a) Xu (b) Normand

Figure 2.4: Classes of primitive SEs. Any 8-convex polygon SE is decomposable into either: (a) Xu class, or (b) Normand class while using union along with $\oplus$.

Another principle of SE decomposition was proposed by [Xu 1991] (similar to [Zhuang 1986]). It claims that any 8-convex polygon (convex on 8-connectivity grid, hence 8-convex) is decomposable into a class of 13 nontrivial indecomposable convex polygonal SEs $Q_1 - Q_{13}$ shown in Fig. 2.4 (a). [Normand 2003] reduces the class of shapes to only four 2-pixel SEs, see Fig. 2.4 (b), by allowing the union operator to take place in SE decomposition. For instance, Q_{12} by Xu is obtained as $(E_3 \oplus E_3) \cup E_4$ by Normand.

2.0.2 Opening and Closing

Concatenations of a dilation and an erosion form elementary filters called opening and closing, see Fig. 2.5.

The binary opening preserves the whole set of the SE if the SE fits into an image. The binary opening by a flat SE B is denoted by $\gamma_B(X)$ and defined as

$$\gamma_B(X) = \bigcup_x \{B(x) \mid B(x) \subseteq X\}. \tag{2-16}$$

The gray-scale opening is defined as the union of all SEs that fit under the graph of a function f such as

$$\gamma_B(f) = \vee\{B + v \leq f\}, \tag{2-17}$$

and it can be implemented by

$$\gamma_B(f) = \delta_{\widehat{B}}[\varepsilon_B(f)]. \tag{2-18}$$

The result of the binary closing filter does not contain any point of SEs that fit the background set. The binary closing of a set X by a flat SE B is denoted by $\varphi_B(X)$ and defined as

$$\gamma_B(X) = \complement\left[\bigcup_x \{B(x) \mid B(x) \subseteq \complement X\}\right]. \tag{2-19}$$

The gray-scale closing is defined as

$$\gamma_B(f) = \wedge\{\widehat{B} + v \geq f\}, \tag{2-20}$$

and it can be implemented by

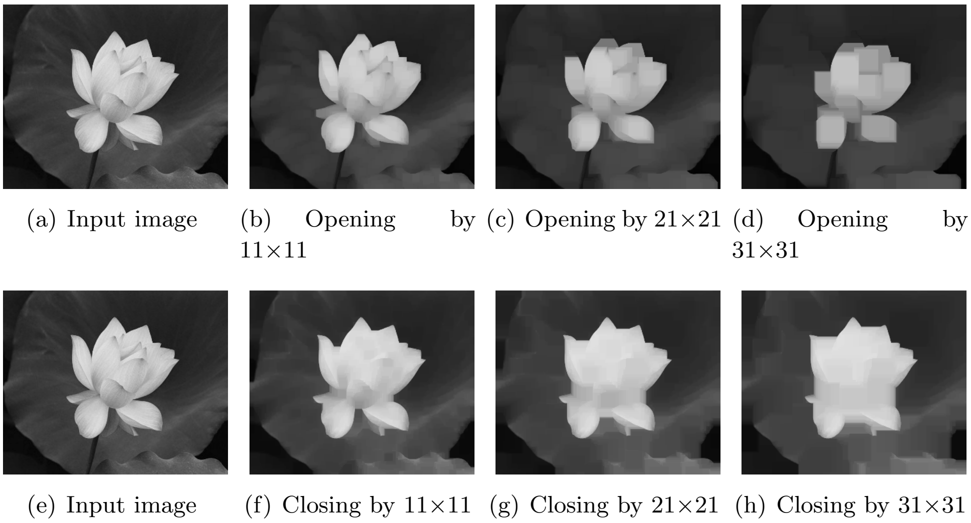
$$\varphi_B(f) = \varepsilon_{\widehat{B}}[\delta_B(f)]. \tag{2-21}$$

The opening and closing are dual operations according to the complementation $\complement$, such as

$$\varphi_B(f) = \complement\gamma_{\widehat{B}}\complement(f) \tag{2-22}$$

2.0.3 Alternating Sequential Filters

From opening and closing, one forms alternating filters obtained as $\gamma\varphi$, $\varphi\gamma$, $\gamma\varphi\gamma$ and $\varphi\gamma\varphi$. The number of combinations obtained from two filters is rather limited. Other filters can be obtained by combining two *families* of filters. This leads to morphological Alternating Sequential Filters (ASF), originally proposed by [Sternberg 1986], and studied in [Serra 1988], Chapter 10. In general, it is a family of operators parameterized by some $\lambda \in \mathbb{Z}^+$, obtained by alternating concatenation


(a) Input image (b) Opening by 11×11 (c) Opening by 21×21 (d) Opening by 31×31

(e) Input image (f) Closing by 11×11 (g) Closing by 21×21 (h) Closing by 31×31

Figure 2.5: Example of images processed by opening and closing with various SEs. (a) (e) input image f, (b) opening by SE 11×11, (c) opening by SE 21×21, (d) opening by SE 31×31, (f) closing by SE 11×11, (g) closing by SE 21×21, (h) closing by SE 31×31.

of two families of increasing and decreasing filters $\{\xi_i\}$ and $\{\psi_i\}$, respectively, such that $\psi_n \leq \ldots \leq \psi_1 \leq \xi_1 \leq \ldots \leq \xi_n$.

The most known ASF are those based on openings and closings, obtained by taking $\psi = \gamma$ and $\xi = \varphi$:

$$ASF^{\lambda} = \gamma^{\lambda}\varphi^{\lambda}\ldots\gamma^{1}\varphi^{1} \tag{2-23}$$

starting with a closing, and

$$ASF^{\lambda} = \varphi^{\lambda}\gamma^{\lambda}\ldots\varphi^{1}\gamma^{1} \tag{2-24}$$

starting with an opening, the example of which for different orders is shown in Fig. 2.6.

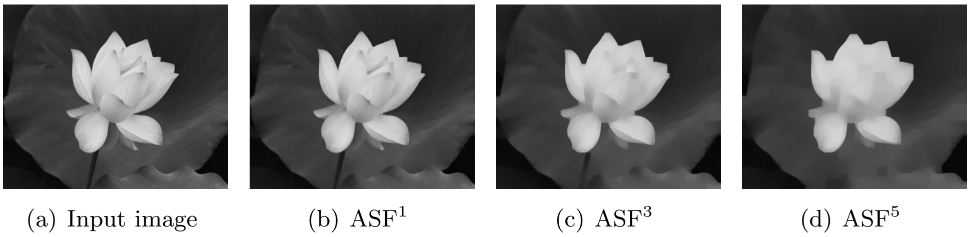

(a) Input image (b) ASF^1 (c) ASF^3 (d) ASF^5

Figure 2.6: Example of images processed by ASF of various orders with a rectangular SE. (a) input image f, (b) ASF^1, (c) ASF^3, (d) ASF^5.

2.0.4 Granulometry and Pattern Spectrum

Another application of opening and closing is called granulometry, or pattern spectrum. The concept of granulometry was introduced by [Matheron 1975] in a study of porous materials. Let $\Psi = (\psi_\lambda)_{\lambda \geq 0}$ be a family of image transformations depending on a parameter λ. This family constitutes granulometry if and only if it forms a decreasing family of openings, that is

$$\forall \lambda \geq 0, \qquad \psi_\lambda \text{ is an opening} \tag{2-25}$$

$$\forall \lambda \geq 0, \mu \geq 0, \qquad \lambda \geq \mu \Rightarrow \psi_\lambda \leq \psi_\mu \tag{2-26}$$

The above definition does not require the opening ψ_λ to be a morphological opening. Algebraic granulometries, e.g., granulometry by area, based on algebraic openings are also valid, see [Serra 1988]. However, we will focus on a morphological granulometry hereafter.

The family $\Gamma = (\gamma_\lambda)_{\lambda \geq 0}$ of openings by homothetics $\lambda B = \{\lambda b | b \in B\}, \lambda \geq 0$, of B is a granulometry if and only if B is convex. In more practical way it means that provided a convex primary grain B, the family of openings with all the scales of B is a granulometry.

The granulometric analysis of a set f is often presented as a *sieving process*, f is sieved through a set of sieves with increasing mesh size. Each opening removes more than the previous one. In order to quantify the rate of sieving f, a measure $m(f)$ is used. In the most cases, $m(f)$ measures the sum of all pixels remaining in f. The measure constitutes a granulometric curve of f with respect to granulometry $\Gamma = (\gamma_\lambda)_{\lambda \geq 0}$ such as

$$G_\Gamma(f) = m(\gamma_\lambda(f)) - m(\gamma_{\lambda-1}(f)). \tag{2-27}$$

The pattern spectrum is an operator very similar to the morphological granulometric curve, but it was defined in different way in [Maragos 1989]. Let $S_{\lambda B}$: $\mathbb{R}^2 \rightarrow \mathbb{R}$ be a single value of the pattern spectrum, parameterized by a SE $B \subset \mathbb{R}^2$ and its size λ, defined as

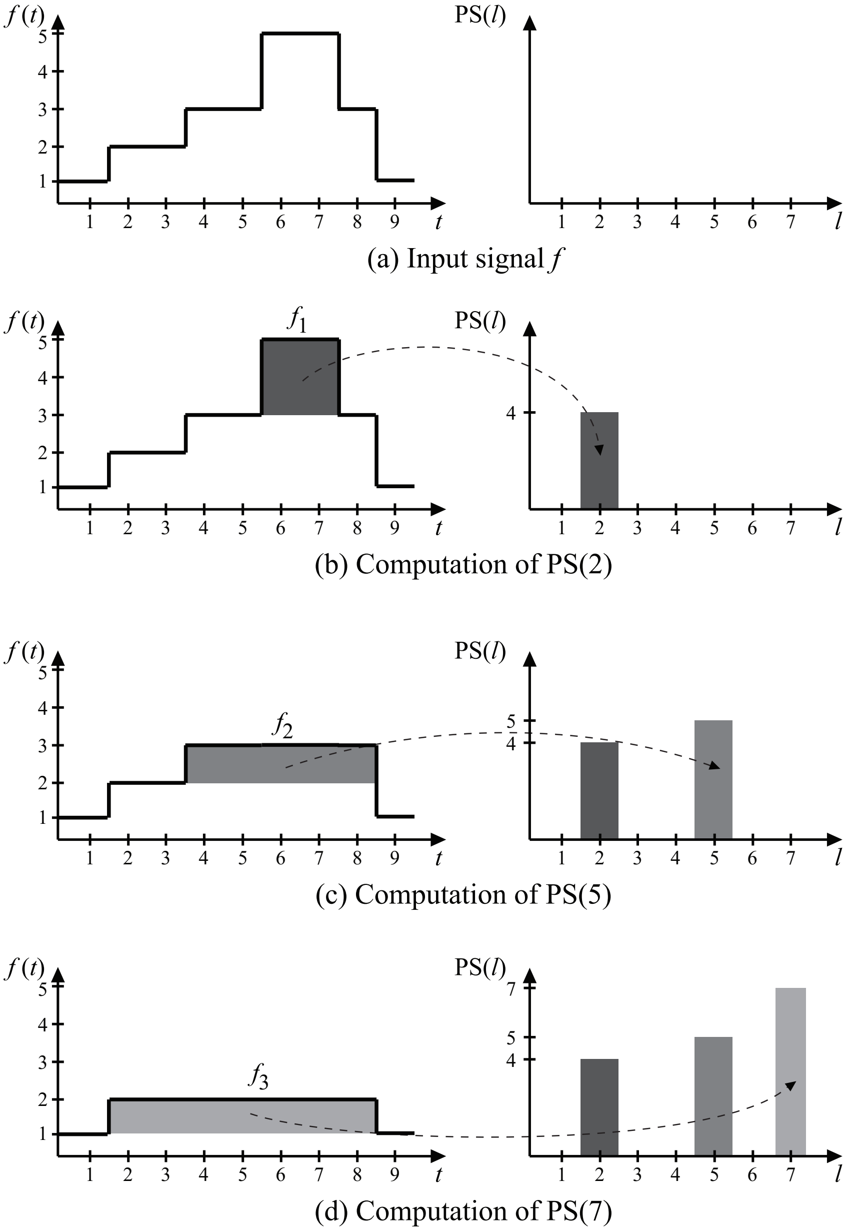
$$S_{\lambda B}(f) = -\frac{d}{d\lambda}\|\gamma_{\lambda B} f\|; \quad f : \mathbb{R}^2 \rightarrow \mathbb{R}. \tag{2-28}$$

Since we are interested in discrete images with bounded support $X \subset \mathbb{Z}^2, X = [1, M] \times [1, N]$, the discrete value of $S_{\lambda B}$ is transformed to

$$S_{\lambda B}(f) = \sum_X \left(\gamma_{\lambda B} f - \gamma_{(\lambda+1)B} f\right); \quad f : D \rightarrow \mathbb{R}, \tag{2-29}$$

considering the pattern spectrum step $d\lambda = 1$.

In the following definitions, we consider γ_l^α be the opening by a line SE L_l^α. This SE has a shape of a discrete line of length l rotated by angle α from the positive x-axis counterclockwise (γ_l^α is also called *linear opening*) and is commonly used for extracting information about orientation of objects. Using this SE we obtain



(a) Input signal f

(b) Computation of PS(2)

(c) Computation of PS(5)

(d) Computation of PS(7)

Figure 2.7: Example of 1-D signal f and creation of the pattern spectrum $PS(l)$.

the equation for the oriented pattern spectrum PS: $\mathbb{Z}^2 \rightarrow \mathbb{R}$ in (2-30). Such a pattern spectrum PS of an anisotropic texture is the size distribution expectancy of a 1-D signal obtained by intersection with a randomly drawn straight line. The expectancy is approximated by the frequency count.

$$[PS(\alpha, l)]\,(f) = \sum_{X} \left(\gamma_l^{\alpha} f - \gamma_{l+1}^{\alpha} f\right). \tag{2-30}$$

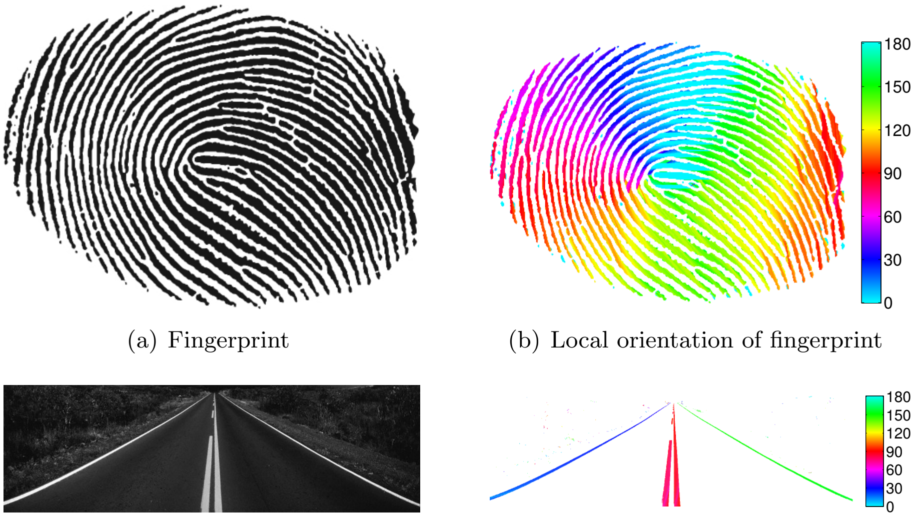
Figure 2.7 shows an intuitive representation of the pattern spectrum $PS(l)$ on 1-D signal f, so only the length of the SE l is variable. We start by computing the first element $PS(1) = \sum(\gamma_1 f - \gamma_2 f)$. Since neither γ_1 nor γ_2 changes the signal f, see Fig. 2.7 (a), $PS(1)$ remains empty. Then we go one step further, we compute $\gamma_3 f$ and subtract that from already computed $\gamma_2 f$. As we can see in Fig. 2.7 (b), γ_3 cuts off the signal cord f_1, the area of which is the result of the sum of the two openings in $PS(2)$. Note that the length of the eliminated cord is 2 (equal to the length of the SE $l = 2$) but PS contains its area 4. In the next two steps, i.e., for $l = \{3, 4\}$, the PS contains zero values as both openings return the same signals. For $l = 5$ in Fig. 2.7 (c), the subtraction of $\gamma_5 f$ and $\gamma_6 f$ results in the area of the 5-pixel-wide cord labeled f_2 that goes to $PS(5)$. The openings γ_7 and γ_8 reveals the 7-pixel-wide cord in Fig. 2.7 (d).

In conclusion, this operator decomposes the original signal into a set of signal cords obtained as the residue in the equation (2-30) above (cords of length l). They are represented as a discrete histogram of the sum of their area so that the area of a cord of length l contributes to the l-th bin.

The oriented linear opening γ_l^α can be used for the detection of local orientation (orientation field) ζ_l: $\mathbb{Z}^2 \rightarrow [0, 180)$ by looking for angle α that causes the greatest response of $\gamma_l^\alpha(f)$ at each point as

$$\zeta_l(f) = \arg \max_{\alpha \in [0,180)} \gamma_l^\alpha(f). \tag{2-31}$$

Examples of ζ_l applications are shown in Fig. 2.8. The first application shows the local orientation of elongated papillary lines of the fingerprint, the second example determines the orientation of the road lines in the image of a road.



(a) Fingerprint

(b) Local orientation of fingerprint

(c) Road

(d) Local orientation of road

Figure 2.8: Extraction of local orientation on (a) a fingerprint and (b) a road image.

Another operator $\chi : \mathbb{Z}^2 \rightarrow \mathbb{R}$ may also take advantage of γ_l^α for image restoration by taking a pixel-wise supremum of openings by different angles in each pixel as

$$\chi_l(f) = \bigvee_{\alpha \in [0,180)} \gamma_l^{\alpha}(f) \tag{2-32}$$

The last two operators are rather application-oriented. They demonstrate the feasible applications of linear openings with arbitrary orientation.

3 State of the Art

Contents

This chapter surveys the state of the art of mathematical morphology from two different perspectives. At first, algorithmic advances of the low-level morphological operators dilation and erosion in literature are reviewed. Dilation and erosion are the fundamental and most common operators; they are utilized in almost every application that concerns mathematical morphology. Also, many complex morphological operators and methods are composed of various concatenations of these basic operations.

Later, we outline major previous contributions to implementation of the mathematical morphology on different platforms. We are interested in three platforms most suitable for complex image processing applications: general-purpose processors (GPP, CPU), graphics processing unit (GPU), and dedicated hardware (chiefly FPGA).

In the following paragraphs, we use the $\mathcal{O}()$ notation to express the asymptotic *computation complexity* (sometimes called *time complexity*, or *big O notation*) of an algorithm as proposed by [Knuth 1976]. The computation complexity stands for a number of atomic instructions that must be executed to apply a given algorithm on a single pixel in dependence on some quantity, size criterion n. Asymptotic property means that we are only interested in the order of complexity, e.g., the algorithm of $\mathcal{O}(100n^2) \equiv\mathcal{O}(n^2)$ has far lower complexity than the algorithm of $\mathcal{O}(0.01n^3) \equiv\mathcal{O}(n^3)$. Hereafter, the quantity criterion n is the size of the SE. On some special occasions, for instance when algorithm is of $\mathcal{O}(1)$ and we want do express the influence of image borders, we use the complexity against the size of the image. That is denoted by $\mathcal{O}_{\text{image}}()$ and should be comparable to $\mathcal{O}_{\text{image}}(MN)$ for $\mathcal{O}(1)$. For further reading and examples of algorithm analysis see [Leiss 2007, Knuth 1997].

Also, a few terms concerning the latency should be noted as we will use them in the thesis. The latency is a measure expressed in a number of data samples, e.g., pixels. We define the latency introduced by the dependence of the result on future data samples as *operator latency*. For example the max filter $y_i = \max(x_{i-2}, x_{i-1}, x_i, x_{i+1}, x_{i+2})$ has operator latency 2, defined by the distance between x_i and x_{i+2}. Operators with non-zero operator latency are sometimes referred to as non-causal. We define as *algorithm latency* any additional latency introduced by the algorithm, e.g., the necessity to perform a reverse scan on data, computing intermediate results, etc. Last, *computing latency* measures the impact of the implementation on computation. For instance, the polyadic max from the example above can either be executed sequentially on a sequential machine, in a pipeline, or entirely in parallel on a dedicated hardware. The *system latency*, or simple *latency*, in the usual sense is the sum of these three terms.

3.1 Advances of Basic Morphology Algorithms

The mathematical morphology itself has been studied since its first appearance to improve efficiency and enrich applicability. The development of algorithms attracted a large portion of mathematicians' attention throughout the whole time. It is worthy to recall that the efficiency of an algorithm directly affects its usability whenever the processing time is the main concern.

The following paragraphs present advances of two essential low-level morphological algorithms, dilation and opening. The most efficient dilation algorithms are based on the SE decomposition to a set of basic, more easily optimized shapes. A special attention is paid to the n-D SE decomposition into 1-D SE because the 1-D algorithms obtain the most significant gain in the overall performance.

3.1.1 1-D Dilation Algorithms

The simplest method to compute dilation is the exhaustive search for maximum in the scope of SE B according to definition (2-6). This naive solution tends to need a large number of comparisons, which are on most platforms diadic (with two operands). The number of comparisons is considered as a metric of algorithm complexity, so the naive algorithm has complexity $\mathcal{O}(l)$ as it has to carry out $l-1$ comparisons for an l pixel long SE. Such complexity suggests that the naive algorithm is inefficient for any large SEs. [Pecht 1985] proposed a method to decrease the complexity based on logarithmic SE decomposition, thereby achieving $\mathcal{O}(\lceil \log_2(l) \rceil)$ complexity.

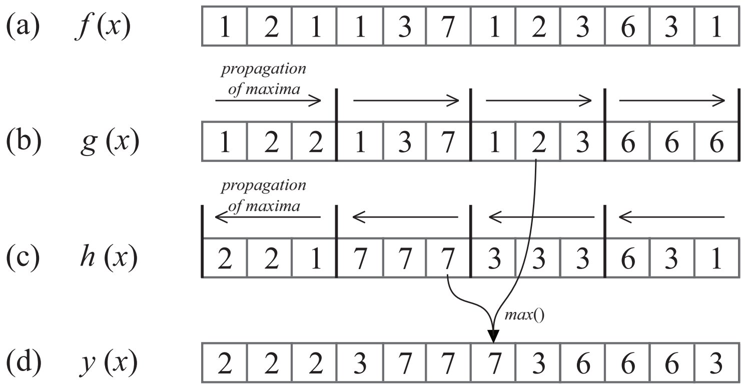
The first 1-D algorithm that reduced complexity to a constant is often referred to as HGW (it was published simultaneously in two papers: [van Herk 1992] and [Gil 1993]). The computation complexity is constant, i.e., of $\mathcal{O}(1)$, which means the upper bound of the computation time is independent of the SE size. The HGW algorithm uses two buffers $g(x)$ and $h(x)$, which are divided into segments of the SE length l. Each segment of the first buffer is filled by the forward propagation of local maxima (3-1) (within the scope of a segment), whereas the second buffer stores the reverse propagation (3-2). The result $y(x) = \delta_B f(x)$ is obtained by merging both buffers such as (3-3).

$$g(x) = \begin{cases} f(x) & \text{if } x \bmod l = 0 \\ \max(g(x-1), f(x)) & \text{otherwise} \end{cases}, x = [0..N-1] \tag{3-1}$$

$$h(x) = \begin{cases} f(x) & \text{if } (x+1) \bmod l = 0 \\ \max(h(x+1), f(x)) & \text{otherwise} \end{cases}, x = [N-1..0] \tag{3-2}$$

$$y(x) = \max(g(x + (l-1)/2), h(x - (l-1)/2)), x = [0..N-1] \tag{3-3}$$

An example of the HGW algorithm run for $l = 3, N = 12$ is illustrated in Fig 3.1. First, the original signal (a) is forward scanned to compute the forward propagation of maxima in buffer $g(x)$ (b). Notice that the maxima do not propagate beyond the segment of l pixels, see for instance the first segment. The second buffer $h(x)$ (c) is computed in the same way while backward scanning $f(x)$. Finally, the two buffers are merged into the output signal by pixel-wise maximum such as $y(x) = \max(g(x+1), h(x-1))$.



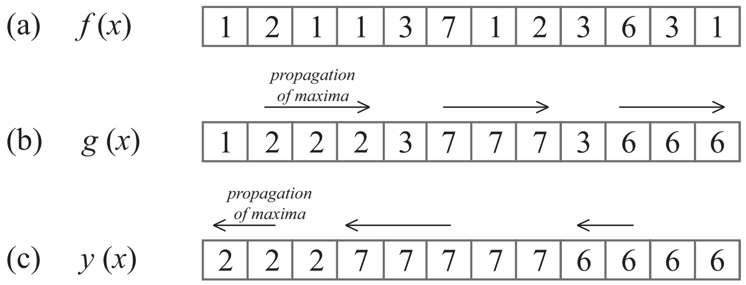
(a)	$f(x)$	1	2	1	1	3	7	1	2	3	6	3	1
(b)	$g(x)$	1	2	2	1	3	7	1	2	3	6	6	6
(c)	$h(x)$	2	2	1	7	7	7	3	3	3	6	3	1
(d)	$y(x)$	2	2	2	3	7	7	7	3	6	6	6	3

Figure 3.1: Illustration of HGW algorithm run with SE B long $l = 3$ px: (a) input signal $f(x)$, (b) forward propagation buffer $g(x)$, (c) backward propagation buffer $h(x)$, (d) output signal $y(x) = \delta_B f(x)$.

The simplified description above does not correctly handle all the possible cases (e.g., when $N \bmod l \neq 0$, border pixels), but rather presents the major drawback of this algorithm, the requirement of two data scans: forward and reverse (so-called causal and anti-causal). The unlike scans impose significant restrictions to the implementation on platforms with limited memory management (e.g., dedicated hardware) and infer high latency, especially in the vertical direction. [Gil 2002] proposed an improved version of HGW that lowered the number of comparisons per element, but at the cost of increased memory usage and implementation complexity.

[Lemonnier 1995] proposes another algorithm of $\mathcal{O}(1)$ that also identifies local extrema and propagates their values. This algorithm does not divide $f(x)$ into segments, but propagates the maxima of $f(x)$ as long as it is covered by the SE

B instead. At first step, the algorithm forward propagates all local maxima by $k = (l-1)/2$ pixels storing results in a buffer $g(x)$, see Fig 3.2 (b) for example of $l = 5, k = 2, N = 12$. We notice that each local maximum is propagated only 2 pixels rightwards. The second step backward propagates maxima of the buffer $g(x)$ for the maximal distance of k pixels leftwards, see Fig 3.2 (c). Again, the limiting forward and reverse scans are needed for every non-causal SEs. Although this algorithm needs only 2 max operations per pixel, its implementation results in a large number of if statements to properly treat all boundary conditions.



(a) $f(x)$ 1 2 1 1 3 7 1 2 3 6 3 1

(b) $g(x)$ 1 2 2 2 3 7 7 7 3 6 6 6

(c) $y(x)$ 2 2 2 7 7 7 7 7 6 6 6 6

Figure 3.2: Illustration of Lemonnier algorithm run with SE B long $l = 5$ px: (a) input signal $f(x)$, (b) forward propagation of maxima in $g(x)$, (c) output signal $y(x) = \delta_B f(x)$.

[Lemire 2006] proposes a fast stream-processing algorithm $\mathcal{O}(1)$ for causal line SEs. It replaces the line buffers of previous algorithms by a more specific memory structure—double-ended FIFO (queue). This algorithm uses two queues of length W in order to store the pixels that form locally monotonous signal (i.e., monotonously increasing and decreasing). Although it produces both erosion and dilation simultaneously, has lower memory requirements and zero latency, it works with causal SEs only. This downside was solved later in [Dokládal 2011] who proposed another queue-based algorithm (see Section 4.1 for further description of his algorithm). The advantages of these queue-based algorithms are low memory requirements, zero latency, and strictly sequential access to data.

Table 3.1 sums up the most important properties of the 1-D dilation algorithms mentioned above.

3.1.2 2-D Dilation Algorithms

As mentioned before, 2-D dilation can be obtained by composition of 1-D dilation. However, this often used technique covers only a limited family of shapes, such as rectangles, diamonds. In the following we will present the overview of algorithms that allow us to obtain more complex 2-D SEs.

[Soille 1996] propose an approach to approximate circles and polygons by using SE decomposition into a set of line SEs rotated by different angles. The complete dilation by a polygon requires several iterations over the image. Each line SE is computed by the fast 1-D HGW algorithm oriented by the desired angle. The

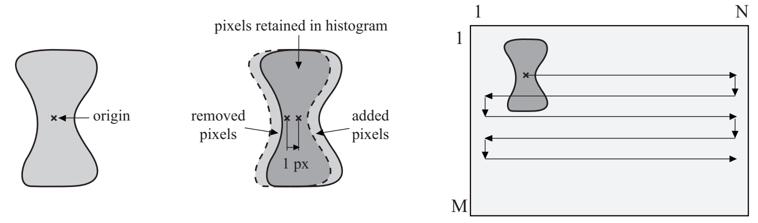
Table 3.1: Comparison of fast 1-D dilation algorithms.

Algorithm	SE type	Comparisons per pixel	Algorithm latency	Data memory	Working memory
Naive 1-D	User	$l - 1$	0	N	0
HGW	Sym	$3 - 4/l$	1	N	$2l$
Lemire	Causal	3	0	0	$2l$
Lemmonier	Sym	NC ($\mathcal{O}(1)$)	N	N	N
Van Droogenbroeck Buckley	Sym	NC ($\mathcal{O}(1)$)	0	N	$N + G$
Dokládal	User	3	0	0	$2l$

Sym = symmetric SE; User = User-defined SE; l = length of a 1-D SE; N = line size; G = number of gray levels; NC = not communicated.

orientation of the SE is achieved through image partition into discrete lines (parallel, with no overlap), along which the HGW operates. The main drawback of such an image partition is that the result SE is translation variant; the shape of the SE varies along the discrete line. The translation variance, which makes that the adjunction is not verified, may introduce undesired artifacts to many application, such as filters.

In [Van Droogenbroeck 1996] the authors proposed an algorithm for arbitrary-shaped 2-D SEs that takes advantage of a histogram to compute the dilation of pixels covered by the SE scanning the image. As the SE slides over the image by 1 px long translation, the histogram is not computed all over again from scratch, but only updated instead. The update of the histogram consists of removing pixels that are no longer covered by the SE (see the left-hand side of the SE in Fig. 3.3 (b)), and adding new pixels that become covered at the current position (see the right-hand side of the SE in Fig. 3.3 (b)). The SE does not slide over an image in the common scan order, it uses a horizontal zigzag pattern instead, see Fig. 3.3 (c). Typical complexity (for a square SE) of this algorithm is $\mathcal{O}(H \log_2(G))$, but this algorithm suffers from usage of the histogram (which does not allow for high-precision numbers) and non-causal zigzag image scan.

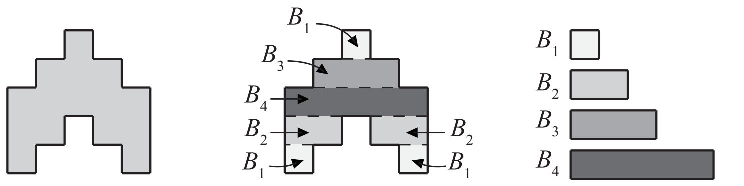


(a) SE (b) SE translation (c) Order of image scan

Figure 3.3: Histogram-based algorithm for arbitrarily shaped SEs: (a) example of SE, (b) SE translation by 1 px, (c) horizontal zigzag scanning pattern.

Recently, [Urbach 2008] propose an algorithm for arbitrary-shaped 2-D flat SEs based on decomposition of the SE into a set of N_c elementary line SEs called chords, see Fig. 3.4 (b). The whole set of chords ($N_c = 4$ chords in the case of Fig. 3.4) is computed for every pixel and stored in a look-up table. The result is then computed by taking a maximum from the values of all chords (stored in the look-up table) corresponding to the shape of the SE. Although the computation time is independent of the image content, a large look-up table (easily dozens of chords for each pixel) and non-optimized search of the maximum from the look-up table are the limiting factors for hardware implementation.

Table 3.2 outlines overview of the 2-D dilation algorithms described above.



(a) SE B (b) B as 7 chords (c) Used chords $B_{1\text{-}4}$

Figure 3.4: Chords decomposition algorithm: (a) example of SE, (b) SE chords decomposition, (c) set of chords B_{1-4} to be computed for each pixel.

TABLE 3.2: COMPARISON OF FAST 2-D DILATION ALGORITHMS.

Algorithm	SE type	Complexity per pixel	Algorithm latency	Data memory	Working memory
Naive 2-D	User	$\mathcal{O}(WH)$	0	MN	0
Urbach-Wilkinson	User	$\mathcal{O}(N_c + \log_2(L_{max}(C))$	MN	MN	$NH\log_2(W)$
Van Droogenbroeck-Talbot	User	$\mathcal{O}(H\log_2(G))$	0	NH	WHG
Dokládal (SE decomposition)	Rect	$\mathcal{O}(1)$	0	0	$2(W + NH)$

Rect = rectangular SE; User = User-defined SE; $N \times M$ = image size; $W \times H$ = SE size; G = number of gray levels; $L_{max}(C)$ = maximum chord length, N_c = number of chords.

3.1.3 1-D Opening Algorithms

1-D opening algorithms can be divided into three classes: (i) two-stage algorithms, (ii) direct computation, and (iii) connected component trees (CCT). The last-named approach is very complex containing several advanced techniques such as building CCT, computation of attributes, image restitution, and is mentioned just for completeness. The description of CCT algorithms can be found in [Salembier 1998] [Menotti 2007] [Wilkinson 2008] [Matas 2008].

The two-stage algorithms stem from a concatenation of erosion and dilation, such as

$$\gamma_B(f) = \delta_{\widehat{B}}[\varepsilon_B(f)]. \tag{3-4}$$

where the hat $\widehat{}$ denotes the transposition of the structuring element, equal to the set reflection $\widehat{B} = \{x \| - x \in B\}$, which may be difficult to achieve efficiently for some shapes of SEs. Also this approach demands two scans of input image. These two downsides can be overcome by the direct computation.

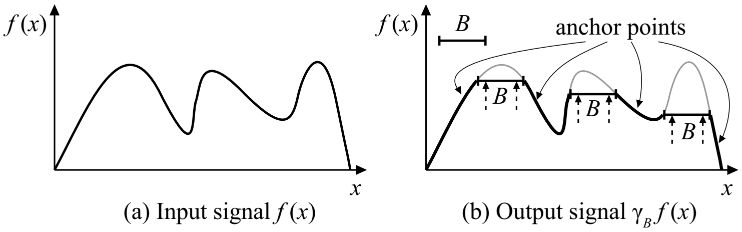


Figure 3.5: Illustration of opening algorithm using anchors. Anchors are those points of the signal that are not changed by opening: (a) input signal $f(x)$, (b) output signal $\gamma_B f(x)$.

One direct approach was introduced in [Van Droogenbroeck 2005]. The authors brought in a new notion of *anchors*, the points that are not changed by the opening operation, i.e., $f(a)$ is an anchor if $f(a) = \gamma_B f(a)$. In order to decide whether a pixel is an anchor or not, 6 different signal patterns are to be tested. All pixels between two anchors are replaced by the value of anchors. This tends to a rather complex code (however, $\mathcal{O}(1)$), large memory demands, random access to data, and due to the use of a histogram, the high-precision data are very penalizing. Therefore, this algorithm is suitable for neither the GPU nor dedicated hardware implementation. On the other hand, it is still the fastest solution for opening on general-purpose processor platforms, which cope with random memory accesses and complex code much better.

[Morard 2011] developed another direct opening algorithm with $\mathcal{O}(1)$ that decomposes the input signal into a set of flat zones (signal segments with a constant value). It uses a stack to store the partially processed flat zones, the endpoint of which has not been encountered yet. When the endpoint of the topmost stacked flat zone is reached, which indicates this flat zone is complete now, all stacked flat zones are examined whether they are complete as well. If any particular complete flat zone is shorter than the SE, it is dropped from the stack. If the flat zone is longer than the SE, this flat zone and all the others currently in the stack form the output signal of opening and are written to the output, then erased from the stack. Writing the output in such a manner is however quite irregular, the output data are written in bursts at a random, data-dependent time.

Recently, [Bartovský 2011a] proposed another direct method for 1-D opening that overcomes the Morard's drawback of the irregular output data access. The algorithm is called *streaming peak elimination*. It uses a queue to store pixels in the scope of the SE and erases all signal peaks detected in the queue. See Section 4.4 for a thorough algorithm description and Section 4.4.4 for a performance comparison. Table 3.3 contains overview of the described 1-D opening algorithms.

Table 3.3: Comparison of fast 1-D opening algorithms.

Algorithm	Comparisons per pixel	Algorithm latency	Data memory	Working memory
Naive 1-D	$2(l-1)$	N	N	0
Morard	NC ($\mathcal{O}(1)$)	N	0	$2N$
Van Droogenbroeck Buckley	NC ($\mathcal{O}(1)$)	0	N	$N+G$
streaming peak elimination	6	0	0	$2l$

l = length of a 1-D SE; N = line size; G = number of gray levels; NC = not communicated.

3.1.4 2-D Opening Algorithms

2-D opening is not separable into two orthogonal 1-D openings as dilation is. Hence, one cannot directly combine two orthogonal 1-D openings to obtain 2-D opening. However following (3-4), one can form two-stage, 2-D openings by concatenating 2-D erosion and 2-D dilation, which are separable into 1-D scopes. To our knowledge, there is no efficient direct algorithm for 2-D opening with comparable gain of efficiency over the direct computation as there is for dilation. The CCT can be theoretically used for morphological 2-D opening, but it is more useful for other kinds of opening, such as attribute opening, area opening, etc.

3.1.5 Choice of Algorithm for Hardware Implementation

Effectively all aforementioned algorithms concentrate on the optimization of dilation/erosion by means of reducing the number of operations without taking into account either the entire application or limited memory. This is a consequence inferred from the computer architecture, which offers only limited parallelism through the parallel instructions SSE or multi-threading but has the huge and cached data and instruction memory. Such a platform can well cope with a complex program consisting a several very heterogenous, different-purpose parts of code, and with very large memory demands.

For the purpose of dedicated hardware, other considerations should be taken into account. The absolute number of comparisons is no longer the best indicator of performance provided that some of them can be evaluated in parallel. The hardware platforms usually possess a very limited amount of a high-speed memory, which is suitable to be used as the working memory. Therefore, the memory requirements of an algorithm is of much greater importance than in the case of computers. The limited working memory, which is often smaller than an image itself, infers further constraints on the data access and makes the algorithms with reverse scan processing hard or even impossible to implement. Needless to say, the algorithms with strictly sequential access to data are preferred.

As a conclusion, the queue-based algorithms supporting stream processing and requiring a low amount of working memory seem to be a reasonable choice for the hardware platforms. From the two aforementioned dilation algorithms we have chosen the *Dokládal* published in [Dokládal 2011] over the one by [Lemire 2006] for its support of non-causal SEs and simpler code (note that [Lemire 2006] computes both dilation and erosion). The [Bartovský 2011a] algorithm seems to be the best choice for opening for the same reasons.

3.2 Advances in Morphology Implementation

In this section, we present an overview of algorithm implementations on different platforms. Besides general-purpose processors, this section aims also at less common platforms in general-purpose image processing: graphics processing units and dedicated hardware. Both platforms can obtain a significant speed-up against computers for certain applications, and can be employed in applications where computers are not suitable, e.g., embedded systems.

3.2.1 General-purpose Processors

A very traditional platform for processing of any kind is a general-purpose processor (GPP, so-called CPU, e.g., personal computer), used by a numerous community of programmers, and does not need any thorough introduction. Since the mathematical morphology has been recently adopted in a several image processing standards, we can find morphological operators in a plenty of image processing libraries or commercial tools, the purpose of which scales from proof-of-concept to professional performance. For instance, see [MATLAB 2012] image processing toolbox, [Octave 2012], [OpenCV 2012] library (leveraging Intel Processing Primitives IPP), [Mamba 2012], or proprietary [Morph-M 2012]. Also, each previously cited algorithm was developed on a GPP first, and therefore, its efficient implementation is included in some small, personal library.

Besides GPP, the mathematical morphology can be a subject of the implementation on signal processors. [Brambor 2006] explores capabilities of the SIMD instruction set to speed up morphological algorithms using Haskel functional language. He proposed a method that divides an image into many macro blocks and executes a particular operation on a several macro blocks simultaneously. [Clienti 2009] took advantage of the VLIW processor achieving thus a reasonable level of instruction parallelism. Such a processor executes different stages of a few threads at the same time. The same author also developed [Fulguro 2010]. It is a library for image processing using SIMD optimizations and smart threading to cope with the real-time constraints.

3.2.2 Graphics Processing Units

Another platform that is supposed to be suitable for mathematical morphology is a GPU. Surprisingly, there are only few implementations targeted to this massively parallel platform. Let us recall that a GPU composes of several light-weighted processors, each of which executes a number of threads of the program at a time. Therefore, there may be thousands of threads running that implies two important aspects to consider. First, the program to execute in the GPU has to be decomposable into thousands of threads (most favorably independent threads) in order to keep occupied as many processors as possible. The second aspect is the mutual synchronization of threads. Although the global memory is very fast with wide data bus, the maximal bandwidth (in orders of 100 GB/s) is achieved only when each block of adjacent threads (32 threads in the case of nVidia [nVidia 2012]) accesses consecutive addresses, so the memory can handle the whole block of threads during a single so-called coalesced memory read/write cycle. As a consequence, the synchronization of threads within such a block is worthwhile of attention to deliver the highest performance. There are several development tools that help a programmer to cope with these issues and to utilize the whole GPU parallelism in a relatively easy-to-use way.

The widely adopted framework for writing programs that execute on different heterogeneous platforms (including many-core CPUs, GPUs, FPGAs) is [OpenCL 2012]. It is derived from C99 language, but it is further extended with vector types, operations, synchronization, and API functions. OpenCL is integrated in drivers and development tools of both major GPU vendors ATI and NVidia. In addition, both vendors support GPUs with their own development toolkits by means of low-level drivers and high-level API that allows prospective designers to program GPUs right away (cf [CUDA 2012] by NVidia). Finally, one can cite OpenGL or OpenVidia libraries that serve for the purpose of producing 2-D and 3-D graphics instead of graphics processing.

Considering the implementation of the basic morphological algorithms, [Clienti 2009] evaluated several trade-offs of a naive algorithm in his thesis. However, his study was focused rather on the comparison of the naive algorithm among different platforms than on searching the algorithm with the best performance. As a matter of fact, even basic benchmarks reveal that many of the publicly available GPU implementations ([OpenCV 2012]) use a naive algorithm. Despite the naive solution tends to long execution time for large SEs due to the quadratic complexity $\mathcal{O}(n^2)$, the regular memory access and simplicity aspects seem to be of higher importance.

As one of non-naive implementations we can cite [Karas 2010]. The authors compute the morphological sequential reconstruction (using HMAX transform for a marker creation) on 3-D images from confocal microscopy with significant speedup. [Karas 2012b] implements in GPU two algorithms for arbitrary-oriented linear opening by [Bartovský 2011a] (described in Section 4.4) and [Morard 2011]. It proves that the state-of-the-art algorithms based on the use of a special memory structures (queue, stack) can be efficiently implemented on the GPU and outperform computers by a speed-up of approximately 35×.

The GPU performance results could satisfy some applications in terms of the absolute throughput, but the energy consumption and space occupation remain high against optimized hardware/embedded architectures. The fixed GPU architecture also put some restrictions on temporal parallelism one can exploit. For instance, consider a concatenation of operators such as $\psi\xi$, where output data of ψ is the input data to subsequent ξ, so we say ξ is dependent on ψ. In order to keep both operators running at the same time, a complex system of synchronization (e.g., semaphores) is to be established. On the other hand if the operators are applied one after the other, the latency will linearly increase with the length of the concatenation and may be unacceptable for time-critical applications.

3.2.3 Dedicated Hardware

The development of dedicated hardware has been always driven by different constraints than GPP. While developing an application for GPP, a programmer's main interest that leads to a fast solution is to keep the number of instructions as small as possible. In the low-level mathematical morphology, the number of instructions almost directly depends on the number of pixel comparisons. Hence, the aforementioned algorithms in Section 3.1 compete in decreasing the amount of comparisons compromising regularity of memory accesses.

On the other hand, a hardware developer can afford a greater amount of comparisons provided they can be applied by dedicated resources in parallel. The memory access patterns are, however, of much bigger concern. Computers are equipped with tens of gigabytes of memory whereas the dedicated hardware has less on-chip memory in orders of magnitude. The on-chip memory is often in range of 1–10 Mbits, which may not fit even one image to process. Therefore, the algorithms working with regular (sequential) dataflows that eliminate the image storing are preferred in dedicated hardware.

So, we assume that the input data to process form a stream (usually horizontal scan ordered) that flows through a processing block. The architectures that processes the data in a stream are often called *dataflow architectures* in literature, which can be classified into three groups: (i) 3×3 neighborhood processors, (ii) partial-result reuse (PRR), and (iii) implementing efficient 1-D algorithm with $\mathcal{O}(1)$.

3.2.3.1 3×3 neighborhood processor

This approach aims at computing morphological operations on a programmable 3×3 SE using some naive computation method. As the SE is small and the max() on the nine values can be parallelized, the naive computation does not deteriorate performance. The neighborhood processors have usually 2 stages: SE extraction, and morphological computation.

The SE extraction stage determines which pixels of the dataflow are currently covered by the SE, and therefore should be taken into account by the computation stage. According to the fact that SE consists of more than one pixel, it must be able to preserve all the pixels of data flow that might be needed by the SE in future samples. The most popular concept for the SE extraction is based on delay lines. The delay-line concept consists of one FIFO memory of length $N-3$ for each line of SE except the first one, and 3 registers for each line of the SE, see Fig. 3.6. Notice that the SE virtually slides over the image as the input data flows through the block. The following architectures use delay lines for SE extraction, and only differ in mechanisms that carry out the computation of morphological operation, i.e., (2-4) or (2-9).

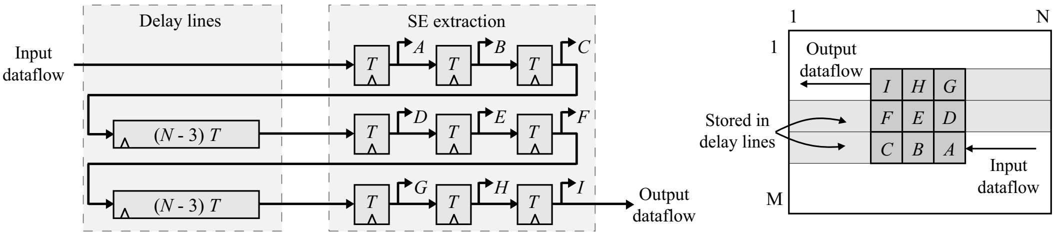


Figure 3.6: Delay-line architecture for 3×3 SE extraction.

One of the first delay-line architectures was the texture analyzer [Klein 1972]. It was optimized for linear and rectangular SE by decomposition into line segments. In [Klein 1989] the authors devised PIMM1 (Processeur Intégré de Morphologie Mathématique) ASIC that contains one numerical unit for gray-scale images and 8 binary units. The scale of supported operations was quite large including dilation/erosion, opening/closing, top hat, distance and so forth. However, the mechanism of computation was not communicated.

More recently, [Velten 2004] proposes another delay-line based architecture for binary images supporting arbitrarily shaped 3×3 SEs. The computation of dilation $\max(\{A-I\})$ is realized by OR gates (topology was not communicated, probably a tree of diadic OR gates similar to one in Fig.3.8) achieving good performance, which was further improved by spatial parallelism. In the parallel mode, the OR gates are p-times duplicated, and p succeeding results are computed over the $(2+p)\times3$ extracted neighborhood at the same time.

[Clienti 2008a] proposes a highly parallel morphological System-on-Chip. It is a set of neighborhood processors PoC optimized for arbitrarily shaped 3×3 SE interconnected in a partially configurable pipeline displayed in Fig. 3.7. Each stage of the pipeline contains 2 processors that can process 2 parallel image streams and an ALU. The reconfiguration allows all the processors to be connected in one chain in order to employ all processors when only one image stream is used.

The PoC processor takes advantage of the delay-line SE extraction mentioned above. The shape of the elementary SE is given by masking the undesired pixels by

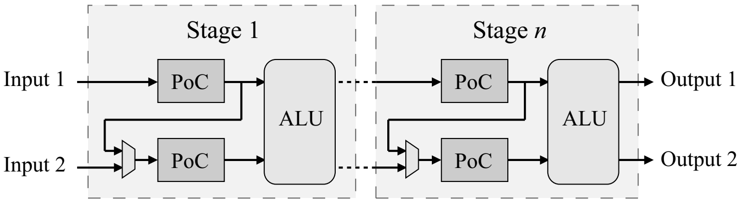


Figure 3.7: Top-level view of the Clienti pipeline architecture.

recessive values ($\wedge f$ for dilation, $\vee f$ for erosion). For instance to obtain the dilation by horizontal 3×1 SE, one masks the pixels $\{A,B,C,G,H,I\}$ by 0. The dilation itself is carried out by a tree of diadic max() operators, see Fig. 3.8. The max-tree is pipelined. It first computes column-wise partial results, and then merges these results into the output value of the desired dilation. The column-wise scheme was chosen for the sake of parallel computation that computes a several consecutive pixels. The SEs of these pixels overlap each other by 1 or 2 columns, so column-wise partial results can be reused and participate in computation of multiple pixels at the same time.

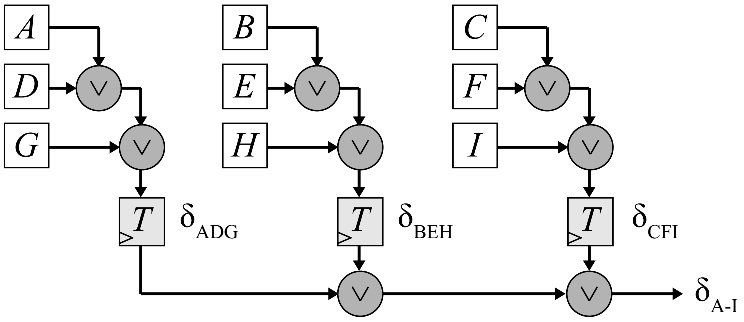


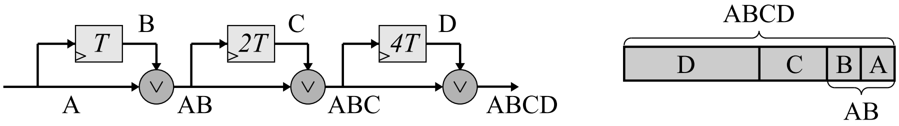
Figure 3.8: Tree of diadic max operators for dilation on the 3×3 SE. $A - I$ denote extracted pixels of the SE.

3.2.3.2 Partial-result reuse

All the previous architectures use a naive method to compute the morphological operations. This approach may be reasonable for small SEs, but it becomes very inefficient for large SEs because of an excessive number of comparisons. On the contrary, the PRR approach (name proposed in [Chien 2005]) does not strictly separate the SE extraction stage and the computation stage from each other, but mixes them. As the name indicates, a partial result of a morphological operation by some neighborhood B_1 in an early stage is delayed by delay lines in order to be reused later in computation by some other neighborhood B_2 obtaining other, usually larger, B_3 decreasing a necessary number of comparisons.

One of the first PRR architectures for 1-D dilation was proposed in [Pitas 1989] and improved in [Coltuc 1997]. The principle is based on an exponential growth of the intermediate neighborhoods in the partial-result reuse scheme. The better understanding can be gained from an example for SE long $l = 8$ px shown in Fig. 3.9 (T stands for one period of the clock, A through D are different neighborhoods). From the left, the input pixel A is compared with 1-cycle-delayed input pixel, i.e., the previous pixel B, resulting in 2-pixel SE AB. This SE is compared with $2T$-delayed version of AB labeled C resulting in 4-pixel SE ABC. This partial result is compared with $4T$-delayed D giving us the result 8-pixel $ABCD$. To get better intuition of how the SE composition works, the delaying of partial results by some SE can be understood as translation of the SE, and the max comparison as a union (merging) of the translated and the original SE.

The advantage of this architecture is a reduced number of comparisons from general $l - 1$ to $\lceil \log_2(l) \rceil$. However, the method, so-called logarithmic SE decomposition, restricts a family of possible SE shapes to rectangles.



(a) Architecture (b) SE composition

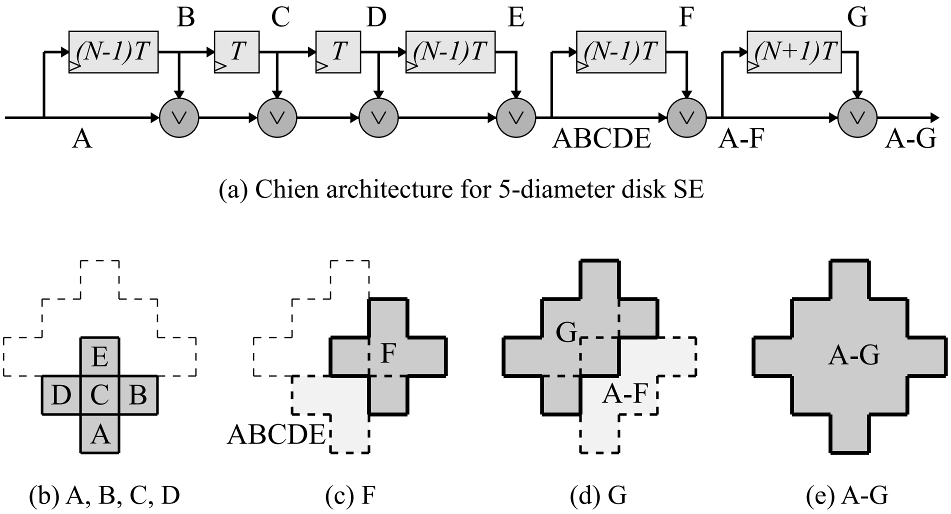
Figure 3.9: Pitas PRR architecture using the logarithmic SE decomposition. $A - D$ denote neighborhoods, T denotes period of the clocks.

The family of SE shapes has been enriched by [Chien 2005]. The authors presented more general concept of PRR that does not stick to the exponentially increasing partial neighborhoods but builds the desired SE by a set of distinct partial neighborhoods computed by a dedicated algorithm. As a result, it supports arbitrary 8-convex polygon at the cost of some additional comparisons. In [Chien 2005] the PRR method was implemented as an ASIC chip supporting 5-diameter disk SE shown in Fig. 3.10 (a).

The desired disk SE is composed as follows. From the left, the input pixel A is delayed by 4 different time intervals to obtain 4 different-located SEs: $(N - 1)T$ for B, NT for C, $(N + 1)T$ for D, and $2NT$ for E. These five singleton SEs are merged to obtain the cross SE $ABCDE$, see Fig. 3.10 (b). The $ABCDE$ is then delayed by $(N - 1)T$ (F), merged with the former one into $A - F$ SE, which is also delayed by $(N + 1)T$ (G) to obtain the result $A - G$ SE, see Fig. 3.10 (d–e).

Although the proposed ASIC chip achieves a high performance (thanks to a high frequency), it has a few downsides limiting its applicability in vision systems. First, the shape of the SE is fixed to 5-diameter disk. Second, the supported image width N is also fixed to 90 px, so any larger image must be divided into 90-pixel vertical stripes, which are processed either sequentially, which infers random access to the image, or in parallel by multiple chips.

A similar approach has been published by [Déforges 2010]. Based on the [Normand 2003] SE decomposition (a SE is decomposed into a number of causal 2-pixel SEs, which are applied in sequence or in parallel, see Section 2.0.1.1) and



(a) Chien architecture for 5-diameter disk SE

(b) A, B, C, D (c) F (d) G (e) A-G

Figure 3.10: Illustration of Chien's approach. (a) ASIC chip architecture for 5-diameter disk SE, (b–e) successive composition of the SE.

combined with a stream implementation, the authors propose a methodology for pipeline architecture design supporting arbitrary convex SEs in only one scan of the input image. It takes advantage of two elementary blocks: programmable delay (either one from $\{T, (N-1)T, NT, (N+1)T\}$, according to four 2-pixel SEs of the decomposition), and max operator. Let us recall that the Normand decomposition employs also a union of SEs, so unlike the Chien architecture that has only one pipeline, the Déforges's uses a couple of pipeline branches in parallel and merges them together to get the desired SE shape.

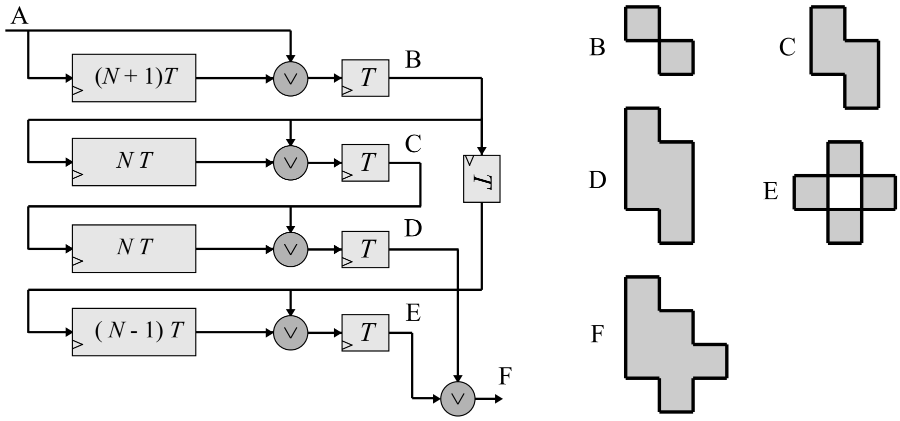
The Déforges method is illustrated on an example of an 8-convex polygon SE in Fig. 3.11. At the beginning, the 2-pixel B SE is computed in the way described above. Then the pipeline is branched. The first branch vertically elongates B through C up to D, whereas the second branch diagonally extends B to E. Finally, both branches are merged by the union operation forming the result F polygonal SE.

This approach presents a method for design of morphology hardware implementation with 8-convex SEs. The principal limitation comes from a limited programmability of the pipe, and therefore, of the SE shape. So, the synthesized architecture is designated just to one SE.

3.2.3.3 $\mathcal{O}(1)$ algorithm implementation

The efficient algorithm implementation field has always been a little bit omitted by the mathematical morphology community in favor of the previous two options. In literature there are only two proposals, moreover published together in [Clienti 2008b].

The first one directly implements the 1-D Lemonnier algorithm (see description



(a) Déforges architecture for convex SE

(b) SE composition

Figure 3.11: Example of Déforges's approach. (a) architecture for a given polygon SE, (b) SE composition.

above). According to the algorithm, the architecture consists of two propagation units, one for forward and one for backward direction, see Fig. 3.12. And here comes the main bottleneck of this solution, transforming the forward scan ordered output of the first propagation unit h into the backward scan ordered input of the second unit h'. This problem is addressed by a ping-pong pair of line buffers with unlike reading and writing orders. The term ping-pong means that when input h is being written to one buffer, the output h' is being read from the other buffer. Reversing the data scan order represents the main disadvantage and infers the following unpleasant properties: large memory requirements $2N$ (recall $N \times M$ image size), large delay of N pixels, and backward output scan order (reversing it back needs another pair of line buffers).

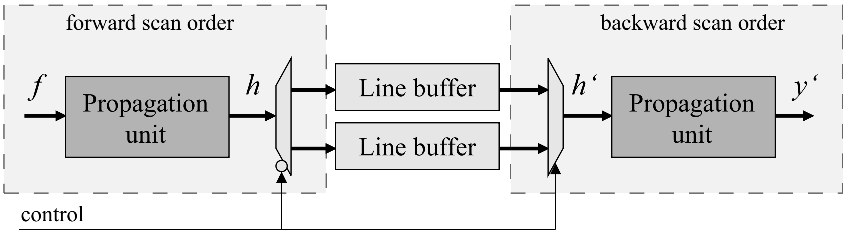


Figure 3.12: Clienti's implementation of Lemmonier algorithm.

The second attempt to implement an efficient algorithm uses the HGW algorithm described above. Since the algorithm also involves backward maxima propagation, the architecture needs some reverse units, similar to the ping-pong buffers in Lemmonier implementation. However, there is no need to reverse the whole line. As the line is divided into independent segments (of length l equal to the SE size) for the maxima propagation in g and h, it is sufficient to reverse the data order within each segment only. This segment reversing unit needs ping-pong buffers of length l instead of N, which significantly reduces memory requirements and latency compared to the original HGW algorithm or the implementation of Lemonnier.

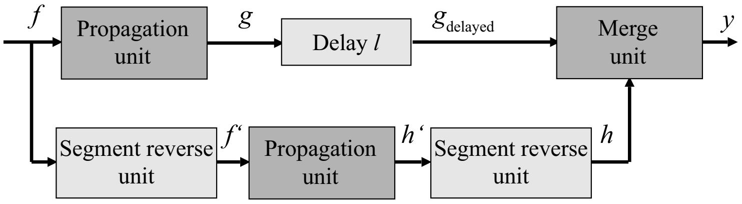


Figure 3.13: Clienti's implementation of HGW algorithm.

The processing of input signal f takes place in two branches. The first one computes the forward propagation g, the second one computes the backward propagation h using a pair of segment reverse blocks. The second branch is too much delayed, so an extra delay unit has to be used in the forward branch to have a correct delay for the merge unit. The total memory requirements of such architecture are reduced to $5l$ pixels and latency to $2l$ pixels. However, these values are still very high considering the 1-D SE. The only upside is that the hardware complexity excluding the memory is independent of the SE size. It means that the two propagation units and the merge unit combined contains only 3 comparators regardless the SE size.

Another major drawback of the two implementations above is incapability of supporting vertical, or 2-D, SE along with horizontal scan data reading. They allow for vertical SE only with vertical scan order, which is insufficient for almost any vision application. The transformation between horizontal and vertical scan orders is very expensive in terms of the time and resources, and it is not possible to demand such transformation for a low-level operation.

3.2.3.4 Miscellaneous

To complete this brief state of the art, we shall also cite some less traditional approaches to morphological architectures. One of them is a systolic array. The systolic array is a matrix network arrangement of dataflow processing units. The common inconvenience of systolic arrays are the need of an intermediate storage for 2-D SE, large number of processing elements, and high response time of the system. [Diamantaras 1997] devised a 1-D systolic architecture for basic gray-scale morphology operations. The concept is scalable with respect to the SE size. The computing of compound morphological filters requires an intermediate storage. [Malamas 2000] proposes a systolic binary morphological architecture equipped with universal morphological processing elements. It supports 2-D SEs of shapes decomposable into 1-D segments. Several image lines are processed in parallel, and their results feed a simple AND/OR gate. It leads to the requirement of the random access to the input image and consequently to computation inefficiency.

[Ikenaga 2000] proposed a Content-Addressable Memory (CAM) based architecture with a large processing element array (up to hundred thousand processing elements). Although the processing speed of the architecture is very high (several microseconds for 512×512 px), the hardware cost of CAM memories and their power consumption become limiting for larger images.

Finally, [Hedberg 2009] proposed an architecture for binary morphology with a spatially variant SE. The SE is decomposed into vertical, one column wide slices, the results of which are at the end merged together by AND gate. The binary morphological erosion within a slice SE is computed via the distance from the currently processed line to the closest upward background pixel. If the distance is greater than the height of the slice, the result is foreground, otherwise background. The drawback of this solution is the support of only binary images.

3.3 Conclusions

After reading the brief state of the art above, we can claim that there are manifold algorithms for mathematical morphology supporting various shapes of SEs. Even when we constrain ourselves to 1-D algorithms, which clearly allows for SE composition into higher dimensions, we still have several efficient $\mathcal{O}(1)$ algorithms to choose between. In a search for the best candidate for hardware implementation we did not primarily follow the optimization effort in terms of the number of comparison that is more common for GPPs, but we rather paid augmented attention to hardware considerations, especially to regularity of data accesses, memory requirements, latency etc. The group of queue-based algorithms conforms the best to these premises, from which we have chosen *Dokládal* algorithm for implementation of dilation because it supports a richer family of SE shapes and its code is simpler than Lemire, and *streaming peak elimination* for opening.

We have also reviewed implementations and found out that there are many solutions for GPPs beginning from personal libraries up to powerful and proprietary software packages. On the contrary, there are few software tools using GPUs, most of which moreover focus on a small number of operations only. In the state of the art of dedicated hardware implementations we have introduced a representative selection of recent solutions, chiefly targeted to an FPGA device.

For our intention of real-time processing, the field of the dedicated hardware is the most suitable one because neither GPP nor GPU satisfies the requirements of vision applications (cf. Section 1.1.1 on page 3) such well as the dedicated hardware. Even though GPP and GPU have satisfactory computational power for a single operator even on large images, the performance of the entire application is not large enough to comply with the real-time constraint. Also the undefined, variable, and usually large latency of these two platforms is unpleasant for hardware implementations requiring strict timing. On the other hand, the dedicated hardware allows for real-time processing of high-demanding applications with fixed latency and low power consumption.

In the target hardware field we have found a few recent publications that attain a reasonable features in terms of either performance and versatility. They are either optimized for small SEs or, on the other hand, they support too wide family of SE shapes. Both these properties result in a decrease of performance, and the respective implementations do not satisfy timing or performance demands of high-end applications. In the course of this manuscript we will describe how these unfavorable properties can be overcome by implementing the algorithm that remains efficient for large SEs, but supports only a limited number of shapes.

4 Algorithm Description

Contents

In this chapter, we describe two queue-based morphological algorithms that we consider to be convenient for the hardware and GPU implementation. We begin with *Dokládal* 1-D dilation (published in [Dokládal 2011]) algorithm as dilation is a fundamental operator and can be found in almost every compound morphological operation (erosion by duality is omitted in the text for brevity). The algorithm allows the SE composition in order to obtain 2-D rectangular and polygonal SEs according to the theory.

In a later part, we introduce a new 1-D opening algorithm referred to as *streaming peak elimination*. This algorithm computes an opening (closing by duality is omitted in the text for brevity, published in [Bartovský 2011a]) and a pattern spectrum ([Bartovský 2012a])) in $\mathcal{O}(1)$ with the same properties beneficial for hardware as the *Dokládal* algorithm. However, because an opening does not have such a composition property as a dilation (two 1-D openings do not form a 2-D opening), this algorithm can only be used for acceleration of the applications of 1-D opening, such as feature enhancement, local orientation extraction, oriented spectrum, etc.

4.1 1-D Dilation Algorithm

The dilation algorithm [Dokládal 2011], referred to as *Dokládal* hereafter, computes the dilation $y = \delta_B f$ for some 1-D finite input signal $f : \{X \subset \mathbb{Z}; X = 1 \ldots N\} \to \mathbb{R}$. The algorithm processes the input signal f sequentially, that is it computes one pixel of the dilated signal $Y = y(wp)$ at the time while reading one sample of input signal $F = f(rp)$. The coordinates wp and rp stand for the current *writing* and *reading* positions in the 1-D signal, which are generally different. The reading position represents the most recently read pixel in the input signal whereas the writing position points to the origin of the SE as depicted in Fig. 4.1. Note that $rp \geq wp$, and the delay between rp and wp is given by the right-hand size of the SE referred to as l_{right}. We consider a 1-D connected SE containing its origin described by the distance from the origin to the right-hand end l_{right} and to the left-hand end l_{left}. The result length is l_{right} + l_{left} +1.

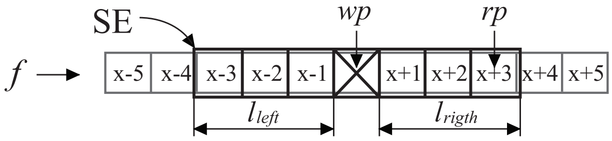


Figure 4.1: Illustration of a centered 1-D SE positioned over signal f. wp/rp writing and reading position, $l_{\text{left}}/l_{\text{right}}$ left-hand and right-hand size of SE.

As the algorithm processes one pixel at the time, the algorithm can be decomposed into the core function One_Pass_Dilation, which accepts one input pixel F, returns one output sample Y (described later, see Alg. 2), and an outer loop Dilate_1D that calls the core function for each input pixel of signal f as outlined in Alg. 1. This function iterates $N + l_{\text{right}}$ times for N-pixel signal; l_{right} is caused by the signal borders. In each cycle, the current reading and writing position is set to $rp \leftarrow column$, $wp \leftarrow column - l_{\text{right}}$ first (max() and min() operations from Alg. 1 only prevent pointers from overflowing the signal boundaries), and then the core function is called. Clearly, the used *for* loop implies the strictly sequential access to input and output data.

Algorithm 1: $y \leftarrow$Dilate_1D (f, l_{right}, l_{left}, N)

```
Input: $f$ - input signal; $l_{\text{right}}$, $l_{\text{left}}$ - SE size towards right and left end; $N$ - length of the signal
Result: $y$ - output signal

init(Q) ;                                                  // Initialize queue
for $column = 1 : N + l_{\text{right}}$ do
    $rp \leftarrow \min(column, N)$;                       // Set current reading position
    $wp \leftarrow \max(column - l_{\text{right}}, 1)$;    // Set current writing position
    $y(wp) \leftarrow$ One_Pass_Dilation ($f(rp)$, $rp$, $wp$, $l_{\text{right}}$, $l_{\text{left}}$, $N$, $Q$) ; // Call core function with one input pixel, it returns one pixel of the dilated signal
```

Now let us focus on the core function One_Pass_Dilation that applies the actual algorithm. Its main principle is to avoid unnecessary comparisons as much as possible. It is achieved by discarding all those pixels that will never take over in the result of dilation as soon as it is known. This step of the algorithm is so-called *elimination of useless values.* The computing $\delta_B f(x)$ needs only those values of f that can be seen from x when looking over the topographic profile of f. The valleys shadowed by mountains contain unneeded values, see Fig. 4.2. Notice that the masked values depend only on f. More formally, for some causal SE B, all $f(i)$ such that $f(i) \leq f(j)$ and $i < j$ are useless values and can be dropped from computation of $\delta_B f(x)$ for $\forall x \geq j$.

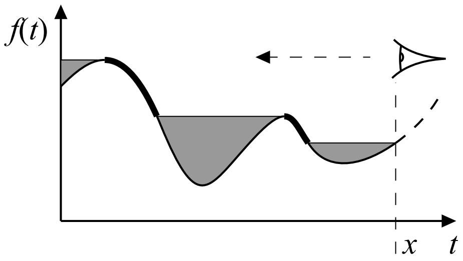


Figure 4.2: Computing the dilation $\delta_B f(x)$: Values in valleys shadowed by mountains when looking from x over the topographic relief of f are useless.

```
Algorithm 2: $Y \leftarrow$ One_Pass_Dilation ($F$, $rp$, $wp$, $l_{\text{right}}$, $l_{\text{left}}$, $N$, $Q$)
Input: $F$ - input sample $f(rp)$; $rp, wp$ - current reading, writing position;
       $l_{\text{right}}$, $l_{\text{left}}$ - SE size towards right and left end; $N$ - length of the
       signal; $Q$ - Queue
Result: $Y$ - sample of $\delta_B f(wp)$
Data: $Q$ - Queue FIFO structure
      back1($Q$).{val, pos} - accesses the latest pair {$F$, $rp$}
      front($Q$).{val, pos} - accesses the oldest pair {$F$, $rp$}

while back1($Q$).val $\leq F$ do
    dequeue($Q$) ;                              // Dequeue useless values
push($Q$,{$F$, $rp$}) ;                         // Enqueue current sample
if $wp$ - $l_{\text{left}}$ > front($Q$).pos then
    pop($Q$) ;                                  // Delete outdated value
if $rp > l_{\text{right}}$ then
    return (front($Q$).val ) ;                  // Return valid value
else
    return ({}) ;                               // Return empty
```

The core function One_Pass_Dilation is based on usage of a queue memory, a FIFO-ordered (First In, First Out) memory structure. In addition to the basic FIFO features *push()*, *pop()*, queue provides *front()*, *back1()*, and *back2()* operations to access the oldest, the latest, and the second to the latest values, respectively, but keep them at their original position unaffected. Also, *dequeue()* operation discards the most recently pushed element. Each element stored in the queue is composed of two attributes $\{F, rp\}$: the pixel gray-level value $F = f(rp)$ and its reading position rp in input data stream. Both attributes can be accessed separately, e.g., back1(Q).val accesses the value of the last element and back1(Q).pos its position.

The queue memory serves as a storage for the past values in the scope of the SE and as the main working memory. One call of the ONE_PASS_DILATION function proceeds in the following steps:

- Dequeue useless values
- Enqueue current sample
- Delete outdated value
- Return output sample

At the first step, all past values of f within the scope of the SE (these values are stored in the queue) that are found to be useless (the shadowed values in Fig. 4.2) should be dropped from the computation. A past value is useless if and only if it is lower or equal to the current value F. As a consequence, the algorithm stores only the decreasing intervals of f (represented by a thick line in Fig. 4.2). The values that happen to belong to increasing or constant intervals in the scope of SE are dropped. Therefore, the search for the useless values to drop starts from the most recently stored value and carries on until some past value greater than F is reached. The dequeuing iterates sequentially as outlined in Alg. 2 lines 1–2.

Second, the currently read value F is pushed into the queue in form of a pair $\{F, rp\}$, the sample F and its reading position rp (line 3).

Third, the outdated values are retrieved from the queue. A value is outdated when it is no longer covered by the sliding SE, and that is determined using the position of the stored value (lines 4-5).

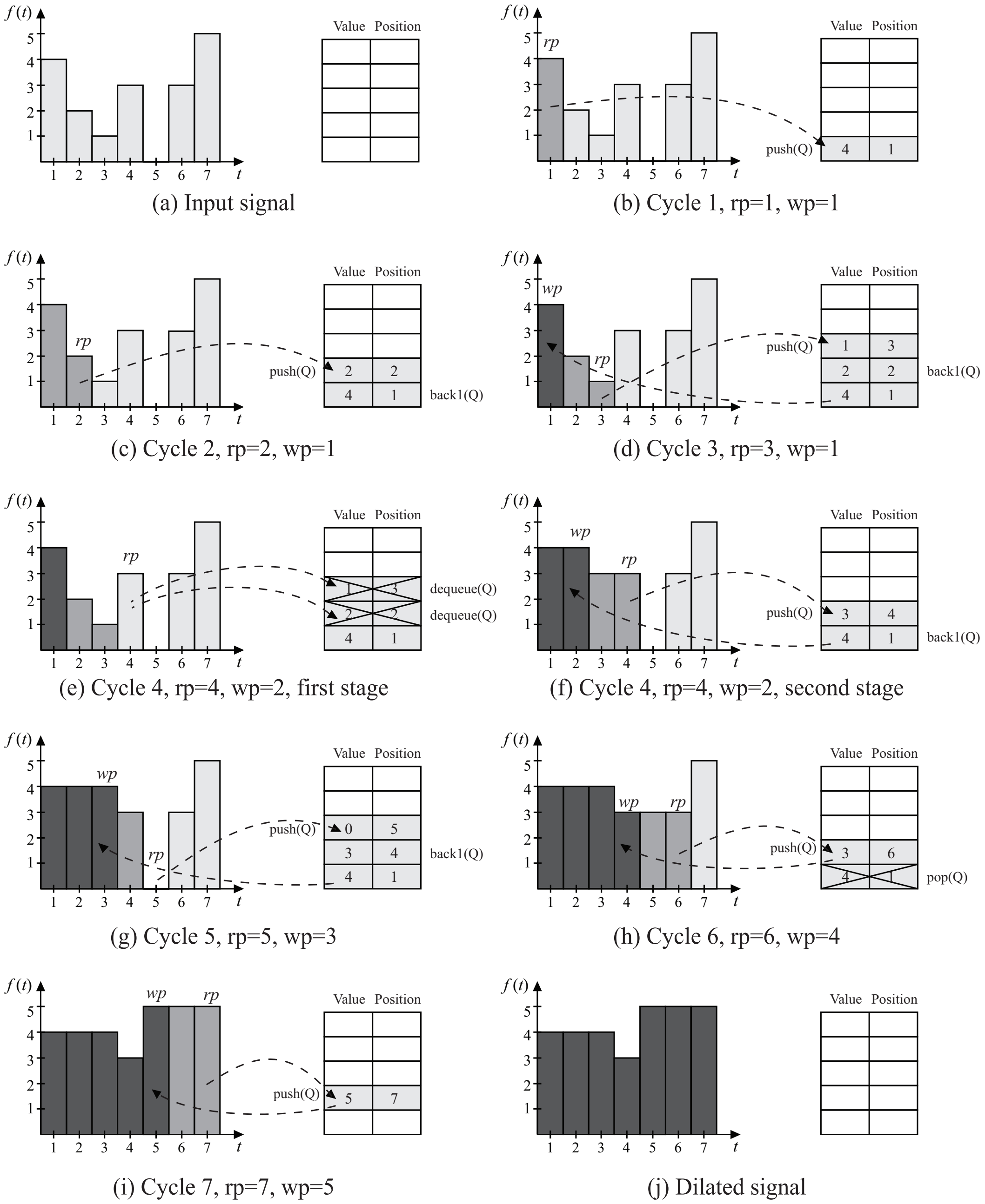
Finally, the result of the dilation is located at the front of the queue (line 7). The result becomes available as soon as enough input data have been read, otherwise the output is empty (line 9). The last condition only transforms the causal SE, which was considered in dequeuing step, to more universal non-causal SE using the property that a dilation commutes with a translation as

$$\delta_{B+t} f(x) = \delta_B f(x - t) \tag{4-1}$$

where $t \in X$.

4.1.1 Illustration of *Dokládal* Algorithm Run

Now, let us observe an illustration of the dilation algorithm run (i.e., DILATE_1D) on a signal example f with SE of $l_{\text{right}} = 2$ px, $l_{\text{left}} = 2$ px in Fig. 4.3 (a)–(i). The input signal f is depicted in (a) along with the queue initialized to empty. In the first cycle of the *for* loop (b), rp and wp are set to 1 and the core function ONE_PASS_DILATION enqueues the first pixel into the queue. The second cycle (c) also only pushes the current second pixel into the queue as its value is smaller



(a) Input signal

(b) Cycle 1, rp=1, wp=1

(c) Cycle 2, rp=2, wp=1

(d) Cycle 3, rp=3, wp=1

(e) Cycle 4, rp=4, wp=2, first stage

(f) Cycle 4, rp=4, wp=2, second stage

(g) Cycle 5, rp=5, wp=3

(h) Cycle 6, rp=6, wp=4

(i) Cycle 7, rp=7, wp=5

(j) Dilated signal

Figure 4.3: Illustration of the *Dokládal* algorithm run: (a) original input signal f, (b)–(i) iterations of the algorithm cycle-by-cycle, (j) dilated signal y. Light, medium, and dark gray rectangles denote input, stored in the queue, and output pixels, respectively. The tables on the right represent contents of the queue at the particular time. SE features: $l_{\text{right}} = 2$ px, $l_{\text{left}} = 2$ px.

than the first pixel. During the third cycle (d) the pixel at the current reading position is again pushed into the queue without dequeuing any other pixel (notice the decreasing signal interval at time samples 1–3). At this point a sufficient amount of data to produce the output at time sample 1 has been processed ($rp > l_{\text{right}}$) so the oldest pixel from the queue is returned as a result.

The fourth cycle $rp = 4$ is divided into two stages in Fig. 4.3 for clarity. The first stage (e) manages the dequeuing step of the core function. We observe that the current pixel has value 3 that is greater than the previously pushed pixel {1,3} with value 1. The latter becomes useless in dilation (it is a signal valley, see Fig. 4.2), and therefore dequeued. The dequeuing *while* loop does not stop but iterates one more time taking into account the pixel {2,2}. This pixel is also smaller, and dequeued. As far as the very oldest pixel {4,1} is reached, the dequeuing process is stopped leaving only one pixel in the queue. The second stage (f) shows the common pushing the current pixel and returning the oldest pixel.

In the fifth cycle (g) the current pixel is pushed and the oldest is returned in an ordinary way. During cycle 6 the dequeuing step erases two pixels from the queue with value ≤ 3 ({0,5}, {3,4}), and the current pixel is enqueued. Then the oldest stored pixel {4,1} is found outdated as its position is smaller than $wp - l_{\text{left}}$, and deleted by the pop() operation. As a consequence, the pixel pushed in this cycle is the oldest one and represents the output sample.

Cycle 7 reads the pixel greater than the stored one {3,6}, and therefore, replaces it (dequeue() and push() operations). The currently read value is also the output. The algorithm proceeds on in the same manner until the whole signal is processed as shown in Fig. 4.3 (j).

4.2 2-D Dilation by Rectangular SE

The separability of n-D morphological dilation into lower dimensions is a well-known and very often used property (cf. Section **??**). Hereafter, we are especially interested in decomposition of dilation by 2-D shapes, especially rectangles and polygons, into 1-D dilation. Despite the fact that a rectangle belongs to the polygon family of shapes in geometry, we follow the traditional description that treats rectangles separately.

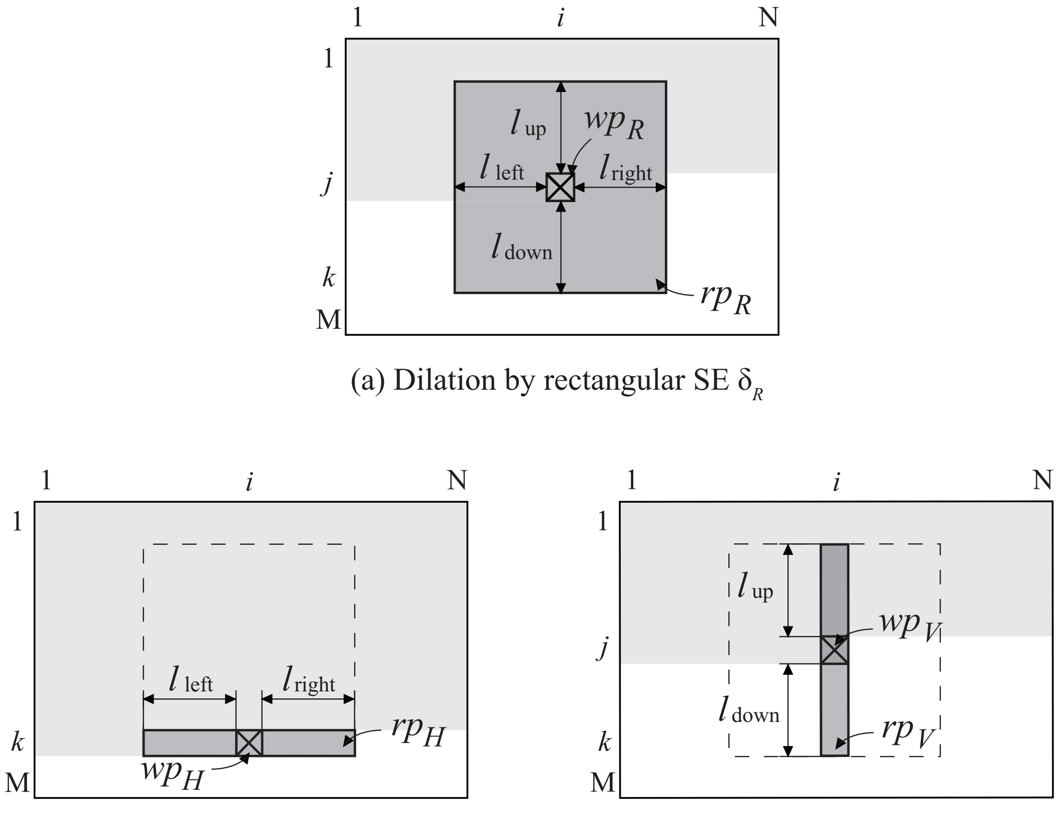
A rectangular SE R decomposes as $R = H \oplus V$ where H and V are horizontal and vertical segments and $\oplus$ is the Minkowski addition. Hence from, the dilation by rectangle R can be computed as a concatenation of two perpendicular 1-D dilations, horizontal and vertical, such as

$$\delta_R(f) = \delta_{H \oplus V}(f) = \delta_V(\delta_H(f)). \tag{4-2}$$

The concatenation of the two dilations means that each pixel of the input image is first processed by horizontal and then by vertical dilation. Effectively all the 1-D algorithms mentioned in Chapter 3 uses two image scans to obtain rectangles, one for horizontal and one for vertical dilation because of unlike orientation of the two image scans. This approach that needs the whole image to be stored in the memory is reasonable for GPP and GPU, but limiting for dedicated hardware due to large memory occupation and latency.

Our approach, on the other hand, aims at preserving stream processing for 2-D dilation. It can be achieved only when both horizontal and vertical dilations process an image in the same stream, i.e., with sequential access to data. Let us assume now that we can compute vertical dilation with horizontal scan access to data. Such vertical dilation is described below in Section 4.2.1.

The example of the dilation by rectangle $R{=}H \oplus V$ of an $N \times M$ image is shown in Fig. 4.4 (a). The image is sequentially read in the common horizontal raster scan, line by line from left to right. The various indices rp and wp denote reading and writing positions of the segments H and V, and the rectangle R, respectively. The computation is illustrated for column i and line j, i.e., the result $\delta_R f(i,j)$ is to be written at writing position wp_R.



(a) Dilation by rectangular SE δ_R

(b) Horizontal dilation δ_H

(c) Vertical dilation δ_V

Figure 4.4: Decomposition of the dilation by rectangle R (a) into two 1-D dilations by horizontal segment H (b) and vertical V (c) according to (4-2). Light gray denotes pixel already output by the respective dilation.

The computation of $\delta_R = \delta_V \delta_H$ decomposes as follows: The current reading position of δ_R rp_R coincides with reading position of horizontal dilation rp_H in Fig. 4.4 (b), that is $rp_R = rp_H$. The result of the horizontal dilation on line k at writing position wp_H is immediately read by the vertical dilation in the corresponding column i in Fig. 4.4 (c), that is $wp_H = rp_V$. The result of the vertical dilation δ_V is written at the writing position wp_R in Fig. 4.4 (a), i.e., $wp_V = wp_R$.

The very important property of this method is that there is no necessary intermediate storage between horizontal and vertical dilation, that is $wp_H = rp_V$ always holds. The latency of δ_R is then given by distance between rp_R and wp_R in the sense of the image scan such as

$$T_{\text{latency}} = l_{\text{right}} + l_{\text{down}} \times N \qquad \text{[px]} \tag{4-3}$$

which is the minimal, further irreducible operator latency.

The origin of R is positioned within the SE by l_{left}, l_{right}, l_{up}, and l_{down} from the left, rigth, up, and down edge of the rectangle, respectively.

4.2.1 1-D Vertical Dilation

The main difficulty of the 1-D vertical dataflow dilation δ_V from (4-2) is to handle the unlike orientation of the vertical SE and the horizontal dataflow. An intuitive solution one can think of uses one vertical instance of the 1-D dilation algorithm per column. As the input data are fetched in the horizontal scan, each vertical dilation has actually only one pixel to process in a course of the whole line; it is kept waiting for the next data to process during the rest of the line. However, such an approach would be inefficient regarding the synchronization between the respective dilations. Because access to data is to be strictly sequential on both input and output ports, the overlap of processing any two adjacent pixels (i.e., in columns k and $k+1$) is undesirable and can be eliminated when the processing of pixels is exclusively column-ordered. That means the pixel in column $k+1$ is not read until the vertical dilation processes the pixel in column k. Then N vertical dilation instances can be replaced by only one dilation δ_V operating with an array of N independent queues called Array of queues AQ. The operator δ_V then uses i-th queue $AQ\ (i)$ for k-th column only.

Figure 4.5 (a) illustrates the coverage of the vertical queues in the image to process after reading the whole line j. In a course of this line, the result pixels in respective columns were written in line i. After that the algorithm starts reading line $j+1$ writing result into line $i+1$. It uses a dedicated queue for computation in each column shifting thus its coverage one pixel down. The situation of processing k-th pixel in this line ($[j+1{,}k]$) looks like outlined in Fig. 4.5 (b). When $[j+1{,}k]$ is processed, the result in k-th column is written to $[i+1{,}k]$, and the next pixel $[j+1{,}k+1]$ can be read, and so on.

In this way the algorithm applies the vertical-oriented 1-D dilation on the horizontally scanned dataflow. Even though this approach increases the number of

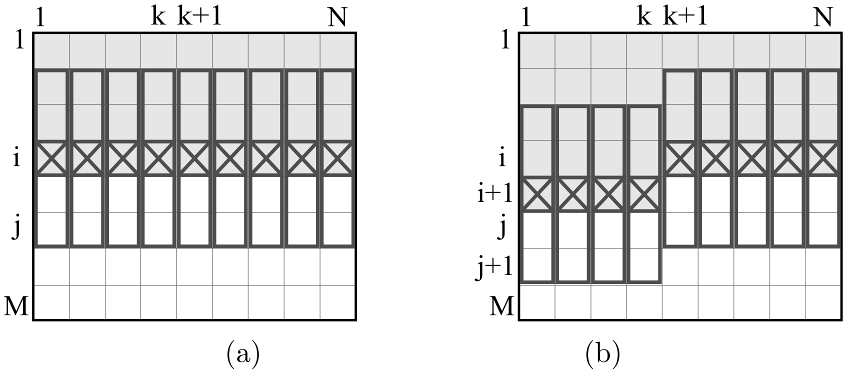


Figure 4.5: Position of N queues within the image: (a) after reading the whole line j (writing the result in line i); (b) after processing the k-th pixel of line $j+1$. Light gray denotes the output pixels already written.

queues to N (in the case of vertical image scan, only one queue is necessary), it actually makes hardware implementation of rectangles feasible as it eliminates the necessity of image storage. The queues represents the working memory so their allocated size defines memory requirements.

4.2.2 2-D Algorithm for Rectangles

At this moment we assemble perpendicular 1-D computations described above into the 2-D computation thanks to the sequential access to data at both levels, 2-D and 1-D, and for both input and output data. There is no additional latency and no intermediate storage of data between the two dilations.

The simplified algorithm for dataflow dilation by rectangles is outlined in Alg. 3. The algorithm processes the 2-D image in an ordinary double loop. For each pixel of the image, a set of reading and writing pointers is determined (lines 5–8), the horizontal dilation is computed (line 11) and stored in dF auxiliary variable, which is used as input value for the vertical dilation (line 15). The conditions on lines 9 and 14 only handles the image boundaries.

4.2.3 GPP Experimental Results of *Dokládal* Algorithm

We present the execution time of two basic benchmarks of *Dokládal* algorithm with rectangular SE in the following. We intend to illustrate the computational complexity of this dilation algorithm as well as to provide a comparison with other efficient algorithms namely Soille *et al.* ([Soille 1996]), Van Droogenbroeck and Buckley ([Van Droogenbroeck 2005]), Urbach and Wilkinson ([Urbach 2008]), Lemire ([Lemire 2006]), and OpenCV 2.2 library ([OpenCV 2012]). The benchmarks were performed on Intel Xeon E5620 @2.4GHz CPU using gcc compiler with -03 optimization. The time reported in the tables below refers to the user CPU time consumed by the respective algorithms averaged over 100 independent runs.

Algorithm 3: $y \leftarrow$ DILATE_RECTANGLE $(f,\ l_{\text{right}},\ l_{\text{left}},\ l_{\text{down}},\ l_{\text{up}}\ ,\ N,\ M)$

```
Input: $f$ - input image; $l_{\text{right}}, l_{\text{left}}, l_{\text{down}}, l_{\text{up}}$ - SE size towards right, left,
       down, and up end; $N$ - image width; $M$ - image height
Result: $y$ - output image
Data: $Q, AQ$ - Queue memories

init($AQ$) ;                                        // Initialize array of vertical queues
for $line = 1 : M + l_{\text{down}}$ do
    init(Q) ;                                       // Initialize horizontal queue
    for $column = 1 : N + l_{\text{right}}$ do
        $rp\_hor \leftarrow \min(column, N)$;                      // Set horizontal rp
        $wp\_hor \leftarrow \max(column - l_{\text{right}}, 1)$;   // Set horizontal wp
        $rp\_ver \leftarrow \min(line, M)$;                        // Set vertical rp
        $wp\_ver \leftarrow \max(line - l_{\text{down}}, 1)$;      // Set vertical wp
        if $line \leq M$ then
            $F \leftarrow f(rp\_hor + N \times line)$;             // Read input pixel
            $dF \leftarrow$ ONE_PASS_DILATION $(F,\ rp\_hor,\ wp\_hor,\ l_{\text{right}},\ l_{\text{left}},\ N,\ Q)$
            ;                                       // Call core function for horizontal orientation
        else
            $dF \leftarrow \emptyset$ ;                              // Use empty value outside image
        if $column > l_{right}$ then
            $Y \leftarrow$ ONE_PASS_DILATION $(dF,\ rp\_ver,\ wp\_ver,\ l_{\text{down}},\ l_{\text{up}},\ N,$
            $AQ(wp\_hor))$ ;                        // Call core function for vertical orientation
            $y(wp\_ver + N \times wp\_hor) \leftarrow Y$ ;         // Write output pixel
```

We use the mountain natural photo as a testing image, originally introduced in [Van Droogenbroeck 2005].

The first benchmark in Fig. 4.6 plots dependence of the algorithm on the size of square SEs. The *Dokládal* algorithm shows only small variation of the execution time throughout the entire scale verifying declared complexity $\mathcal{O}(1)$. Observing the other algorithms, Van Droogenbroeck outperforms our solution by a firm rate (2.5× in average). However, this algorithm has a couple of properties that are limiting when either different settings is used or hardware implementation is considered. They are: (i) use of histogram, limiting for high-precision data and hardware; (ii) large memory requirements of $2MN$ unsuitable for hardware; (iii) non-sequential access to data, which also makes hardware implementation difficult. The other fast solution is the one provided by popular OpenCV 2.2 library. It performs quite well thanks to the use of highly optimized Intel Processing Primitives. Soille and Lemire perform little worse than *Dokládal*. Urbach and Wilkinson performs the worst from the selected algorithms as their algorithm is designed for arbitrary-shaped 2-D SE and it becomes inefficient for large, regular SEs, which can be easily decomposed into lower dimensions.

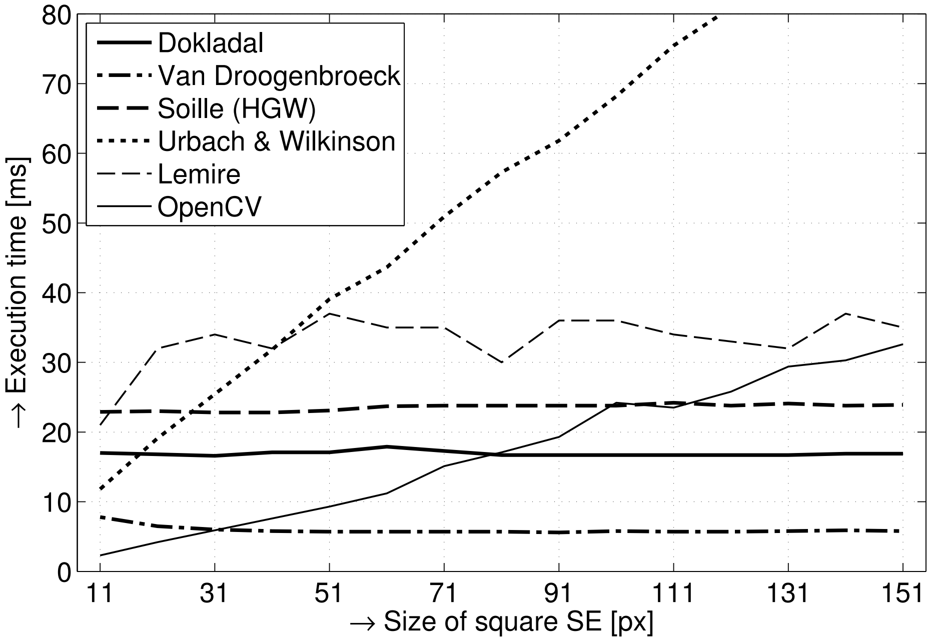


Figure 4.6: Execution time of dilation versus the SE size, SVGA natural image.

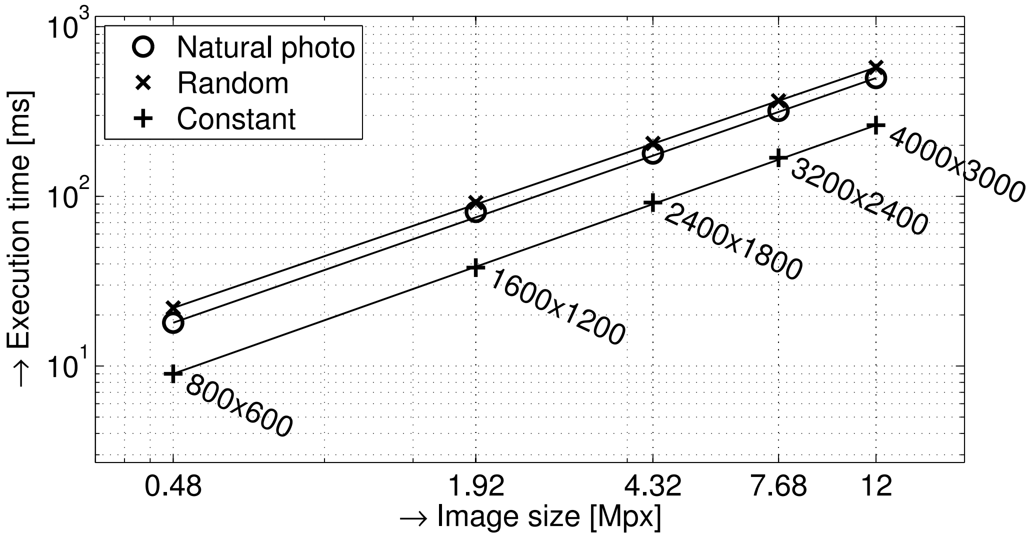


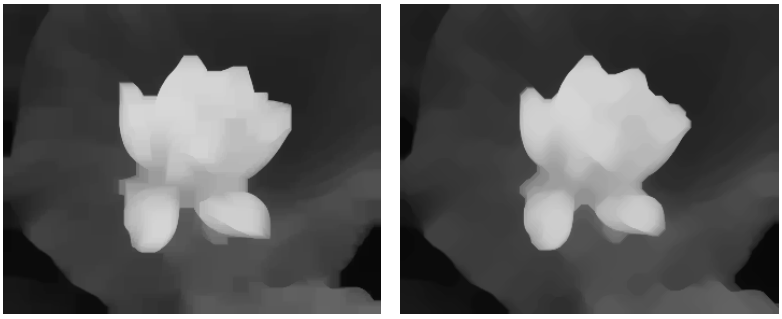
Figure 4.7: Execution time of dilation versus the image size. The structuring element is square 101×101 px.

Second, we evaluate the execution time with respect to the image size in Fig. 4.7. The results for each image type lie on a perfect straight lines (straight lines connect values obtained for the smallest and the largest image) in the log-log scale that illustrates the complexity of *Dokládal* algorithm is effectively independent of the image size. However, due to the bounded image support, the exact complexity per image is little affected by the SE size as $\mathcal{O}_{\text{image}}((N + l_{\text{right}})(M + l_{\text{down}}))$. The reason for extra algorithm iterations is the algorithm latency. In other words, when the last pixel of the image is processed, there are still $l_{\text{down}}N$ pixels to be output. As these iterations do not read new input pixels, the dequeuing process is not involved, and consequently, they are faster then ordinary iterations. So the complexity $\mathcal{O}_{\text{image}}((N+l_{\text{right}})(M+l_{\text{down}}))$ reaches $\mathcal{O}_{\text{image}}(NM)$ provided $l_{\text{right}} \ll N$ and $l_{\text{down}} \ll M$.

The same Fig. 4.7 also shows dependence on the image content. Such dependence is caused by the dequeuing loop that may iterate multiple times, or not at all, during processing one pixel with respect to the past image data stored in the queue. The random number of loop iterations does not change the complexity because each pixel is at most once pushed into the queue and also at most once discarded from the queue (by either pop() or dequeue() operations). The only thing that changes pixel by pixel is when the pixel is discarded. In the case of constant image, the loop iterates exactly once per pixel, but for random noise image, the number of iterations per pixel is random. This uncertainty makes various CPU optimizations, such as branch prediction, more difficult and tends to slow down the computation. However, the natural photo image used in benchmarks, which contains some superimposed noise, performs at similar performance as the worst-scenario random noise image.

4.3 Polygonal SE

Rectangular SEs are very popular for their simplicity. However, they suffer from angular anisotropy as the difference between the side length and the diagonal length is significant, see Fig. 4.8. Such a SE may not be feasible for some operations, such as image enhancement, filtering, or granulometry.



(a) ASF^6 by rectangle (b) ASF^6 by octagon

Figure 4.8: Image filtered by (a) rectangles, and (b) octagons. Notice better isotropy of octagons.

These operations are highly sensitive to the shape of the SE, so, unless one has some a priori knowledge of image contents, circular SEs are preferred. However, circles are very difficult to implement efficiently because there is little regularity in their shape (discrete), and none of the SE decomposition methods can be used for efficient implementation. As a consequence, circles are often approximated by regular polygons (all sides have the same length) that are easily decomposable into 1-D SEs, cf. [Adams 1993, Xu 1991]. Any $2n$-top ($n \in \mathbb{N}$) regular polygon SE P_{2n} can be decomposed into a set of n line SEs L_{α_i}, $i = [1, n]$,

$$P_{2n} = \underbrace{L_{\alpha_1} \oplus \cdots \oplus L_{\alpha_n}}_{n \text{ times}} \tag{4-4}$$

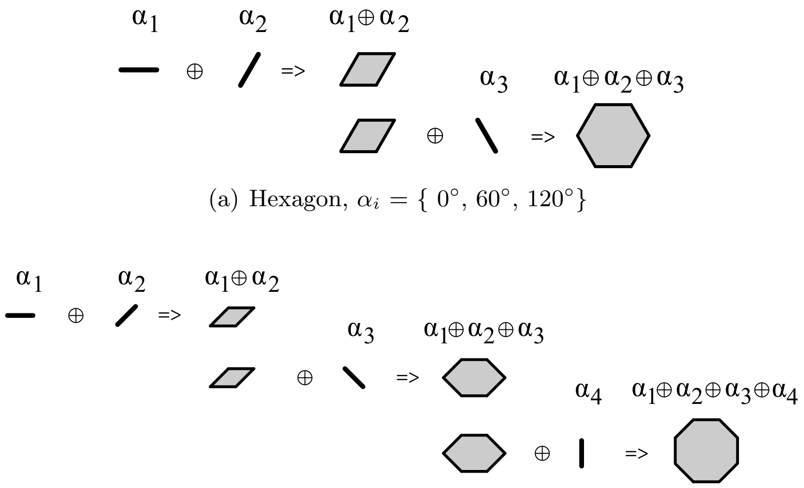
oriented by angles α_i, such that

$$\alpha_i = (i-1)\frac{180°}{n} \quad [°] \tag{4-5}$$

The length of all L_{α_i} is equal to the side of the desired polygon and can be computed from the circumcircle radius r as

$$\|L_{\alpha_i}\| = 2r \sin\left(\frac{180°}{2n}\right). \tag{4-6}$$

For example, a hexagon can be obtained by three L_{α_i} oriented in $\alpha_i = \{\, 0°, 60°, 120°\}$ on a 6-connected grid, and an octagon by four L_{α_i}, $\alpha_i = \{\, 0°, 45°, 90°, 135°\}$ using an 8-connected grid, see Fig. 4.9.



(a) Hexagon, $\alpha_i = \{\, 0°, 60°, 120°\}$

(b) Octagon, $\alpha_i = \{\, 0°, 45°, 135°, 90°\, \}$

Figure 4.9: Polygon SE composition of line SEs. (a) hexagon is composed of 3 segments, (b) octagon is composed of 4 segments. $\oplus$ operator stands for the Minkowski addition; α_i stands for L_{α_i}.

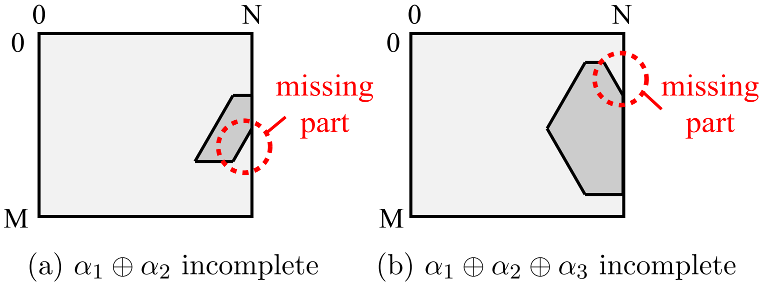
Hence from (2-15) and (4-4), a 2-D dilation by a $2n$-top polygon $\delta_{P_{2n}}$ of some function $f : \mathbb{R}^2 \to \mathbb{R}$ can be obtained by n consecutive 1-D dilations $\delta_{L\alpha_i}$, $i = [1, n]$, by line segments oriented by α_i as

$$\delta_{P_{2n}}(f) = \underbrace{\delta_{L\alpha_1}(\ldots\ \delta_{L\alpha_n}}_{n \text{ times}}(f)). \tag{4-7}$$

The aforementioned decomposition holds true for the unbounded support $\mathbb{Z}^2$. However, when using real images with a bounded support $D \subset \mathbb{Z}^2$, $D = [1, M] \times [1, N]$, decomposition boundary effects appear if at least one $L_{\alpha_i} \neq \{0°, 90°\}$ is used. The cause is that the Minkowski addition of all decomposed line segments of (4-4), which are cropped by image boundaries after every L_{α_i} of that concatenation, does not necessarily correspond to P_{2n} cropped by image boundaries just once as desired. It is expressed by the following expression where $D\cap$ represents an intersection with the image support D

$$D \cap (L_{\alpha_1} \oplus \ldots \oplus L_{\alpha_n}) \neq D \cap (L_{\alpha_n} \oplus \ldots D \cap (L_{\alpha_2} \oplus D \cap (L_{\alpha_1}))). \tag{4-8}$$

The illustrative example of such boundary effects with a hexagonal SE is depicted in Fig. 4.10. We can see that the composition $\alpha_1 \oplus \alpha_2$ is incomplete compared to the desired one in Fig. 4.9; a small part of the SE is missing. It holds true even for the entire hexagon, the composition $L_{\alpha_1} \oplus L_{\alpha_2} \oplus L_{\alpha_3}$ is also incomplete. It is caused by the right boundary cropping not only the final P_{2n}, but also all intermediate results. The cropped values are later missing to form an appropriate polygon section.



(a) $\alpha_1 \oplus \alpha_2$ incomplete (b) $\alpha_1 \oplus \alpha_2 \oplus \alpha_3$ incomplete

Figure 4.10: Polygon SE composition without padding. The desired SEs presented in Fig. 4.9 are incomplete, a small triangle is missing.

This issue is solved by adding a padding to the image. The section of P_{2n} contained inside the image support is then complete, the missing part of P_{2n} is located in the padding area. The added padding contains recessive values, i.e., values that do not affect the computation of particular morphological operator (for $f{:}D \to V$, $\wedge V$ for dilation, $\vee V$ for erosion). Thickness of the padding is different in horizontal and vertical direction and is determined by the size of oblique segments, particularly by the half of vertical and horizontal projection

$$B_{\mathrm{H}} = \|L_{\alpha_i}\| \cos(\alpha_2)/2 \quad \text{[pixels]} \tag{4-9}$$

$$B_{\mathrm{V}} = \|L_{\alpha_i}\| \sin(\alpha_2)/2 \quad \text{[pixels]}. \tag{4-10}$$

The 1-D *Dokládal* algorithm described above can be used for the computation of L_{α_i} segments in a stream as well. Then, for example, three instances of this algorithm can be concatenated to compose a hexagon as depicted in Fig. 4.11. The image is sequentially read by horizontal $L_{0°}$ at the global reading position rp_P (a).

The result of the horizontal segment is immediately provided as input to the first oblique $L_{60°}$ at (b) so that the reading position of $L_{60°}$ coincides with the writing position of $L_{0°}$. By the very same rule, the result of the $L_{60°}$ is brought as input data to the second oblique α_3 at (c), the writing position of which is the writing position of the complete polygon (d). The total latency is then defined by distance between reading (a) and writing positions (d) of the polygon P.

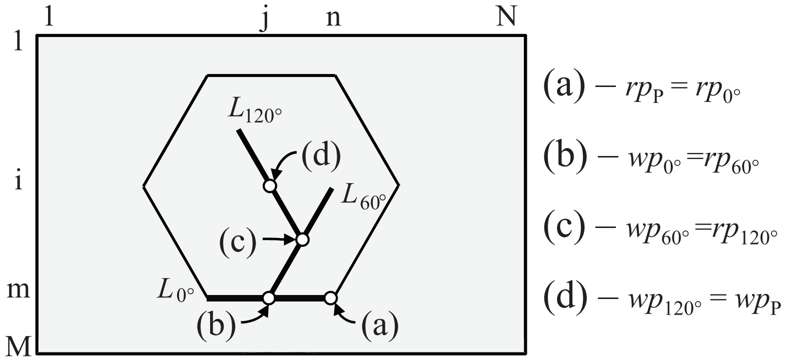


Figure 4.11: Stream concatenation of three L_{α_i} into a hexagonal SE P. rp/wp - reading/writing position.

Considering the computation of particular segments of the polygon decomposition (4-7), the horizontal segment $L_{0°}$ as well as the vertical $L_{90°}$ is computed on the corresponding line or column separately. There is much redundancy in the computation because any two adjacent segments (adjacent according to the orientation α_i) have a large overlap, e.g., two $L_{0°}$ of length l at the points $[x, y]$ and $[x + 1, y]$ have overlap of $l - 2$ points. This property allows a great speed-up by using *Dokládal* algorithm or any other algorithm that reuses some results computed for pixel $[x, y]$ in the computation of the following pixel $[x + 1, y]$. At the same time, the processing can be ordered in the way that the input and output are read and written in the raster scan provided that each pixel is read exactly once.

The aforementioned manner of computation would bring a considerable speed-up for oblique segments if it was preserved. Generally, for L_{α_i}, $\forall \alpha_i \neq \{0°, 45°, 90°, 135°\}$ at 8-connected grid or $\forall \alpha_i \neq \{0°, 60°, 120°\}$ at 6-connected grid, the overlap of two adjacent segments is both small and disconnected.

This fact has an impact on the computation of hexagons (or other polygons more complex than octagons, e.g., decagon, dodecagon; we will focus on hexagons only) that can be carried out using either 6-connected or 8-connected grid. When using the 6-connected connectivity, which is usually artificially made from 8-connectivity for the purpose of hexagonal neighborhoods, the *Dokládal* algorithm allows for perfectly shaped hexagons. In the following implementation sections we will use the 6-connected grid for hexagons.

In the case of 8-connected grid, the exact computation of hexagons cannot be done in the way of partitioning the image into inclined corridors as in the case of scanning in rows or columns, or 6-connectivity. However, when the computation is done in inclined corridors it retains the large overlap of two adjacent L_{α_i} and enables the computation speed-up. As a downside, it makes L_{α_i} translation variant, and consequently, the resulting hexagon is approximated, see Section 4.3.2.

4.3.1 Oblique 1-D Structuring Element

The oblique segments included in a hexagon and octagon, i.e., L_{α_i}, $\alpha_i = 45°$, $60°$, $120°$, $135°$, need some mapping in order to determine which pixels belong to L at given position. Because all inclinations verify $\alpha_i \geq 45°$, and the tangent coefficients k_i verify $k_i = \tan \alpha_i \geq 1$, modifying the column position *column* handles the mapping of an image into an inclined space. The inclination deviation from the vertical direction is then obtained by additive constant line/k_i. Hence at the position [*line*, *column*], the shifted position corresponds to $[\mathit{line}, \mathit{column}_{\text{shift}}]$,

$$\mathit{column}_{\text{shift}} = (\mathit{column} + \mathit{line}/k_i) \bmod (N + 2B_{\text{H}}), \tag{4-11}$$

where the inclination deviation from the vertical direction line/k_i is called *offset*. Notice that in the horizontal scan *offset* changes its value only with the beginning of a new line. In our implementation of polygons we use $k_{45°,135°} = \pm 1$ for octagons on 8-connected grid, and $k_{60°,120°} = \pm 2$ for hexagons on 6-connected grid, which assure that the computation of polygonal SE is translation invariant. Unfortunately, this does not hold true for more complex polygons or other connectivity. Then the oblique segments become translation variant as it is described in the following section.

Algorithm 4: *offset* ← DISCRETE_LINE (*Long_leg*, *Short_leg*, *Init*)

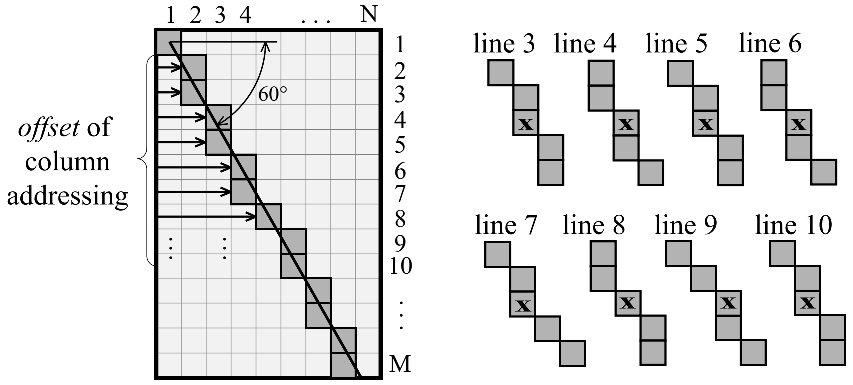
Input: *Long_leg* - long leg of the right triangle; *Short_leg* - short leg of the right triangle; *Init* - initilize/reset
Result: *offset* - inclination offset of the discrete line
Data: *error* - current error variable
offset - current offset variable

```
if $Init = true$ then
    $error \leftarrow Long\_leg/2$ ;          // Initialization of error and offset variables
    $offset \leftarrow 0$ ;
else
    $error \leftarrow error - Short\_leg$ ;
    if $error < 0$ then
        $error \leftarrow error + Long\_leg$ ;
        $offset \leftarrow offset + 1$ ;                        // Increment offset value
    return $offset$ ;                                       // Return current offset value
```

4.3.2 Translation-Variant SEs on 8-connected Grid

In the following paragraphs we will discuss the issues related to the computation of oblique SEs on 8-connected grid using the image partitioning into inclined corridors. Even though such an approach results in translation-variant SEs, it speeds up the computation through usage of an efficient 1-D algorithm. The translation variance issue on 8-connected grid has been already studied by [Soille 1996].

Fig. 4.12(a) presents an example of the mapping for L_{60° on an 8-connected grid. The pixels of the oblique corridor (emphasized by dark grey color; the nearest pixels to the precise Euclidian line oriented by the desired angle) have the same shifted column address thanks to the added *offset*. Consequently, these pixels are processed using the same queue dilating the image in an inclined direction. Obviously, the task of *offset* calculation is equivalent to the computation of discrete line coordinates from its tangent k. For the sake of an efficient implementation, we consider that $k_i \in \mathbb{Q}$, $k_i = \tan\alpha = \mathit{Long\_leg}/\mathit{Short\_leg}$ is a rational number defined by two legs of a right triangle (*Short_leg*, *Long_leg*) including α. The *offset* of the discrete line is then calculated by the simplified Bresenham line algorithm in Alg. 4. This algorithm computes *offset* by using only one addition, subtraction, and comparison of two integer numbers per one pixel.

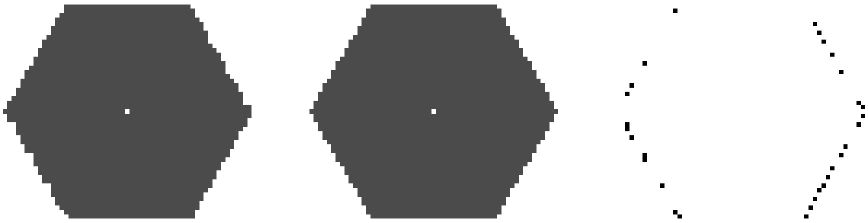


(a) (b)

Figure 4.12: Oblique linear segment computation. (a) Pixels of the discrete line lying under Euclidian line, and (b) the spatial variance of oblique segments.

The translation variance can be observed in Fig. 4.12(b). The depicted segments of the computed discrete line $\alpha = 60^\circ$ from image lines 2 to 9 display that the shape of the segment varies with translation. The effect of translation-variant 1-D segments to the result hexagon is displayed in Fig. 4.13. One can see that the hexagon on 8-connected grid has edges little bit corrupted in terms of the border pixels. In addition, the shape of such SEs is also translation variant.

In conclusion, the 6-connected grid is the preferred one for hexagons as it allows for the fast and exact computation. We will also consider 6-connectivity in the implementation part below. For more complex polygons or lines oriented by arbitrary angle more natural 8-connectivity is used resulting in a slight shape approximation and variance to translation.



(a) 8-connectivity (b) 6-connectivity (c) Difference

Figure 4.13: The translation-variant hexagon SE obtained on 8-connected grid, translation-invariant SE on 6-connected grid, and the difference.

4.4 1-D Opening Algorithm

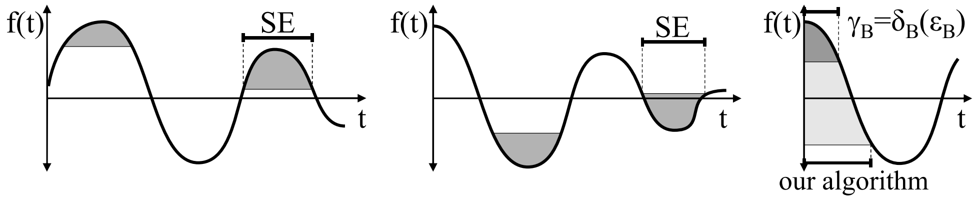
We describe the main principles of the originally proposed 1-D opening algorithm referred to as *streaming peak elimination* algorithm hereafter. We begin with the description of the main principle of the algorithm, and develop its pseudocode. Later, we enrich this algorithm by computation of the pattern spectrum in a single image scan. We also present the arbitrary angle orientation of the 1-D SE on a 2-D image support.

At first, let us observe the influence of opening γ_B (2-18) on a simple 1-D signal $f : \mathbb{Z} \to \mathbb{R}$ in Fig. 4.14. The opening literally cuts off the peaks narrower than the length l of the SE (closing on the other hand fills the valleys narrower than l). Remark here that γ_B is invariant to the translation of the SE B so γ_B is not affected by the origin of B.

In the previous sections, we have already established an assumption that sequential access to data is very beneficial, and effectively necessary, for high performance in hardware. Accordingly, our algorithm accesses both input and output data strictly sequentially with the fixed latency equal to the operator latency (l in the case of horizontal SE).

The proposed algorithm executes the recursive peak elimination with each new sample of input data. Naturally, this elimination is applied only on l recently read data covered by the current position of the SE, which slides in time over the input data f. The elimination process cuts a peak iteratively from the top downwards by each gray-level the peak contains.

Contrarily to the traditional approach in which opening is obtained through concatenation of erosion and dilation by (2-18) on page 14, our algorithm handles image borders correctly such that the peak of the whole size of l is eliminated even if it touches the border, see Fig. 4.14. The behavior of (2-18) near borders is affected by the origin of B, and it is always incorrect at either edge. If the SE is centered, (2-18) cuts only peaks narrower than $l/2$ pixels.



(a) (b) (c)

Figure 4.14: Effects of opening and closing on a 1-D signal: (a) opening cuts the peaks off; (b) closing fills the valleys; (c) opening near an edge: our algorithm processes the signal by the full length of the SE l, compared to the conventional solution (2-18) that uses only the half of the SE length.

In the following paragraphs we will describe the main principles of this opening algorithm. For better understanding let us first suppose that the input signal $f : \{X \subset \mathbb{Z}; X = [1, N]\} \to \mathbb{R}$ does not contain any constant intervals so it verifies $f(x) \neq f(x + 1)$, $x \in X$. The proposed 1-D algorithm, see Alg. 5, computes the opening $y = \gamma_B f$ in a loop by calling the core function One_Pass_Opening in Alg. 6 for each sample of f. It reads input pixel $F = f(rp)$ at the *reading position* rp and outputs pixel $Y = y(rp - l)$ of the result image in the course of a single call of core function One_Pass_Opening in Alg. 6.

```
Algorithm 5: $y \leftarrow$ Opening_1D ($f$, $l$, $N$)
  Input: $f$ - input signal; $l$ - SE size; $N$ - length of the signal
  Result: $y$ - output signal

  init(Q) ;                                               // Initialize queue
  for $column = 1 : N + l$ do
      $rp \leftarrow \min(column, N)$;                    // Set current reading position
      $y(wp) \leftarrow$ One_Pass_Opening ($f(rp)$, $rp$, $l$, $Q$, 0) ; // Call core function
      of streaming peak elimination algorithm
```

The core function One_Pass_Opening is based on usage of the queue memory. The queue memory serves as a storage for the past values in the scope of the SE and as the main working memory just like in the case of One_Pass_Dilation algorithm. One call of this function in Alg. 6 proceeds in the following steps:

- Eliminate peak values
- Enqueue current sample
- Delete outdated value
- Return output sample

At the first step, all past values of f within the scope of the SE (these values are stored in the queue) that are found to be peak values are dropped. A peak is a point of the input signal $f(x)$ when both its very precedent $f(x-1)$ and subsequent $f(x+1)$ points are smaller, such as

$$f(x) > f(x-1) \text{ and } f(x) > f(x+1) \tag{4-12}$$

In order to reveal a peak, the algorithm recognizes 4 possible configurations of these three points ((a) and (b) are peaks, (c) and (d) are not), see Fig. 4.15, each of which is treated in a different manner.

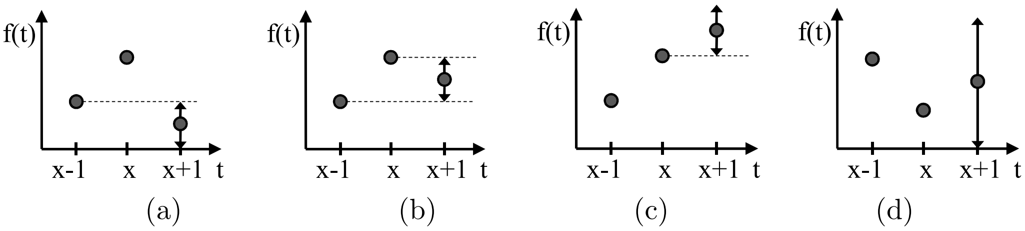


Figure 4.15: Four different pixel configurations for peak identification. Conf. (a) and (b) characterize a peak, conf. (c) and (d) do not.

The points $f(x-1)$, $f(x)$, and $f(x+1)$, which are needed for the peak elimination, are reachable in the queue as follows:

- $f(x-1) \leftarrow$ back2(Q).val (second to the latest)
- $f(x) \leftarrow$ back1(Q).val (latest)
- $f(x+1) \leftarrow F = f(rp)$ (current)

The peak elimination step proceeds in one main while loop (code line 1) that ensures the condition $f(x+1) < f(x)$ of (4-12). If two consecutive pixels are equal $f(x+1) = f(x)$, the first one is erased from the queue (line 2-4), and replaced by the second one. As a consequence, a flat plateau (zone of constant value) is represented in the queue by the last pixel and its position. We can see that the initially imposed premise of no constant intervals in f was introduced only for the sake of simplicity of configurations; the constant intervals are handled correctly.

Then the condition $f(x-1) < f(x)$ of (4-12) is tested (back2(Q).val< back1(Q).val on line 6). If the result is false, $f(x)$ is not a peak and the elimination loop is quit (configuration (d), line 11). Otherwise, $f(x)$ is a peak and will be erased from the queue (line 9) and replaced by either $f(x-1)$ in configuration (a) (back2(Q).pos $\leftarrow$ back1(Q).pos on line 8), or by $f(x+1)$ in configuration (b) (needs only line 9). This is decided upon condition $f(x+1) < f(x-1)$ ($F <$ back2(Q).val on line 7). Obviously, the while loop iterates until a non-peak configuration ((c) or (d)) is encountered.

When all peaks in the scope of B are erased, the current pixel value is unconditionally pushed into the queue along with the current reading position (line 12). The oldest stored pixel is checked whether it has been stored in the queue for too long. This check is carried out by comparing the stored reading position plus the SE size with current rp (line 13). Outdated values are immediately deleted. The

Algorithm 6: $Y \leftarrow$ One_Pass_Opening (F, rp, L, Q, α)

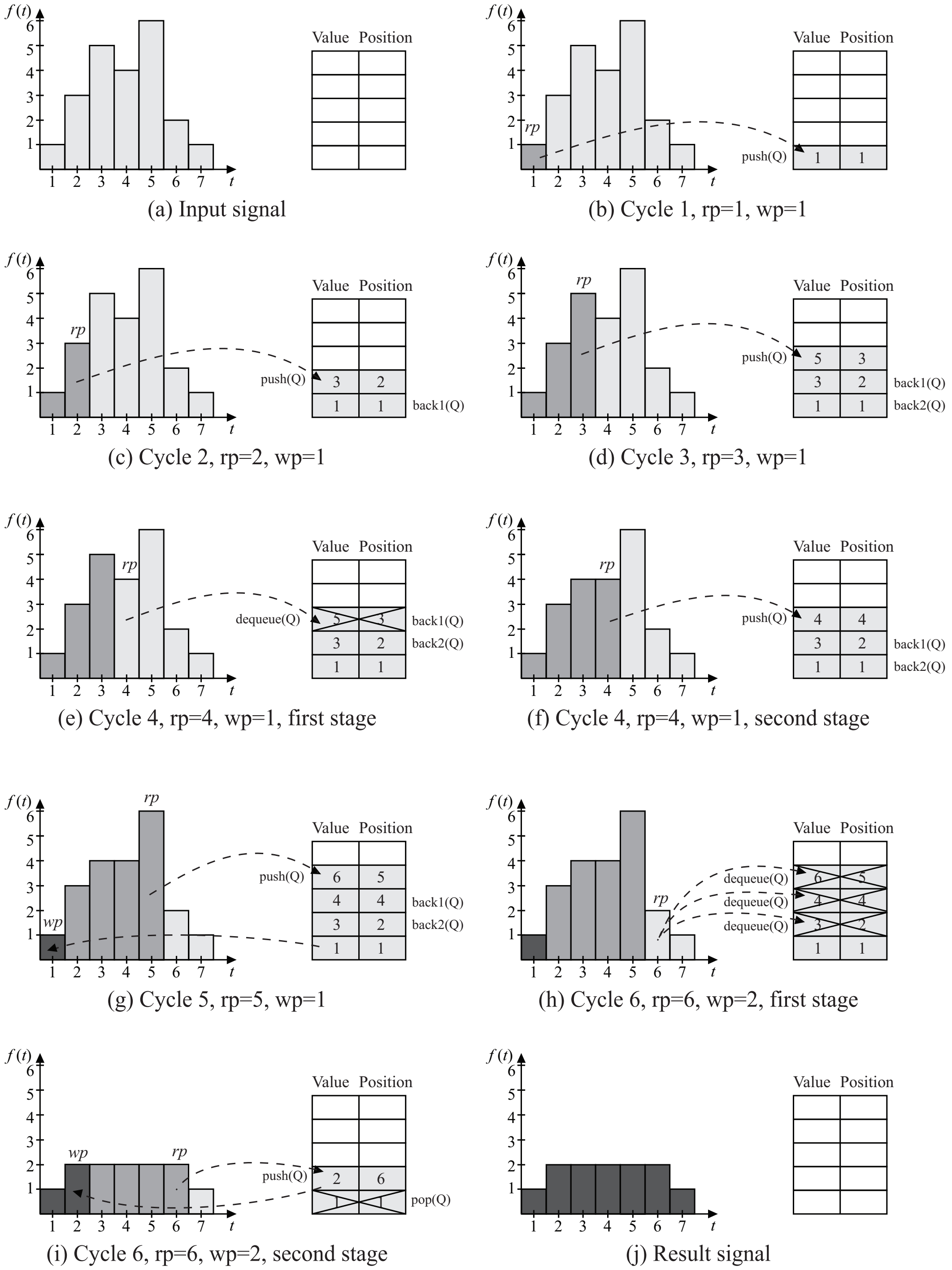
```
   Input:  $F$ - input sample $f(rp)$; $rp$ - current reading position; $l$ - SE size; Q -
           Queue; $\alpha$ - angle
   Result: $Y$ - sample of $y(rp - l)$
   Data:   Q - Queue
           back1($Q$).{val, pos} - accesses the latest pair {$F$, $rp$}
           back2($Q$).{val, pos} - accesses the second to the latest pair {$F$, $rp$}
           front($Q$).{val, pos} - accesses the oldest pair {$F$, $rp$}
1  while $F \leq$ back1($Q$).val do
2      if $F =$ back1($Q$).val then
3          dequeue($Q$) ;                              // Remove equal values
4          break ;
5      else
6          if back2($Q$).val<back1($Q$).val then
7              if $F <$ back2($Q$).val then
8                  back2($Q$).pos ← back1($Q$).pos ;    // Configuration (a)
9              dequeue($Q$) ;          // Discard peak, configuration (b), (a)
10         else
11             break ;                                 // Configuration (d)
12 push($Q$,{$F$, $rp$}) ;                             // Enqueue current sample
13 if $rp =$ front($Q$).pos $+ l$ then
14     pop($Q$) ;                                      // Delete outdated value
15 if $rp \geq l$ then
16     return (front($Q$).val ) ;                      // Return opening sample
```

oldest stored value front(Q).val is the result of Alg. 6 as soon as rp exceeds the SE size l (line 15).

4.4.1 Illustration of *Streaming Peak Elimination* Algorithm Run

Now, let us observe an illustration of the *streaming peak elimination* algorithm run (i.e., Opening_1D) on a signal example f with SE of $l = 5$ px in Fig. 4.16 (a)–(i). The input signal f is depicted in (a) along with an empty queue. In cycles 1 through 3 the new read samples are only pushed into the queue as there are no peaks on f present. Notice for instance in (d) that the first three pixels constitute a monotonously increasing signal that conforms to the non-peak configuration (d).

Cycle 4 represents more interesting behavior, and therefore it is divided into two stages. At the first one in Fig. 4.16 (e) the current pixel {4,4} reveals a peak on the input signal f because $f(x+1) < f(x) > f(x-1)$ having the values $f(x+1) = 4$, $f(x) = 5$, $f(x-1) = 3$. So the pixel {5,3} is dequeued. No other peak on f is



(a) Input signal

(b) Cycle 1, rp=1, wp=1

(c) Cycle 2, rp=2, wp=1

(d) Cycle 3, rp=3, wp=1

(e) Cycle 4, rp=4, wp=1, first stage

(f) Cycle 4, rp=4, wp=1, second stage

(g) Cycle 5, rp=5, wp=1

(h) Cycle 6, rp=6, wp=2, first stage

(i) Cycle 6, rp=6, wp=2, second stage

(j) Result signal

Figure 4.16: Illustration of the *streaming peak elimination* algorithm run: (a) original input signal f, (b)–(i) iterations of the algorithm cycle-by-cycle, (j) result signal y. Light, medium, and dark gray rectangles denote input, stored in the queue, and output pixels, respectively. The tables on the right represent contents of the queue at the particular time. SE features: $l = 5$ px.

present, so the algorithm can proceed to pushing the current pixel in the queue, see the second stage in Fig. 4.16 (f). The next pixel read in cycle 5 (g) has no impact on peak elimination due to its value is greater then the others and is just pushed to the queue. At this moment the large enough portion of data has been processed to produce the correct result of the opening by SE with $l = 5$ px, so the oldest sample in the queue is written on output.

The sixth cycle is divided to two stages, too. During the peak elimination step (h), first the pixel {6,5} is dropped, then the pixel {4,4} is dropped, and finally the pixel {3,2} is dropped, all by configuration (a). In the second stage (i) the current pixel is pushed into the queue. The oldest sample {1,1} is found to be outdated and is erased from the queue. That makes the currently pushed pixel the oldest one (the only one) and also the output value. The result signal after the algorithm finishes is depicted in Fig. 4.16 (j)

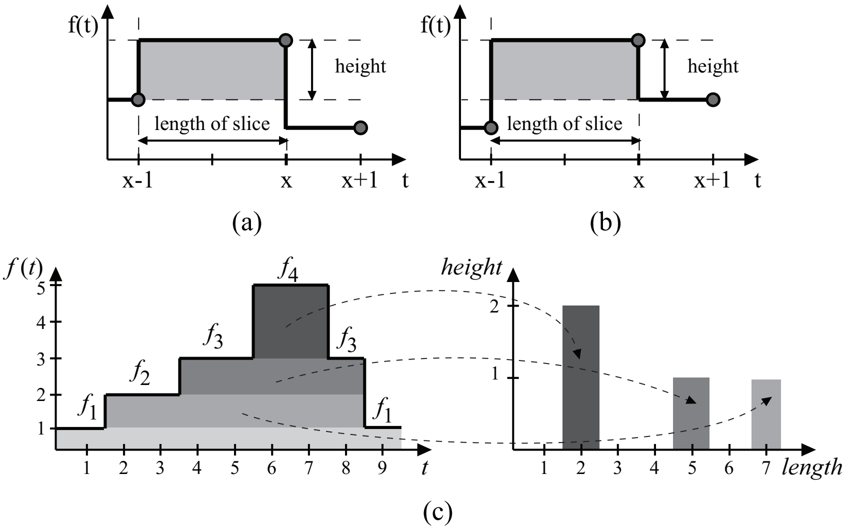


Figure 4.17: Size spectrum increment for: (a) configuration (a); and (b) configuration (b). (c) The peak is sliced by each gray level it contains.

4.4.2 Pattern Spectrum from Opening

The algorithm presented so far computes opening using the principle of peak elimination that discards the peak values. Now, we extend the opening core function Alg. 6 to compute the pattern spectrum PS, see the extended core function in Alg. 7. As the opening algorithm eliminates a peak successively level-by-level, it literally cuts the peak by each gray level the peak contains. The peak slices of the same length are can be then accumulated in the same variable to obtain the pattern spectrum, which is usually obtained by openings of increasing size, see Section 2.0.4 on page 16. The length of the slice is determined by the distance between its endpoints $f(x)$ and $f(x-1)$ that need not necessarily be 1 but varies from 1 to $l-1$ ($f(x \pm 1)$ designates precedent/subsequent value of $f(x)$). See the computation of the length on line 7. The height of a peak obviously depends on the mutual relationship of $f(x+1)$ and $f(x-1)$ according to configuration (a), (b), see Fig. 4.17 and lines 9, 12. The increment of an appropriate length in the pattern spectrum is carried out on line 13. Thereby we obtain the pattern spectrum PS with minimum extra effort.

```
Algorithm 7: $Y \leftarrow$ ONE_PASS_OPENING_PS ($F$, $rp$, $L$, $Q$, $\alpha$)
  Input: $F$ - input sample $f(rp)$; $rp$ - current reading position; $l$ - SE size; Q -
         Queue; $\alpha$ - angle
  Result: $Y$ - sample of $y(rp-l)$; $PS$ - pattern spectrum
  Data: Q - Queue
          back1($Q$).{val, pos} - accesses the latest pair $\{F, rp\}$
          back2($Q$).{val, pos} - accesses the second to the latest pair $\{F, rp\}$
          front($Q$).{val, pos} - accesses the oldest pair $\{F, rp\}$

 1 while $F \leq$ back1($Q$).val do
 2     if $F =$ back1($Q$).val then
 3         dequeue($Q$) ;                                      // Remove equal values
 4         break ;
 5     else
 6         if back2($Q$).val<back1($Q$).val then
 7             $length \leftarrow$ back1($Q$).pos $-$ back2($Q$).pos ;     // Length of the slice
 8             if $F <$ back2($Q$).val then
 9                 $height \leftarrow$ back1($Q$).val - back2($Q$).val ;   // Height of the slice
10                 back2($Q$).pos $\leftarrow$ back1($Q$).pos ;            // Configuration (a)
11             else
12                 $height \leftarrow$ back1($Q$).val - $F$ ;              // Height of the slice
13             $PS(\alpha, length) \leftarrow PS(\alpha, length) + height$;  // Accumulation of PS
14             dequeue($Q$) ;                    // Discard peak, configuration (b), (a)
15         else
16             break ;                                          // Configuration (d)
17 push($Q$,{$F$, $rp$}) ;                                      // Enqueue current sample
18 if $rp =$ front($Q$).pos $+ l$ then
19     pop($Q$) ;                                               // Delete outdated value
20 if $rp \geq l$ then
21     return (front($Q$).val ) ;                               // Return opening sample
```

4.4.3 Arbitrary SE Orientation

The *streaming peak elimination* algorithm can be used for a 2-D input image support $D \subset \mathbb{Z}^2$, $D = [1, M] \times [1, N]$ as well. It only needs an image to be partitioned (mapped) into independent, 1 pixel thin discrete lines (called corridors) oriented in the same angle as the SE and computed according to Alg. 6. See Fig. 4.18 for examples of such partitioning. Because the input data arrive sequentially, and the computation takes place in one corridor at the time, each corridor does not need the whole instance of Alg. 6, but only queue Q for storage of intermediate results. So an array of queues AQ is used, and one instance of Alg. 6 uses all the queues in a circular order, see Alg. 8. The pixel at position $[j, i]$ (line j, column i) is mapped to p_{ji}-th queue of AQ, $Q_{ji} = AQ(p_{ji})$,

$$p_{ji} = \begin{cases} (i - j\tan(90 - \alpha))\mathrm{mod}(\#AQ) & \text{if } 45^\circ \leq \alpha \leq 135^\circ \\ (j - i\tan\alpha)\mathrm{mod}(\#AQ) & \text{otherwise} \end{cases} \tag{4-13}$$

where $\#AQ$ is a number of queues with upper-bound limit $N + L\cos(45^\circ)$. The set of corridors is a partition of the image support D; hence each pixel is read by the algorithm once and only once. The orientation also influences the definition of reading position such as

$$rp = \begin{cases} j & \text{if } 45^\circ \leq \alpha \leq 135^\circ \\ i & \text{otherwise.} \end{cases} \tag{4-14}$$

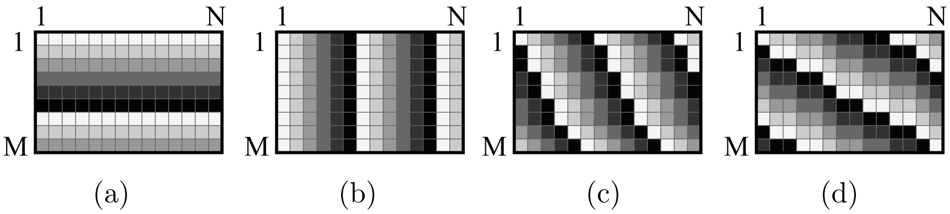


Figure 4.18: Image corridors (discrete lines) mapping for different SE orientations (a) a horizontal SE, (b) a vertical SE, and (c) and (d) inclined SEs.

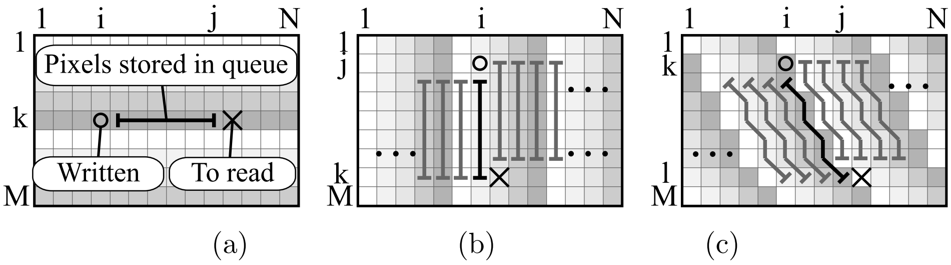


Figure 4.19: Image configuration for different SE orientations (a) a horizontal SE, (b) a vertical SE, and (c) an inclined SE. × denotes the pixels to be read in the next iteration, ∘ denotes the previous output pixels.

The main advantage of partitioning the image into a set of discrete lines, corridors, along which the 1-D algorithm computes, is a large overlap of two adjacent SEs (adjacent within a corridor), already discussed in Section 4.3.1. This property enables the use of an efficient 1-D algorithm such as *streaming peak elimination* to speed up the computation. On the other hand, resulting SEs are in general translation variant (but opening with these SEs remains idempotent). We have learned in the previous sections that the translation variance is a limiting property of dilation or erosion because for some B we are not necessarily able to find $\widehat{B}$, and therefore, construction of some operators (opening, filters, gradient, etc.) is impossible compromising thus the number of suitable applications.

The case of oriented 1-D opening is quite different. This operation is not often used together with another operation that needs the transposed SE $\widehat{B}$, which would be limiting. Indeed, the linear opening and the pattern spectrum are often used completely alone to measure the properties of image features such as the length or the orientation, see for instance equations (2-30), (2-31), or (2-32) from the introduction. For such purposes, the slight spatial variation of the SE does not bring any worse results than the translation invariant SEs by means of the obtained information.

Algorithm 8: $y \leftarrow$ OPENING_ORIENTED(f, l, N, M, α)

```
Input: $f$ - input image; $l$ - SE size; $N$ - image width; $M$ - image height
Result: $y$ - output image
Data: $AQ$ - Array of queue memories

init($AQ$) ;                                        // Initialize array of queues
if $45° \leq \alpha \leq 135$ then
    $steep \leftarrow$ true ;                         // Rather vertical orientation
else
    $steep \leftarrow$ false ;                        // Rather horizontal orientation
for $line = 1 : M + l\cos\alpha$ do
    for $column = 1 : N + l\sin\alpha$ do
        if $steep$ then
            $rp \leftarrow \min(line, M)$ ;             // Set reading position to line number
            $p_{ji} \leftarrow (column - line\tan(90° - \alpha))\mathrm{mod}(\#AQ)$ ; // Compute index
            of queue
        else
            $rp \leftarrow \min(column, N)$ ;  // Set reading position to column number
            $p_{ji} \leftarrow (line - column\tan\alpha)\mathrm{mod}(\#AQ)$ ;      // Compute index of
            queue
        if $read\ conditions$ then
            $F \leftarrow$ read($f$);            // Read the following pixel of input image
        $Y \leftarrow$ ONE_PASS_OPENING ($F$, $rp$, $l$, $AQ(p_{ji})$, $\alpha$) ;         // Call core
        function
        if $write\ conditions$ then
            $y \leftarrow$ write($Y$);                // Write result pixel to output image
```

Figure 4.19 illustrates the image configuration for different α. Let us focus on (a) that displays the horizontal configuration for processing line k. The algorithm

(Alg. 6) has just read pixel $[k{,}j]$, i.e., it has output pixel $[k{,}i]$. It will read $[k, j+1]$ and output $[k, i+1]$ in the next iteration. In the vertical case (b), the algorithm uses AQ, but only one Q is used at the time (marked by black color), others are put aside (gray color) until the computation proceeds to their column. The depicted configuration in Fig. 4.19 (b) corresponds to the time after processing $[k{,}i]$. The situation with the inclined direction is very similar to the vertical one except the corridors are inclined. Note that all pixels of the dataflow between $\times$ and $\circ$ are stored in the queues defining thus the system latency. Nevertheless, it is the minimal achievable and fixed operator latency considering unlike orientations of the SE and the scan order.

The final algorithm for oriented 1-D opening is outlined in Alg. 8 in a simplified version. At the beginning the steepness of the SE is determined. This variable affects which queue will be used for the current pixel as well as the sense of the rp increment. The algorithm works in a common double for loop over the whole image support, which is extended by the size of the SE for boundary handling. For each pixel of this extended support rp and p_{ji} are determined (line 8–13), new input pixel is read (line 15), the core function is called (line 16), the return value of which is written into the output image (line 18). The read and write conditions handle correctly support boundaries and vary with different orientation of the SE. More details on the algorithmic issues and a demonstration copy including source code of this algorithm can be found online at [Karas 2012a].

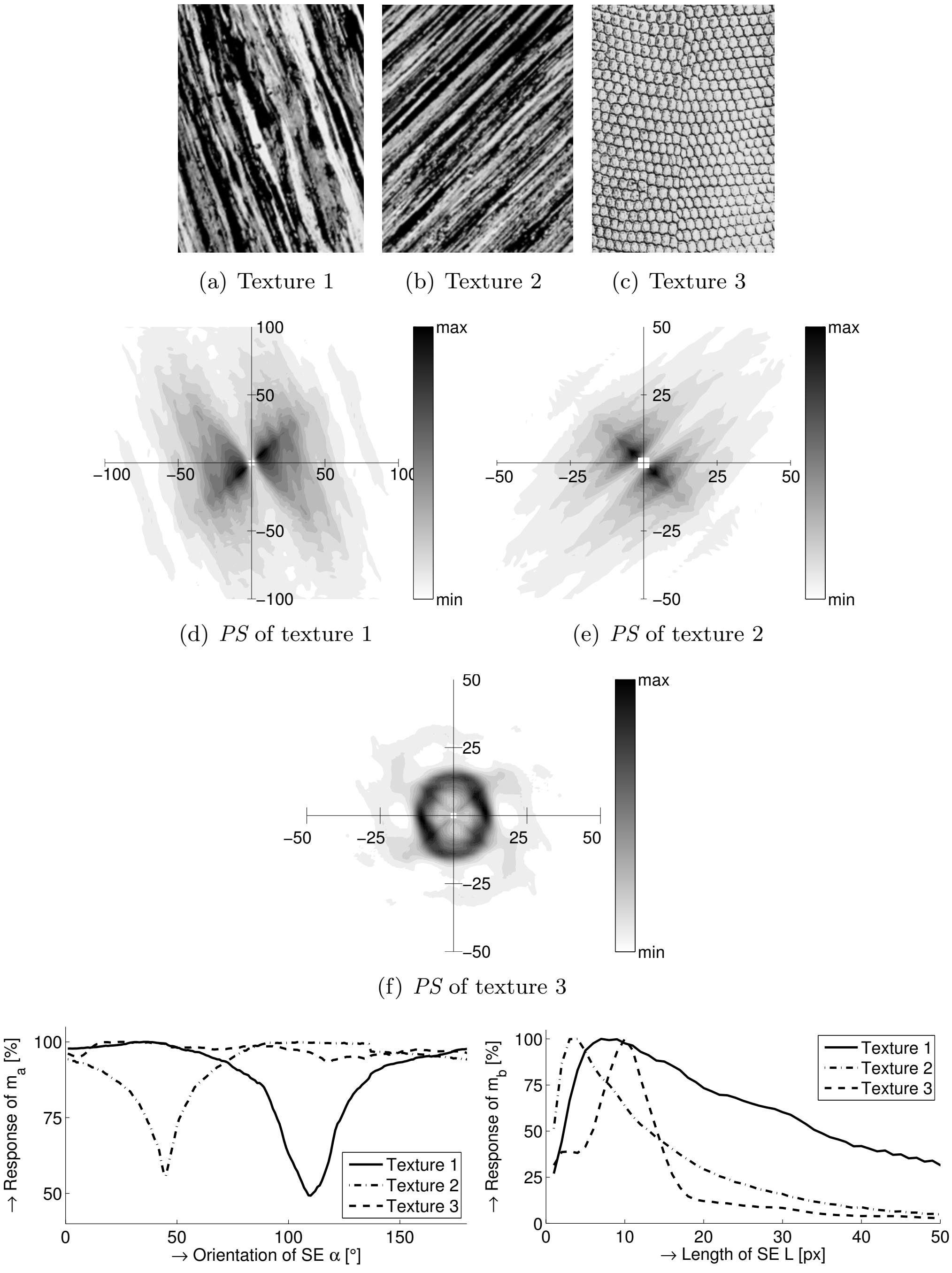
The proposed algorithm has very limited memory consumption. The only memory elements are queues whose depth and count are inferred by the SE orientation and image width N. The formulas are provided later in Section 5.5.

Examples of the pattern spectra for three different textures can be found in Fig. 4.20. Comparing the PS of textures 1 and 2 one can see the different orientation of the “butterfly-shaped” spectrum whereas the PS of texture 3 is “circle-shaped”, i.e., uniform in all the orientations. Such a circular PS suggests that the texture is rather isotropic, any non-symmetrical PS reveals anisotropic textures. We perform two measures on PS for better illustration of the texture analysis. The first measure reveals the influence of orientation by taking the sum over all the slice lengths such as

$$m_a(\alpha) = \sum_{\forall l_i < l_{\mathrm{MAX}}} PS(\alpha, l_i). \tag{4-15}$$

This measure is depicted in Fig. 4.20 (g) and can be used to determine the dominant orientation as $\alpha_{\mathrm{DOM}} = \arg\min_{\forall\alpha}(m_a)$. There are clear negative peaks at m_a for our textures 1 and 2 that measures their dominant orientation of 45° and 110°, respectively. The dominant angle of texture 3 is 118°, but as the m_a curve is flat, texture 3 is very isotropic and the dominant angle measure does not have much sense.

The second measurement m_b consider the thickness of the image objects regard-



(a) Texture 1 (b) Texture 2 (c) Texture 3

(d) *PS* of texture 1 (e) *PS* of texture 2

(f) *PS* of texture 3

(g) Measure m_a w.r.t. α (h) Measure m_b w.r.t. L

Figure 4.20: Pattern spectra of textures with different isotropy and their measurement. $\triangle\ \alpha = 1^\circ$, $L_{\mathrm{MAX}} = \{50, 100\}$.

less its orientation such as

$$m_b(l) = \sum_{\forall \alpha_i < \alpha_{\mathrm{MAX}}} PS(\alpha_i, l). \tag{4-16}$$

This second measure is plotted in Fig. 4.20 (h) and it shows different thickness of veins in textures 1 and 2 as well as the typical diameter of cells in texture 3.

4.4.4 Experimental Results of *Streaming Peak Elimination* Algorithm

We present essential CPU and GPU timing benchmarks of opening algorithms in this section. The main intention is to illustrate the computational complexity of the *streaming peak elimination* algorithm as well as to provide a comparison against a few other efficient opening algorithms, namely Soille *et al.* ([Soille 1996]), Van Droogenbroeck and Buckley ([Van Droogenbroeck 2005]), Urbach and Wilkinson ([Urbach 2008]), and Morard ([Morard 2011]). The benchmarks were performed on Intel Xeon E5620 @2.4GHz CPU running 64-bit Linux, and nVidia Tesla C2050 GPU with 14 MPs at 1.15 GHz and 3 GB RAM. The time reported in the tables below refers to the user time consumed by the respective algorithms.

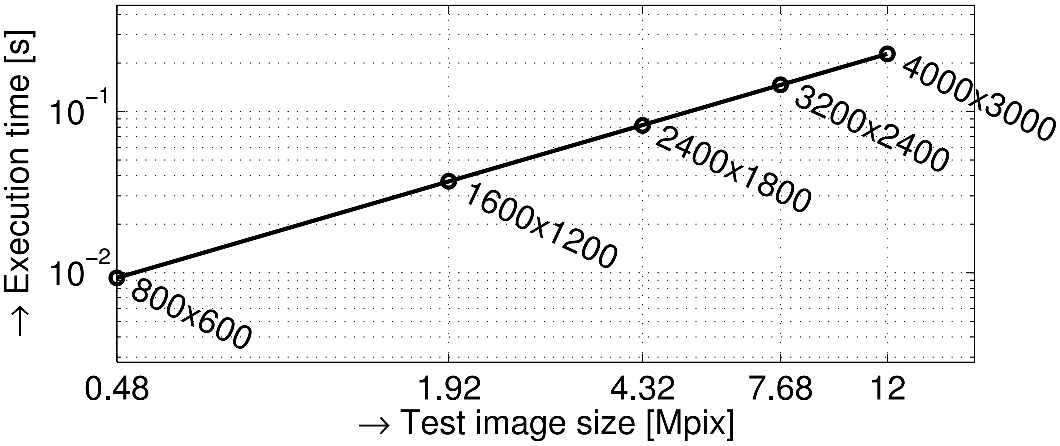


Figure 4.21: Execution time of opening/spectrum versus the image size. Structuring element is vertical 101 px.

CPU Implementation

At first, we evaluate the CPU execution time benchmark with respect to (shortened as w.r.t. hereafter) the image size, see Fig. 4.21. The results that lie on a perfectly straight line in the log-log scale illustrate the complexity of our algorithm is effectively independent of the image size. However, due to the bounded image support, the precise complexity per image is theoretically affected by the SE size ($W \times H$ denotes projected width and height of the SE) as $\mathcal{O}_{\text{image}}((N+W)(M+H))$. The reason for extra algorithm iterations is the operator latency. So when the last pixel of the image is processed, there is still approx. HN of pixels to be output. As these iterations do not read new input pixels, the peak elimination process is not involved, and consequently, they are faster then ordinary iterations. So the complexity $\mathcal{O}_{\text{image}}((N + W)(M + H))$ can be approximated as $\mathcal{O}_{\text{image}}(NM)$ provided $W \ll N$ and $H \ll M$.

The second benchmark in Fig. 4.22 retains the same image size and varies the length of the horizontal SE to show that the execution time is independent of the SE size (however the value of the execution time is dependent on the image content). The measured curve is practically constant (except a small decrease for short SEs) and approves thus the constant complexity $\mathcal{O}(1)$. Notice that the boundary effect takes place here as well, but it is compensated by some other effect, probably caused by optimization processes in a CPU. The same phenomenon can be observed for Morard and Van Droogenbroeck as well.

Observing the other algorithms, Van Droogenbroeck outperforms our solution by a non-negligible rate (3× in average). It is worth mentioning that this measure was obtained with the configuration quite favorable for Van Droogenbroeck: (i) the 8-bit gray-scale image (Van Droogenbroeck uses a histogram, which increases memory consumption and the computation time with a higher data precision), (ii) the horizontal orientation (Van Droogenbroeck is implemented for horizontal and vertical orientation only; although we assume the performance would not drop down drastically for the arbitrary orientation if the discrete-line partition method was used).

The Morard algorithm is also faster than ours, however the difference is less significant. One of the reasons may be the less homogeneous algorithm run that results in less comparisons and conditions per pixel, but infers non-regular access to data and larger memory requirements. However, the CPU can cope with the latter inconveniences quite well.

The complexity of Urbach and Wilkinson is of $\mathcal{O}(\lceil log_2(l) \rceil)$, so the computation time increases in steps every time the size of the SE l exceeds a power of two. Our solution becomes faster than this one for SEs longer than 64 px. The implementation by Soille is more than two times slower than ours. However, an opening in Soille's case is obtained by concatenation of dilation and erosion, so performing two complete image scans.

The experiment in Fig. 4.23 reveals an influence of the SE orientation angle α to the execution time. The proposed algorithm exhibits a small variation of execution time for the different octants. It is caused by the different spatial relation between corridors and horizontal scan that demands few additional conditions to be added for certain angles ($\alpha < 45°$ or $\alpha > 135°$). Also the border effects expressed in the complexity per image $\mathcal{O}_{\text{image}}((N + W)(M + H))$ very slightly affects the execution time. The border effect is strongest for $\alpha = \{45°, 135°\}$ and decreases for other angles up to purely horizontal and vertical orientation that has the minimal influence. The border effects also diminish with larger images verifying $W \ll N$ and $H \ll M$.

The Soille and Morard algorithms hold a constant value for the entire scale of orientation. The property that most likely allows for such consistency is random data access. They do not preserve the horizontal scan order for non-horizontal

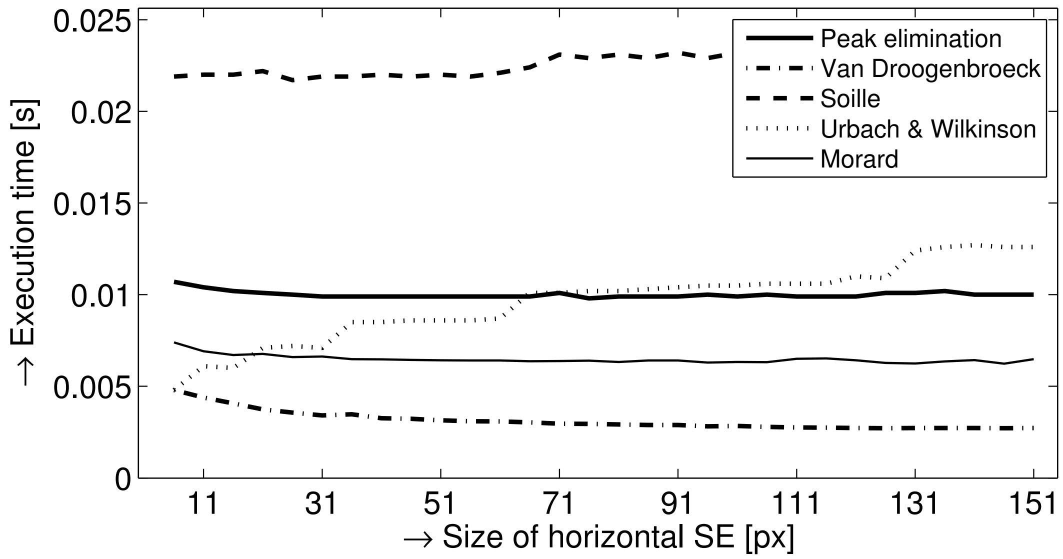


Figure 4.22: Execution time of opening versus the size of the horizontal structuring element. Natural 8-bit photo 800 × 600 px is used.

orientation as our algorithm, but scans the input image along the α-oriented discrete lines instead. Obviously, the resulting random access is not an issue on a CPU platform. The Urbach and Wilkinson algorithm results in much larger variation w.r.t. the orientation. The reason is hidden in the used decomposition of the SE into horizontal or vertical (whichever is better suited) chords that are computed using an efficient algorithm. As soon as dilation for these chords is computed, the results are naively compared together. And the naive computation of dilation from chords is the main bottleneck of this algorithm for arbitrary-oriented line SEs. One can see that for $\alpha = 0°$ the SE consists of only one long chord. While increasing α the number of horizontal chords is also increasing up to $\alpha = 45°$ when the SE decomposes into l chords of length 1 and the algorithm becomes completely naive. The further increase of α decreases the number of chords (now vertical). This decomposition, which can handle SEs of arbitrary shapes, results in triangular peaks in our case as in Fig. 4.23.

The last experiment in Table 4.1 reveals the influence of the image data precision to the execution time. The results suggest that although the performance is slightly worse for long integer and floating point data formats, the absolute difference is not significant. It is not surprising, the higher data precision only demands more memory for queues and higher bit-width of pixel data comparisons that is not a burden for CPU platforms either.

There are two additional thoughts to be pointed out to make our case more convincing. First, our algorithm computes not only opening, but also the pattern spectrum PS (so does Morard). Recall that the pattern spectrum requires a large number of openings when the traditional residual approach (2-29) is chosen. For example, let us consider the pattern spectrum for $l_{\mathrm{MAX}} = 100$. Even if we omit

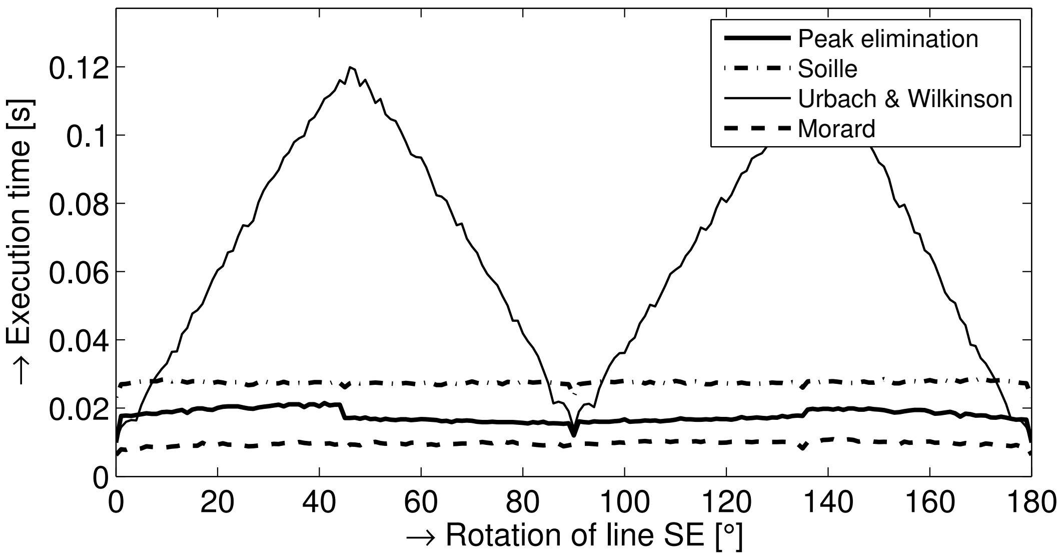


Figure 4.23: Execution time of opening versus the rotation angle α. Natural photo 800 × 600 px is used.

Table 4.1: Execution time of opening versus the data type. Structuring element is vertical segment 101 px; image size is 800 × 600 px.

Data type	char	short	int	long	float	double
Bit length	8	16	32	64	32	64
Execution time [ms]	10.7	10.5	10.6	11.6	11.3	11.9

the arithmetic operations, the pure time for computation l_{MAX} times γ_l will take 100×2.7 ms = 270 ms using the fastest Van Droogenbroeck's algorithm. Our algorithm computes the pattern spectrum in a single run, i.e., in 9.9 ms with speed-up 27×. Second, the CPU is not the only possible platform that takes advantage of efficient morphological algorithms. In the following sections we will consider GPU and FPGA as well.

GPU Implementation

In [Karas 2012b] we have implemented several state-of-the-art 1-D opening algorithms on a GPU and proved that *streaming peak elimination* opening algorithm is actually the best solution for a GPU platform achieving a high performance. Naturally, the algorithm designed to perform well on the dedicated hardware can be expected to bring good results on GPU platforms also, as both these platforms share some common requirements and have similar constraints. Recall that the beneficial properties of our algorithm are: sequential access to input/output data, minimal and fixed latency, small memory requirements. Let us break down how these properties influence the GPU implementation.

The sequential access to data has a positive impact on the performance. Even though the image is divided into many blocks of threads, 1 thread per column, scheduled by the GPU itself, and the blocks may be processed in arbitrary order, sequential access to data helps the threads within each block to be better synchronized, so-called coalesced. The thread synchronization is very important because the GPU achieves a large bandwidth to the global memory only when accesses are coalesced, that is multiple accesses can be handled by one memory operation. Accesses are coalesced if their addresses are consecutive, for example i-th thread accesses address $A_i = A_0 + i \times \text{sizeof}(\mathit{float})$.

The fixed latency is another factor that helps the scheduler to keep the threads within a block synchronized. Alternatively, if the latency of two threads was not fixed but variable, memory accesses at either reading or writing side could not be coalesced.

The memory requirements property is not very significant by means of the amount of necessary memory to allocate because modern GPUs contain large global memories of size in orders of GBs. However, limited memory requirements allow us to better use a shared memory; a small local memory that is much faster than the global memory and does not need coalesced accesses. Further aspects of GPU programming and memory hierarchy can be learned from [nVidia 2012].

In the aforementioned publication [Karas 2012b] we exploited all these advantageous properties and provided the GPU library for arbitrary-oriented opening available online in [Karas 2012a]. According to the inherent structure of GPUs, a few optimization techniques were developed to provide better and more consistent performance for different image sizes and orientation. The optimization includes an enhancement of parallelism by splitting the image into smaller partitions, a partial use of the shared memory, and rotating the image when beneficial.

The first benchmark of the GPU opening algorithm w.r.t. the image size is displayed in Fig. 4.24. The size of the SE is approximately equal to 5% of the image width. The performance of *streaming peak elimination* algorithm increases with the image size because larger image contains more threads that allow the scheduler to better cope with the global memory latency, cache misses, etc. There is a simple rule of thumb that more threads bring better performance until a sufficiently large number of threads is achieved, for which the performance saturates. The same performance growth is also observed for the Morard algorithm.

The OpenCV 2.2 implementation does not follow this trend of the performance growth, but on contrary it drops down. The reason is that OpenCV implements the morphological operators in a less efficient way (of complexity $\mathcal{O}(l)$, l is length of the SE); therefore, the larger image using proportionally larger SE implies lower performance.

In the second benchmark in Fig. 4.25, we display the impact of variable SE size on our algorithm. In contrary to the CPU when the performance was nearly constant, we can observe a slight decrease of the throughput w.r.t. the SE size. This phenomenon can be explained by the use of shared memory, the amount of

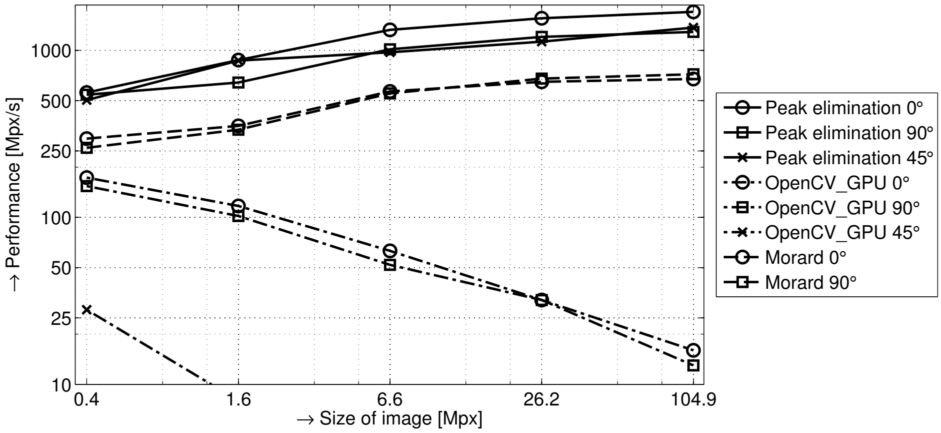


Figure 4.24: Performance of GPU opening versus the image size, SE is approx. 5% of the image width. We used D15 texture from [T. Randen 2012].

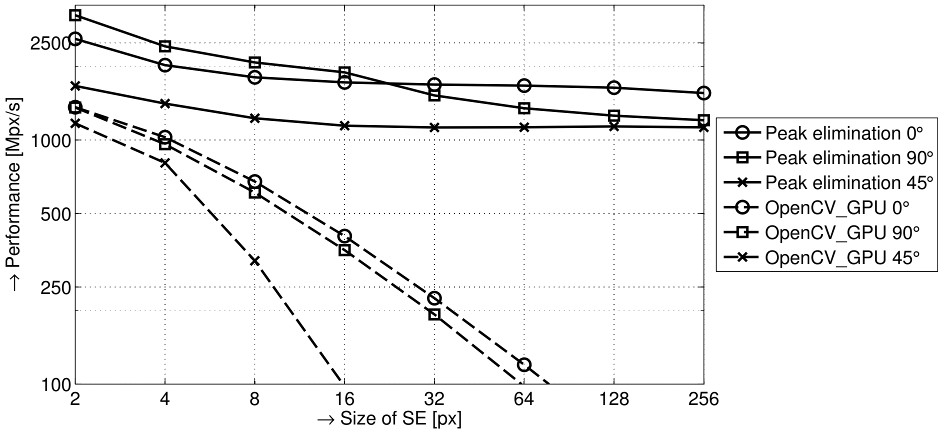


Figure 4.25: Performance of GPU opening versus SE size, image [T. Randen 2012] D15 5120×5120

which is limited. So for larger SEs when the size of queue exceeds the per-thread dedicated amount of shared memory, the slower global memory must be used. On the other hand, the performance saturates on a value exceeding 1000 Mpx/s for arbitrarily long SEs.

4.5 Conclusions

We have discussed the *Dokládal* dilation algorithm, its properties, and how instances of this algorithm can be concatenated in order to compose 2-D SEs, either rectangular or polygonal. Even for 2-D SEs, it preserves the sequential access to data and small latency, the properties that are very favorable to the hardware implementation. In addition, the benchmark-verified constant complexity $\mathcal{O}(1)$ suggests that the prospective implementation shall be efficient for a large scale of SE sizes, which is an important intention of our contribution.

As for the *streaming peak elimination* algorithm, the main principles of the algorithm and how it can make use of a queue memory were described. Then we enriched the algorithm with the capability of obtaining the pattern spectrum at effectively no additional cost. The direct computation of the pattern spectrum conveys a significant speed-up against a conventional approach using residue of opening. The algorithm also supports arbitrary angle orientation of the SE that enlarges the family of possible applications by those involving the notion of orientation, such as thin feature enhancement, local orientation measure, or oriented spectrum.

The algorithm retains the sequential access to data and minimal latency regardless the orientation of the SE. The importance of the considered sequential access was also supported by the GPU implementation that achieves the best results among several GPU solutions partly thanks to the sequential access. Furthermore, it computes opening and a pattern spectrum in $\mathcal{O}(1)$, that is independently of the SE size. Taking into account the implementation on a GPU and an FPGA, the proposed opening and pattern spectrum algorithm is an important contribution to the state of the art. Even though the CPU performance is only moderate, the algorithm achieves the highest performance to date using a GPU. All these properties suggest that the *streaming peak elimination* algorithm should suit well the dedicated hardware platform and allow for an efficient implementation, especially for large SEs.

5 Hardware Implementation

Contents

This chapter deals with the hardware implementation of basic morphological operators, which constitutes one of the main contributions of the thesis. We already know that most of the previous proposals followed the naive computation taking advantage of the massive parallelism in the dedicated hardware, especially FPGAs. However, while increasing the SE size, the naive approach becomes eventually inefficient by means of performance or hardware resources.

We, on the other hand, have decided to use efficient algorithms $\mathcal{O}(1)$ that are expected to allow for an implementation that retains its efficiency for various sizes and shapes of SEs. From all $\mathcal{O}(1)$ algorithms, the *Dokládal* dilation algorithm and the *streaming peak elimination* opening algorithm showed the most convenient properties for the dedicated hardware. The description is structured as follows.

First, we focus on the implementation of the *Dokládal* algorithm that forms a basic programmable 1-D block supporting either horizontal and vertical SEs (published in [Bartovský 2010]). Thanks to the separability, it can be used as a building brick in concatenations of any length. We illustrate this *inter-operator parallelism* below on the 2-D rectangular dilation. Then, we introduce the further *intra-operator parallelism* that almost linearly increases the performance of rectangular SEs (in [Bartovský 2011b]).

Second, we extend the basic 1-D block to compute inclined SEs and create a pipeline concatenation of such blocks that provides a polygonal SE within a single image scan. The *intra-operator parallelism* is also employed (in [Bartovský 2012b]).

Last, we implement *streaming peak elimination* algorithm and provide a programmable block supporting the oriented opening and pattern spectrum. This block can take advantage of *inter-operator parallelism* to speed up computation of the applications that includes multiple opening or pattern spectrum operations (in [Bartovský 2012c]).

5.1 1-D Dilation Architecture

At first glance at the 1-D *Dokládal* algorithm presented in Section 4.1, it is clear that it comprises an inherent sequential behavior. It contains a *while* loop with an a priori unknown number of iterations that can not be unrolled. The common way to implement such a system is the Mealy Finite State Machine (FSM) [Mealy 1955]. The FSM is generally an abstract machine that can be in one of a finite number of states. It changes the state upon some triggering condition called a transition. During a transition one command or a signal can be issued, used for various purposes (typically control of some underlying system).

In our implementation the FSM manages the algorithm behavior, controls the input and output dataflow, and issues all the FIFO operations to the queue. It consists of 2 main states $\{S1, S2\}$, see Fig. 5.1 for state diagram.

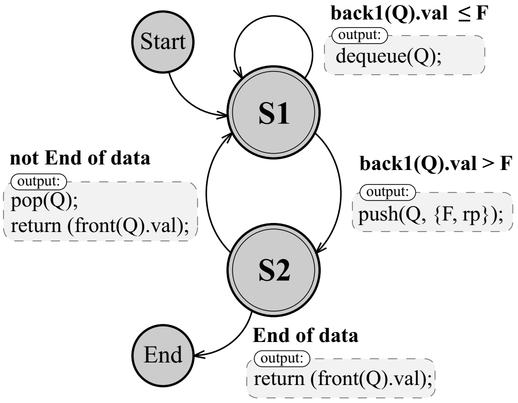


Figure 5.1: State diagram of the 1-D algorithm FSM. State transition conditions are typed in bold; the output signals are given in gray rectangles.

The $S1$ state *dequeues useless values* of Alg. 2 on page 2. It is a data-dependent stage of the algorithm as it dequeues an a priory unknown number of pixels. This is represented in the code by the *while* statement (code line 1). Consequently, its computation time varies from 1 to l clock cycles (l denotes the length of the SE) in the worst case when all the previously stored pixels are useless. The *enqueue current sample* operation (code line 3) is issued upon the transition from $S1$ to $S2$.

The $S2$ state handles code lines 4 and 5, *delete too old values*, and the lines 6 to 9 *return valid value* or *return empty.* These instructions are independent and simply executed in parallel. Consequently, the execution of $S2$ takes only one clock cycle. In the following we will describe architectures supporting horizontal and vertical SEs separately.

5.1.1 Horizontal Architecture

The hardware implementation for horizontal SEs is divided into 2 areas (Fig. 5.2), the FSM part and the memory part. The FSM manages the entire computing procedure and temporarily stores values in the memory part. The memory part contains one queue implemented in the dual-port RAM memory. The Control unit is a sequential circuit that manages the state transitions. It increments the *rp*, *wp* and manages the *rp* counter appropriately. The Control unit also performs the queue memory operations and handles the backward full flag used for a dataflow control.

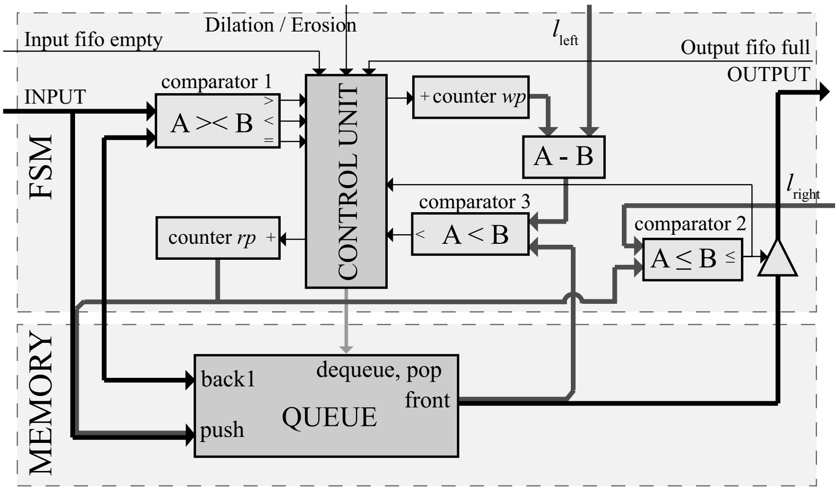


Figure 5.2: Overview of the 1-D horizontal architecture. The FSM part manages computation, the memory part contains one queue.

Principle

In the beginning of $S1$, the last queued pixel is invoked by the back() operation from the queue and fetched to Comparator 1 where it is compared with the current sample. The Control unit decides on the basis of comparison results and selected morphological function (dilation or erosion) whether the enqueued pixel is to be dequeued (lines 1-2). Otherwise, the current pixel is extended with the reading position *rp* and enqueued (line 3).

The $S2$ invokes the oldest queued pair $\{F, rp\}$ by front() operation. The read pixel is a correct result if *rp* has already reached or exceeded the l_{right} parameter.

This output condition (line 6) is checked by Comparator 2. The deletion of outdated values is performed by comparing the reading position of the oldest pair with the current *wp* minus the l_{left} length in Comparator 3. Notice that the deletion has no impact on the output dilation value because the pop() operation (lines 4 and 5) issued by the Control unit has effect only with the next clock edge.

The entire set of parameters, i.e., SE dimensions and selection of the morphological function, is run-time programmable. These parameters have to be set before processing a frame. In addition, no further controller is needed; the internal behavior is driven only by the regular scan order dataflow. Such a processing unit is sometimes called data-stream-driven machine as the incoming dataflow triggers the computation.

5.1.2 Vertical Architecture

The architecture supporting vertical SEs stems from the horizontal one and is depicted in Fig 5.3. The main distinction is that the vertical architecture does not have in the memory part only one queue but N queues called Array of queues AQ in Section 4.2.1. As the algorithm uses only i-th queue $AQ(i)$ in the i-th column, the whole AQ can be considered as a set of independent, parallel queues that can be addressed by one variable called *page*. Then keeping *page* equal to the column coordinate will handle the appropriate switching of queues such that the pixels in each column are not mixed up with pixels in other columns. That conforms to the desired column-by-column image partitioning. Incrementing *page* variable is carried out by the *page* counter that is incremented with every new pixel and reset at the end of the image line.

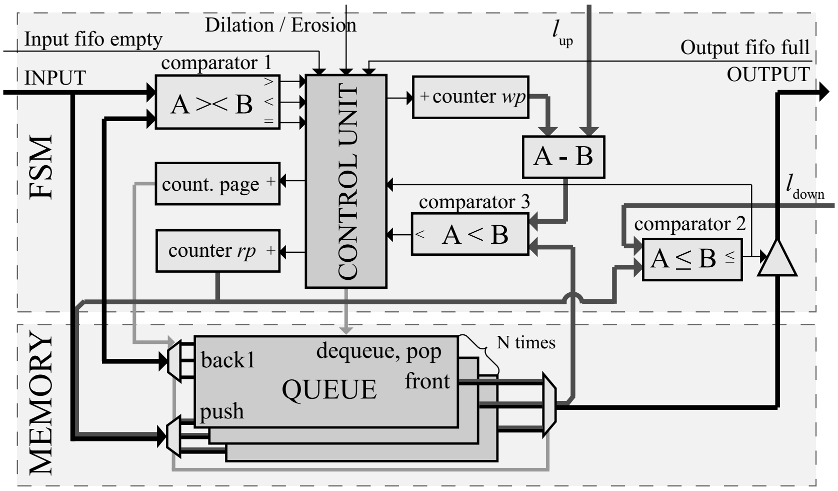


Figure 5.3: Overview of the 1-D vertical architecture. The FSM part manages computation, the memory part contains N queues called Array of queues.

5.1.3 Reducing the Impact of Data Dependency

Hereafter, we briefly describe two techniques brought to the system to obtain a higher throughput and lesser area occupation. This optimization affects both architectures described above, however, the impact is more significant for the more complex vertical architecture.

Number of Dequeue Steps

The data dependent number of dequeue steps (below denoted by *Steps*) has unpleasant consequences on the hardware design: longer balancing FIFOs (see Fig. 5.6), lower data throughput. The number of pixels stored in the queue is within the interval $[1, l)$. For a hardware design it is important to minimize the worst-case upper bound $\mathit{Steps}_{\mathrm{orig}}{=}l-1$. The worst case appears when the queue is full and the current input pixel is greater then all the queued pixel, which therefore have to be erased at once.

Suppose that we are about to dequeue D pixels at a given moment. We know that the pixels are queued in a strictly decreasing order. Thus, if the DL-th pixel ($DL < D$) can be dequeued, all previous pixels can be dequeued, too. This can be done in a single atomic operation. The number of dequeue steps is then given by

$$\mathit{Steps} = D \operatorname{div} DL + D \operatorname{mod} DL, \tag{5-1}$$

where div and mod denotes the integer division and the remainder operations. The term $D \operatorname{div} DL$ represents the number of *large* dequeue steps (DL pixels at the time) and $D \operatorname{mod} DL$ the number of ordinary dequeue steps (1 pixel) for given values of D and DL. For example, let $D = 30$ and $DL = 7$. *Steps* is then equal to $30 \operatorname{div} 7 = 4$ *large* steps of 7 pixels each and $30 \operatorname{mod} 7 = 2$ ordinary steps.

When searching for the worst case, we have to examine all the possible values of D for the given length of the SE l, such as $D{\in}[1, l)$. The worst-case number of dequeue steps reduces to

$$\mathit{Steps}_{\mathrm{WC}} = \max_{D<l}(D \operatorname{div} DL + D \operatorname{mod} DL). \tag{5-2}$$

For example let $l = 31$ and $DL = 7$. When we set $D = l - 1 = 30$, *Steps* is equal to 6 as computed above. However, what if only 27 pixels are to be dequeued? Then $\mathit{Steps} = 27 \operatorname{div} 7 + 27 \operatorname{mod} 7 = 3 + 6 = 9$ is actually worse than the former case of 30 pixels, and proves that the search over all $D < l$ is necessary.

In order to find the optimal worst-case number of dequeue steps $\mathit{Steps}_{\mathrm{optim}}$, we minimize the worst case (5-2) over all possible DL values such as

$$\mathit{Steps}_{\mathrm{optim}} = \min_{DL<D} \max_{D<l}(D \operatorname{div} DL + D \operatorname{mod} DL). \tag{5-3}$$

The argument of the DL minimization gives us the optimal value DL_{optim} that ensures the minimal number of the worst-case dequeue steps such as

$$DL_{\mathrm{optim}} = \arg \min_{DL<D} \max_{D<l}(D \operatorname{div} DL + D \operatorname{mod} DL). \tag{5-4}$$

Table 5.1 presents the original and optimal number of dequeue steps for some SE sizes. Notice that more than one DL_{optim} can exist. The DL is also a programmable parameter of the proposed architecture so its value can be modified when different l is used.

Table 5.1: Optimal dequeue length, original and reduced number of dequeue steps for selected SE sizes

SE size l	3	11	21	31	41
$Steps_{\text{orig}}$	2	10	20	30	40
DL_{optim}	2	3, 4	4, 5, 6	5, 6, 7	6, 7
$Steps_{\text{optim}}$	2	4	7	9	10

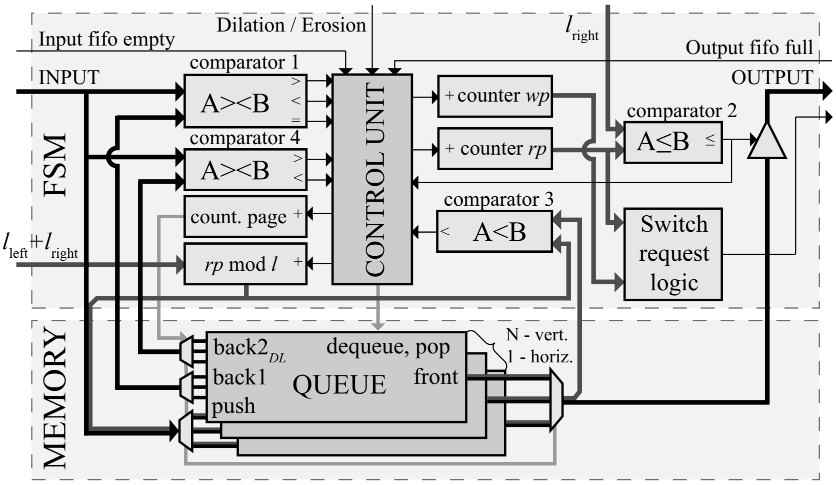


Figure 5.4: Overview of the optimized 1-D vertical architecture. Comparator 4 is added for DL dequeuing, and deleting outdated values is simplified.

The DL dequeuing needs DL-th pixel from the back to be fetched from the queue, so the queue is extended by the back2$_{DL}$ operation for this purpose, see Fig 5.4. The retrieved value is then compared against the current input pixel, and the Control unit dequeues either DL pixels, one pixel, or none depending on the outcome of comparators. Notice that the added memory port and comparator do not increase a number of cycles to process a pixel as the dedicated hardware performs the respective operations in parallel (the queue is implemented in a dual-port memory, so reading two values on different addresses at the same time is possible).

The DL dequeuing significantly decreases the number of worst-case dequeuing steps, e.g. for $l = 31$, from $Steps_{\text{orig}} = 30$ to $Steps_{\text{optim}} = 9$. This optimization is convenient for hardware implementation as the processing is more regular and the balancing FIFOs may be smaller, see (5-5) (5-6) below.

Pixel Addressing

The absolute pixel addressing in the queues can be advantageously replaced in the hardware by using the modulo addressing. Hence, instead of the absolute reading position rp, we use the relative modulo position $rp \bmod l$. The pixels are enqueued by push$(Q, \{f, rp \bmod l\})$ (code line 3).

The delete condition of line 4 changes accordingly. Using the modulo addressing, a stored pixel becomes outdated whenever its modulo address equals the current pixels's one ($rp \bmod l =$front(Q).val).

The advantage of the modulo addressing is a smaller data width. It fits into $\lceil \log_2(l-1) \rceil$ bits, whereas the absolute addressing requires $\lceil \log_2(N-1) \rceil$ bits. This is mainly advantageous for vertical orientation which needs N queues per a unit.

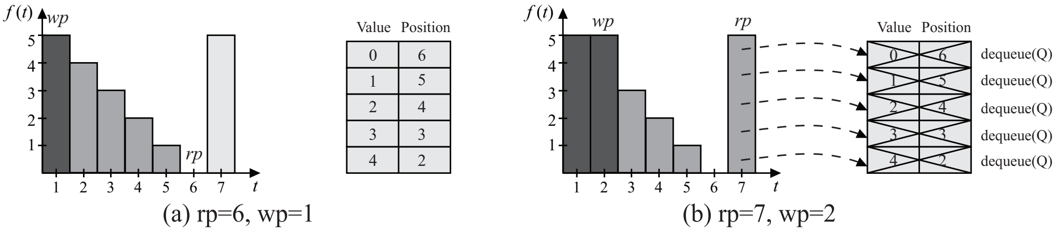
5.2 2-D Rectangular Dilation Architecture

As we know, dilation is separable into lower dimensions, e.g., $\delta_R(f) = \delta_{H \oplus V}(f) = \delta_V(\delta_H(f))$, see (4-2). The dilation by a rectangle can be implemented as a concatenation of two 1-D dilation blocks working with horizontal and vertical SEs. Before connecting two processing blocks into a pipeline we study their latency and processing pixel rate.

Considering the *operator latency*, the latency introduced by the dependence on the future data samples, the horizontal architecture shows latency of l_{right} pixels and the vertical architecture $l_{\text{down}}N$ pixels (recall N denotes the image width). The value of this latency is highly dependent on the size of the SE, and it is given in terms of pixel-delay between input and output dataflows.

The *computing latency* measures the number of clock cycles needed for processing one pixel. In both horizontal and vertical cases its value is highly dependent on image contents, it varies from pixel to pixel. This latency consists of 1 to $1 + \mathit{Steps}_{\text{optim}}$ iterations of the FSM state $S1$ and 1 iteration of $S2$. The best-case computing latency of 2 clock cycles happens when there is no pixels to dequeue (constant or decreasing signal, empty queue, etc.). The worst case arrives whenever a monotonously decreasing signal is followed by a value greater than the previous signal, see Fig. 5.5 for an example. The pixels of the decreasing interval have to be stored in the queue for a possible future use in the computation filling up the queue in Fig. 5.5 (a). When a high value arrives in Fig. 5.5 (b) all pixels stored in the queue have to be discarded at once. As an a priory unknown number of pixels may be dequeued, the computing latency varies in the interval $[2, 2 + \mathit{Steps}_{\text{optim}}]$.

The average pixel rate denotes the number of clock cycles needed per pixel in average and it stays in the interval from 2 clock cycles per pixel for the best-case image (e.g., constant image), up to 3 clock cycles per pixel for the worst-case image (it is not a random noise but saw-shaped signal as shown in Fig. 5.5). The best-case 2 cycles per pixels (one $S1$ and one $S2$) takes place when no pixels are dequeued, the worst-case 3 cycles per pixel arrives when all the processed pixels are dequeued



(a) rp=6, wp=1

(b) rp=7, wp=2

Figure 5.5: Illustration of the worst-case computing latency for an anti-causal SE $l = 6$. (a) The queue is full after processing the decreasing signal, (b) the following high value causes that the queue has to be emptied at once.

(it adds one cycle per pixel in average). The current rate between 2 and 3 clock cycles per pixel depends on the image contents.

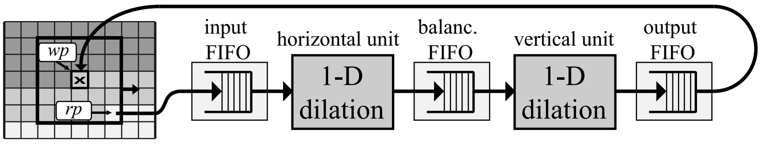


Figure 5.6: 2-D implementation is composed of two 1-D blocks, one for each direction.

In order to concatenate the processing units with different and varying latency and processing rates, we need to use some coupling elements to balance the difference. The most common solution is insertion of a FIFO memory as shown in Fig 5.6. The depth of this FIFO directly defines the upper bound of the system latency of the 2-D block and increases memory requirements. In cases when preserving continuity of the input/output dataflows is crucial, that is the dataflows are not interruptible and the processing must be stall-free, the necessary depths of input and balancing FIFOs can be computed from the dequeuing worst case and the stream rate (a number of clock cycles per pixel) as follows

$$F_{\text{input}} = \frac{Steps_{\text{optim}} + 2}{StreamRate} - 1 \tag{5-5}$$

$$F_{\text{balance}} = N\left(\frac{Steps_{\text{optim}} + 2}{StreamRate} - 1\right). \tag{5-6}$$

The output FIFO ensures a permanent stream delay in all circumstances. Its maximal size is a sum of both FIFOs (input and balancing). The instantaneous filling of output FIFO is complementary to the filling of both FIFOs combined. The overall delay does not change. If more 2-D blocks are pipelined to form compound operators (e.g., opening, closing, ASF), only one output FIFO at the end is necessary.

The output and balancing FIFOs can be merged (see Fig. 5.7) into one memory thanks to the following properties: i) the vertical unit reads exactly one pixel from

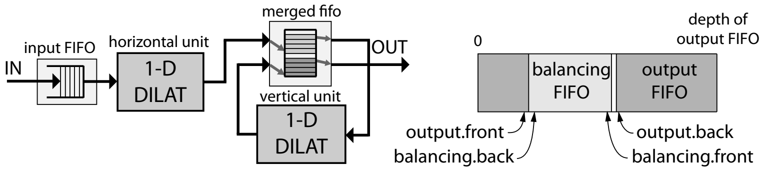


Figure 5.7: Merged FIFO replaces the balancing and output FIFOs to reduce memory requirements.

the balancing FIFO for each pixel written to the output FIFO. Consequently, filling of these two FIFOs is complementary; the occupied memory spaces can not collide with each other, ii) the read/write activity is at most 1 access per 2 clock cycles. Hence, reading ports of both FIFOs can use one memory port and the writing ports can use the other memory port (without overloading). Merging both FIFOs reduces the memory to approximately one half. The result merged memory (see Fig. 5.7) has two pairs of standard FIFO ports, but it contains only one dual-port RAM.

Memory Requirements

The memory requirements of the 2-D architecture consist of horizontal and vertical computation-involved queues and two balancing FIFOs, defined by (5-5) and (5-6). The size of the queue is equal to the upper bound of the programmable SE size.

In the vertical case, the algorithm uses many queues. Instantiating N separated memories would be resource inefficient because the FPGA RAM blocks could not be exploited. Instead, these queues are gathered in a single dual-port memory (see Fig. 5.8) since only one queue is accessed at the time (the others are idle). A single memory block also allows usage of either on-chip block memory or off-chip memory.

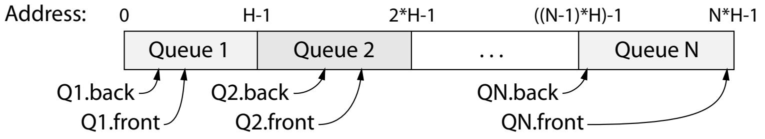


Figure 5.8: Vertical queues are mapped into linear memory space side by side. The front and back pointers are stored in a separated memory.

Every queue has a related pair of front and back pointers which must be retained throughout the entire computation process. The appropriate pair is always read before the particular queue is used and the result pointers are stored back after the computation left the queue. These pointers are stored in a separated pointer memory. The queues are efficiently packed into RAM blocks resulting in a small memory extension.

Let W×H denote the width×height of the rectangular SE, and bpp bits per pixel. The memory requirements per a 2-D unit is given by:

$$M_{\text{hor}} = W(bpp + \lceil \log_2(W-1) \rceil) \qquad \text{[bits]} \qquad (5\text{-}7)$$

$$M_{\text{ver}} = N(H(bpp + \lceil \log_2(H-1) \rceil) + 2\lceil \log_2(H-1) \rceil) \qquad \text{[bits]} \qquad (5\text{-}8)$$

The following example illustrates very low memory consumption achieved thanks to the stream processing. Neither the input, output, nor any intermediate image is buffered.

Example: Consider a dilation of 8-bit SVGA image (i.e., 800×600=$N \times M$) by a square, 31x31 SE. The computation (the queues) requires (5-7) and (5-8)

$$M_{\text{hor}} = 31(8+5) = 403 \text{ bits}$$

and

$$M_{\text{ver}} = 800(31(8+5) + 2 \times 5) \approx 330.4 \text{ kbits}$$

resulting in a total of 331 kbits for the 2-D dilation.

The input and the balancing FIFOs for stall-free stream processing require (5-5) and (5-6)

$$F_{\text{input}} + F_{\text{balance}} = (N+1)\left(\frac{Steps+2}{StreamRate} - 1\right) 8\text{bit} =$$
$$= (800+1)\left(\frac{9+2}{3} - 1\right) 8\text{bit} \approx 17 \text{ kbits}$$

The total memory needed to implement the 2-D dilation is 331+17=349 kbits. This is far below the raw size of the image itself $800 \times 600 \times 8$bpp$\approx$ 3.84 Mbits which does not need to be stored.

These low memory requirements allow to fit the architecture to small devices supporting large images. It should be pointed out that memory requirements are larger for landscape images ($N > M$) than for portrait images ($N < M$). Another important aspect is that the architecture can be dimensioned to different image size by only changing the size of the memory, which can be placed out of the chip.

5.2.1 Parallel Rectangle Architecture

This section develops and implements the concept of *intra-operator parallelism* in the dilation/erosion operator. Its main objective is to increase the pixel rate while maintaining the beneficial properties of the proposed algorithm, namely sequential access to data and minimal latency as much as possible. We know that the pixel rate of the rectangle architecture is in the interval from 2 to 3 clock cycles per pixel. Such a throughput may not be sufficient for the most demanding application targeted to high definition images.

The principle of parallelization is based on utilization of concurrently working units that process different parts of the image simultaneously. The number of units used in parallel for horizontal and vertical directions defines the parallelism degree (PD). In the ideal case with a zero overhead, the computation should be sped up PD times.

Using PD units of each kind in parallel needs us to determine which part of the image will be processed by each unit, such that each pixel is assign to one unit of each kind. This process is commonly called partitioning. Considering that the input data are fetched line by line, we propose a solution minimizing the waiting-for-data periods of all units.

The partition of the image among horizontal units is interleaved line-by-line, such that i-th line is processed by $\{((i-1) \bmod PD) + 1\}$-th horizontal unit. The vertical units use the partition into compact blocks of height M and width N/PD (if $N \bmod PD \neq 0$, some of the blocks may be 1 pixel wider than the others). The image partition for 2-D dilation conforms to the intersection of the two partitions (Fig. 5.9). Its granularity is determined by PD.

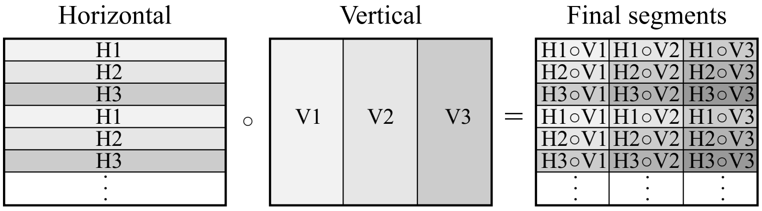
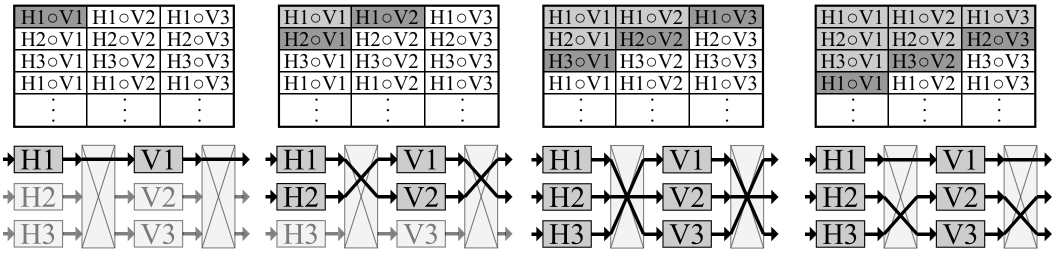


Figure 5.9: Example of image partition for PD =3: image is divided horizontally line by line and into PD equal stripes in a vertical direction. The final image partition is obtained by intersection.

During the parallel processing, the computation runs simultaneously at multiple segments of the image, see Fig. 5.10. These segments must belong to different columns and lines, i.e., must be placed on a diagonal. It is given by the fact that one unit can compute only one segment at the time, so e.g., segments H1∘V1 and H2∘V1 can not be processed at the same time because V1 can not compute two segments simultaneously. Also all the units computes segments in the order of the horizontal data scan.



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

Figure 5.10: Image partitioning and switch routing in parallel processing for PD =3. Decomposed in time: (a) beginning of processing, (b)–(d) after kN pixels, k=1..3. The shading denotes the state: Dark gray - being computed, light gray - already computed, white - waiting.

The input data rate can be theoretically PD-times faster than the computational throughput of the unit. Therefore, each image line needs to be buffered in a line buffer, so there are PD input line buffers and PD output line buffers. According to the partitioning, the i-th buffer stores $\{i+(k-1)PD\}$-th image line. The input line buffers are filled at the external, fast pixel rate and read by the internal PD-times slower rate.

Figure 5.10 gives an example for $PD = 3$, hence we have three horizontal (H1–H3) and tree vertical (V1–V3) processing units. As soon as the line buffer receives the first pixel, the first horizontal unit H1 starts the processing and feeds results to the first vertical unit V1. Its output is fed to the first output line, see Fig. 5.10(a). After N received pixels, the output of H1 is connected to V2 which is connected to output line 1. Since the H1 left V1 and line 2 is already being read, the H2 can start processing second line feeding V1 connected to output line 2, see Fig. 5.10(b). As soon as $2N$ input pixels have been received, the H1 connects to V3, H2 connects to V2 and H3 connects to V1, see Fig. 5.10(c), and so on.

The parallel architecture depicted in Fig. 5.11 contains four separable generic parts scalable by PD: input buffer, horizontal and vertical parts and output buffer. The input buffer is mainly composed of 1-to-PD multiplexer and PD line buffers. It divides the fast input stream into PD (PD-times slower) streams processed by computation units as the i-th image line is connected to the $\{((i-1) \bmod PD)+1\}$-th line buffer. The output buffer merges PD slow streams of the processed data into a single fast output stream respecting the image horizontal scan order. The operator blocks can be concatenated into more complex functions (opening, closing, ASF, etc.), the buffers are used only at the beginning and at the end of the chain.

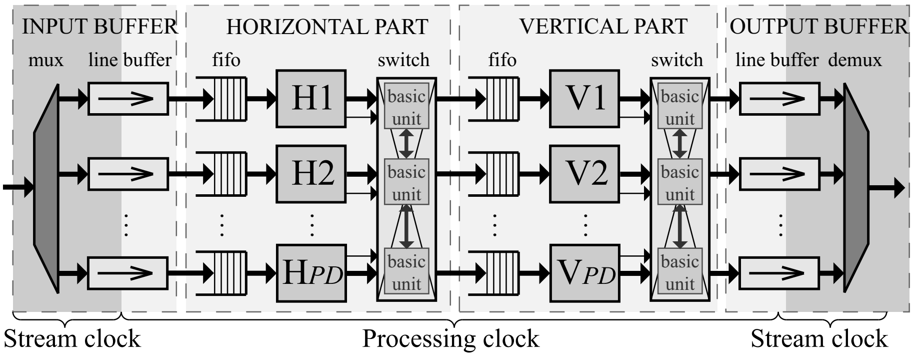


Figure 5.11: Overview parallel 2-D architecture. The horizontal and vertical stages can be instantiated several times between input/output buffers to create compound operators.

Both horizontal and vertical parts instantiate PD balancing FIFOs, PD horizontal or vertical units, and one switch that manages the interconnection. Each horizontal unit along with the front-end FIFO conforms to Section 5.1.

The width of segments proportionally affects both vertical memories, see (5-6) and (5-8). The area of every horizontal unit remains unchanged, since every unit processes the entire line. The overall memory of the horizontal part is a factor of PD. Contrarily, the memory requirements of every vertical part is divided by PD because it processes only a fraction of the original image width.

Switching

The routing of the computation units is handled by a switch block. Every switch contains *PD* input ports from previous units and the same number of output ports linked to the subsequent units. The purpose of the switch is to manage up to *PD* interconnection channels. Notice that they are bidirectional: forward data and backward FIFO full flag. As described in Fig. 5.10, the output switching of all input ports is circular, i.e., V1, V2 ... V*PD*, V1, V2, ... and so forth. This property makes the switching easier because the only condition to evaluate is when to switch and whether the requested output resource is available; the destination port is given by the sequence. The switching moment is provided by the preceding unit switch request logic which generates a switch request impulse every time it crosses the border of adjacent segments.

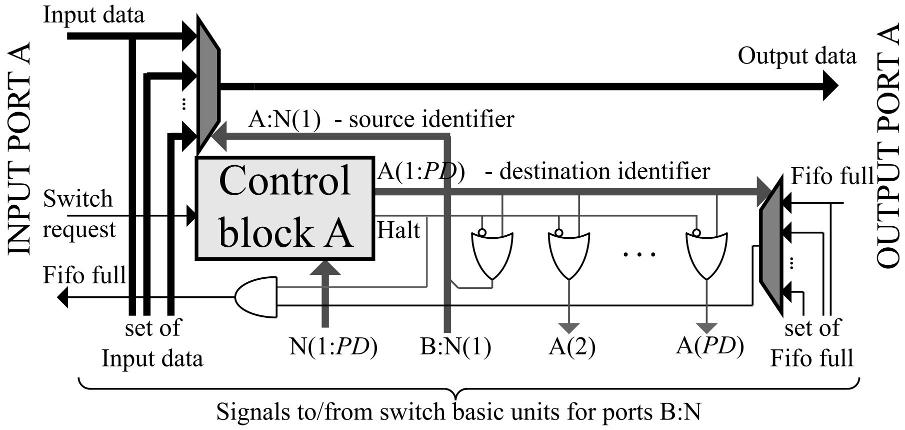


Figure 5.12: Basic unit of the switch. Every switch contains n basic units for a correct routing between n input/output ports.

Figure 5.12 depicts the basic unit of the switch for one pair of input/output ports referred to as A. For *PD* pairs of ports this circuitry is instantiated *PD*-times. Each input port possesses a related control unit block that manages all channel transitions considering the availability of the requested partition. If this is still occupied, the requesting computation unit is stalled by holding its FIFO full flag active.

5.2.2 Conclusions

In the previous paragraphs we have implemented the *Dokládal* algorithm in the form of the 1-D horizontal and vertical units and concatenated them to constitute the 2-D rectangle unit. The rectangle unit processes the image of programmable size in a stream by either erosion or dilation using the rectangular SE of programmable size. All these parameters are programmable until their respective upper bounds that are application-specific and used in the synthesis process as constants affecting the system clock frequency and FPGA area occupation. The memory requirements are very small and the latency is mostly equal to the operator latency (given by causality of the SE). In order to speed up the computation we have applied the concept of *intra-operator parallelism* that allows us to linearly increase the pixel rate of the simple rectangle unit at the cost of increased FPGA area occupation.

The proposed rectangle unit supports simple unit concatenation, so many units can be connected one after each other to create a pipeline that computes the whole application in one image scan. This is very advantageous for traditionally costly operators like the ASF and granulometry.

5.3 2-D Polygonal Dilation Architecture

The rectangular SEs described so far have one unpleasant property for image processing, the angular anisotropy. In the applications of size measurement or image enhancement, one prefers to use more isotropic SEs that affect the image equally in all directions. Because circles with a programmable diameter are very difficult to implement efficiently, one uses approximation by regular polygons that can be decomposed into lines. The theoretical background of this decomposition has been given in Section 4.3 on page 52. Any $2n$-top regular polygon SE P_{2n} can be decomposed into a set of n line SEs L_{α_i}

$$P_{2n} = \underbrace{L_{\alpha_1} \oplus \cdots \oplus L_{\alpha_n}}_{n \text{ times}}, \tag{5-9}$$

hence from

$$\delta_{P_{2n}}(f) = \underbrace{\delta_{L\alpha_1}\,(\,\ldots\;\;\delta_{L\alpha_n}}_{n \text{ times}}\;(f)). \tag{5-10}$$

A hexagon can be obtained by three L_{α_i} oriented in $\alpha_i = 0°$, $60°$, and $120°$ on a 6-connected grid; and an octagon by four L_{α_i}, $\alpha_i = 0°$, $45°$, $135°$, and $90°$ on an 8-connected grid. Therefore, we need to enrich the 1-D dilation block described in Section 5.1, which was designed for the horizontal and vertical orientation only, to rotate the SE under the angles $\alpha_i = 45, 60, 120, 135°$. We will call this 1-D dilation block Line Unit (LU) described in the following.

5.3.1 1-D Line Unit Architecture

The architecture of the LU unit capable of dilation by oriented line segments is shown in Fig. 5.13. The basic behavior of this architecture conforms to the version supporting only horizontal and vertical orientation from section 5.1. A few modifications, however, have been applied to the former version to allow the oblique L_{α_i}. We have namely added the Slope control unit, highlighted in Fig. 5.13.

The other modification is that the memory part instantiates one queue in the case of horizontal segment, N queues in the vertical case (N is the image width), or $N + B_{\mathrm{H}}$ queues in the slope case (B_{H} stands for the horizontal padding size).

Input and output ports are multiplexed; hence a multiplexer select signal can easily address the one queue to work with at a time. The shifted column address col_{shift} (4-11) is used as the select signal according to the theoretical description in Section 4.3.1.

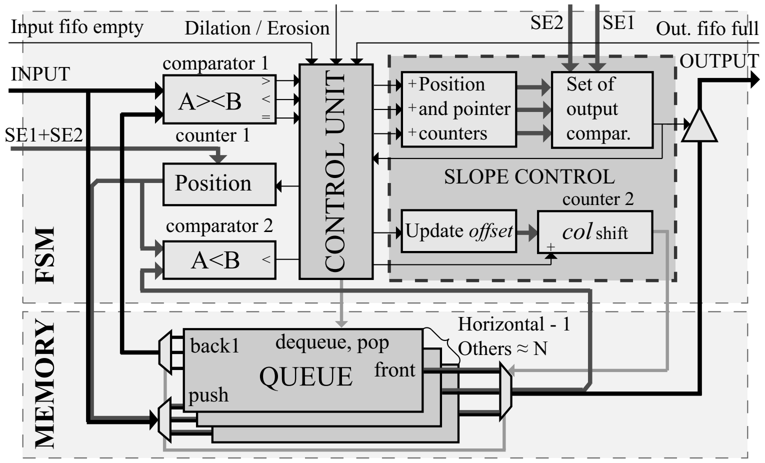


Figure 5.13: Overview of Line Unit architecture. The FSM part manages computation, memory part contains the data storage–queues.

The purpose of the Slope control is to select the corresponding queue memory which is currently used by Alg. 2. The queues are addressed by the col_{shift} counter, which is incremented with every pixel of the input image because any two horizontally adjacent pixels belongs to different corridors, recall $k_i = \tan \alpha_i \leq 1$, and reset at the end of image line. The initial reset value of the col_{shift} counter is *offset* (introduced in Section 4.3.1).

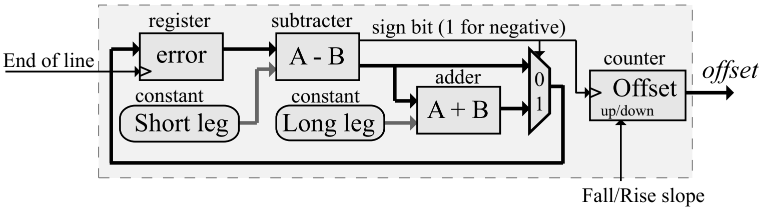


Figure 5.14: Inner schematic layout of Update offset block.

The *offset* is updated at the end of every image line. For some general inclination, it is done on a basis of one iteration of the Bresenham line algorithm that is implemented as shown in Fig. 5.14. It can handle any angle the tangent of which can be expressed as a rational number. In such a case the angle can be expressed by legs of the right triangle including it. For instance, one can use leg constants 26 and 15 for hexagonal SE on an 8-connected grid as the angle is equal to $\alpha = \arctan(26/15) = 60.02°$. As we have already shown, the SEs with angles $\alpha_i \neq i45°$, $i \in \mathbb{N}$ are translation variant that makes them unfeasible for some applications, and therefore, we use hexagons on 6-connected grid only. The use of proper connectivity simplifies the computation of *offset* because $k_{45^\circ} = 1$ and $k_{60^\circ} = 2$ are both integers. Then the circuit for Bresenham algorithm can be replaced by the combination of a simple counter and a 2× frequency divider. The sense of the slope is defined by the *offset* counter direction; up-counting for a fall slope and down-counting for a rise slope.

5.3.2 Polygon Unit Architecture

The previously described LU units can be arranged in a sequence to form a 2-D polygon unit called Polygon Unit (PU). It is allowed by the LU property of the strictly sequential input and output data access. The overall architecture of the PU unit, Fig. 5.15, is composed of three different purpose parts: computation part, controller, and padding part.

The computation part mainly contains four LUs for distinct L_{α_i}. There are the horizontal unit ($\alpha_1 = 0°$), the rise inclined unit ($\alpha_2 = 45°$ or $60°$), the fall inclined unit ($\alpha_3 = 135°$ or $120°$), and the vertical unit ($\alpha_4 = 90°$) connected in a simple pipeline; the output of each unit is read by the successive unit which processes the image by further L_{α_i}. The computation part is able to operate either with hexagon or octagon SE. The vertical unit is bypassed in the case of hexagon SE.

Note that the output of every computation unit is a partially processed, scan-ordered image data which can be brought out and used for another purpose, e.g., a multi-scale analysis descriptor. Then the dilation by line, rectangle, and octagon SEs (all centered) can be obtained during a single image scan (considering units re-ordering). Only the Remove padding block is to be cloned several times for each output data stream.

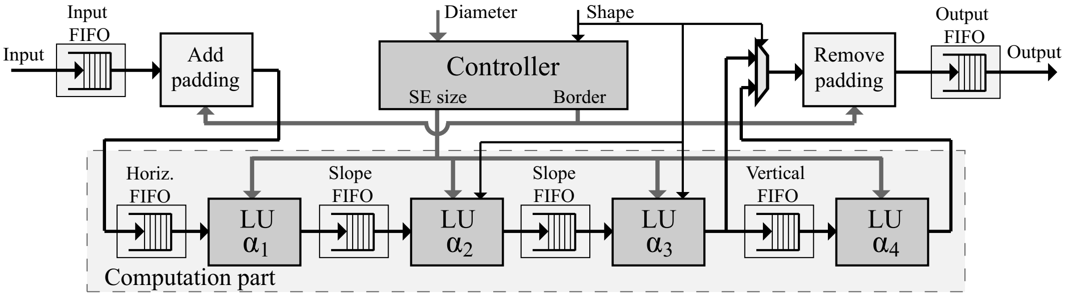


Figure 5.15: Overall architecture of polygonal PU unit. It contains one LU for each $\delta^{\mathrm{L}}_{\alpha_i}$ of (4-7), control, and border handling unit.

According to the borders handling in Section 4.3, the inclined units need the padding to extend the original image at the front end of the processing. The very same padding is removed after the last 1-D unit. It is carried out by a pair of dual padding handling blocks at the beginning and the end of computation part.

The controller ensures the correct global system behavior. It accepts the SE diameter and the shape select signal, then it deduces particular SE sizes for every LU and padding from them, and initiates the computation. The entire set of parameters, i.e., the image width and height, SE features (size and shape), and morphological function select is run-time programmable at the beginning of the frame. These parameters are run-time programmable within the upper bound specified during the synthesis.

For instance, consider the SE size upper bound a 91×91 bounding box, and octagon-capable architecture. This means, the architecture has four LU; each LU supporting at most l=31 pixels segments. During the operation – at the beginning of a frame – the SE can programmed to either of the following: a line up to 31 pixel long, a rectangle up to 31×31 pixels, or an octagon up to 91×91.

To enable processing a uniform input stream, one needs to handle unequal processing rates of LUs. As we know, the algorithm has a variable latency to compute a dilation for one pixel. Therefore, the balancing FIFO memories are inserted in front of each 1-D unit, and to the input and output ports. The depth of input and output FIFOs depends on the timing of input data stream according to (5-5) and (5-6).

Memory Requirements

At this moment, let us evaluate the memory requirements of the polygonal unit PU. Not surprisingly, the obtained formulas are very similar to those obtained for the rectangular unit. The most significant memory demand is made by the set of queues. Although the algorithm works with separated queues, the queues within each LU are merged into a single dual-port memory, mapped side by side in a linear memory space. Every queue has a related pair of front and back pointers which must be retained throughout the entire computation process in the pointer memory. This approach leads to more efficient implementation.

The LUs have the following memory requirements (considering N×M image including padding, L_{α_i} with bounding boxes $W_{\{\mathrm{H,\ V,\ S}\}} \times H_{\{\mathrm{H,\ V,\ S}\}}$, and bpp bits per pixel):

$$M_{\mathrm{hor}} = W_{\mathrm{H}}(bpp + \lceil \log_2(W_{\mathrm{H}} - 1)\rceil) \quad \text{[bits]} \tag{5-11}$$

$$\begin{aligned} M_{\mathrm{ver}} =& N((H_{\mathrm{V}} - 1)(bpp + \lceil \log_2(H_{\mathrm{V}} - 1)\rceil) \\ &+ 2\lceil \log_2(H_{\mathrm{V}} - 1)\rceil) \quad \text{[bits]} \end{aligned} \tag{5-12}$$

$$\begin{aligned} M_{\mathrm{slope}} =& (N + W_{\mathrm{S}})((H_{\mathrm{S}} - 1)(bpp + \lceil \log_2(H_{\mathrm{S}} - 1)\rceil) \\ &+ 2\lceil \log_2(H_{\mathrm{S}} - 1)\rceil) \quad \text{[bits]} \end{aligned} \tag{5-13}$$

Example: suppose a dilation of 8-bit SVGA image (i.e., 800×600=$N \times M$) by a hexagon with radius 41 px. Such a SE is decomposed into horizontal SE 21 px wide, and 2 slope SE each 11 px wide and 19 px tall (hexagon SE bounding box is 41×37 px).

The computation memory (the queues) requires (5-11,5-13)

$$\begin{aligned} M_{\text{hor}} =& 21(8+5) = 273 \quad \text{[bits]} \\ M_{\text{slope}} =& (811+11)((19-1)(8+5)+10) = \\ =& 200,568 \quad \text{[bits]} \end{aligned}$$

resulting in a total consumption of $M_{\text{all}} = M_{\text{hor}} + 2M_{\text{slope}} \cong 392$ kbits for the 2-D dilation by hexagon. This is far below the mere size of the image itself $M_{\text{image}} = 800 \times 600 \times 8bpp \cong 3.66$ Mbits which does not need to be stored at any moment.

5.3.3 Parallel Polygon Architecture

This section describes the Parallel Polygon Unit (PPU) that aims at increasing the computational performance while maintaining as much as possible the beneficial streaming properties of the proposed algorithm as in the case of rectangles. The parallelism is again achieved through utilization of concurrently working units that simultaneously process different parts of the image (spatial parallelism). The number of instantiated units defines the parallelism degree (*PD*), exactly like in the rectangle unit case. Since the processing runs in a stream, we propose a solution that transforms the input stream into a set of *PD* streams in a way to minimize the waiting-for-data periods of all units. For the sake of clarity, we use *PD*=2 in the description hereafter.

The partition of the input image is twofold, see Fig. 5.16: an interleaved line-by-line partition for the horizontal α_1 units, and vertical stripes for the vertical and inclined $\alpha_2, \alpha_3, \alpha_4$ units. The final image partition of 2-D image is the intersection of both.

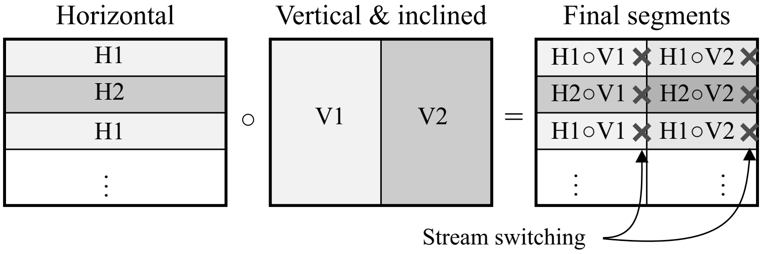


Figure 5.16: Example of image partition for *PD*=2: line by line for horizontal orientation; vertical stripes for non-horizontal orientation.

Intuitively, the streams have to be transformed from one type to the other between α_1–α_2, and α_4–*output* in the PU. The transformation is done by simple circular stream switching when a partition edge is encountered in the very same manner as with rectangles. With the beginning of the image, it starts with the H1∘V1 segment at the first line. When the end of this segment is reached, the streams are switched such that segments H1∘V2 (1st line) and H2∘V1 (2nd line) are processed at the same time. Later, it proceeds to segments H2∘V2 (2nd line) and H1∘V1 (3rd line) and so forth. In general *PD* segments located on a backward diagonal run simultaneously throughout the image (note that the streams are mutually delayed by N/PD pixels).

Processing the partition segments separately introduces undesired border effects on each partition edge. A common solution – similar to padding at image borders – is to introduce an overlap of partitions. Contrarily to the padding that adds recessive values, the overlap extends a partition by a portion of the neighboring partition. The width of the overlap depends on the size of the SE, and is equal to the width of the horizontal padding B_{H}. Intuitively, the overlap introduces redundant computation, and slightly degrades the performance and minimal latency.

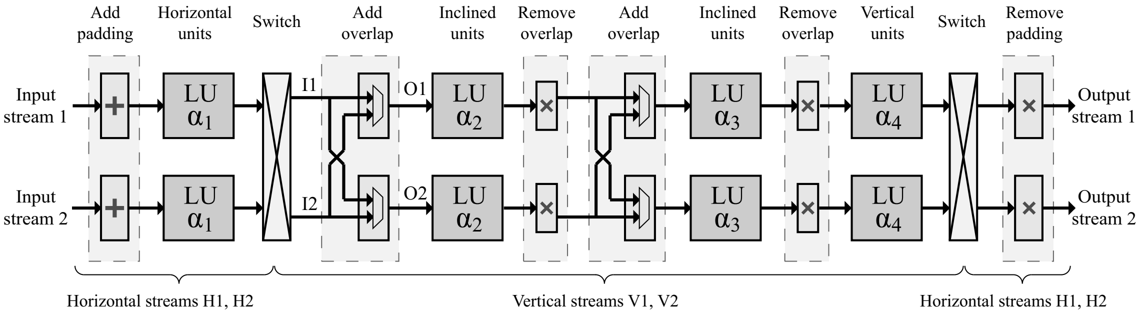


Figure 5.17: Overall architecture of the parallel polygonal unit PPU for *PD*=2. The controller and balancing FIFOs are omitted.

At this point, all the previously mentioned principles are brought together to form the Parallel Polygon Unit (PPU). The PPU (see Fig. 5.17) is scalable with respect to *PD*, the number of parallel streams it can process at the time. Each stream needs one pipeline of four LUs ($\alpha_i, i = 1..4$, just like the PU), two *add overlap* blocks in front of inclined LUs, two *remove overlap* blocks behind inclined LUs, *add padding* at the front end, and *remove padding* at the back end. The PPU also contains a pair of switches to transform the streams from one type to the other.

Figure 5.18 shows the introduction of the overlap in a course of the *i*-th image line. As we know, this line is split into two streams. The streams are labeled I1, I2 before the padding addition and O1, O2 after (refer also to Fig. 5.17). The entire I1 stream plus B_{H} pixels of I2 form O1 output stream with overlap, and last B_{H} pixels of I1 and the whole I2 stream form O2.

During the overlap sections, either I1 or I2 stream is mapped to both output streams at the same time. This data duplication temporarily disables parallel processing of both streams and may result in stalling of either stream. However, the effect of the overlap is negligible given $B_{\mathrm{H}} \ll N$.

Two important properties are to be noted: (i) input and output streams are mutually delayed by N/PD (ensured by stream switching); (ii) several PPUs can be chained into a pipe. The result schematic of some application, e.g., an ASF, may look like Fig. 5.19. At the front end there is an input buffer transforming the input stream (which is *PD*-times faster than each of *PD* processing streams) into *PD*

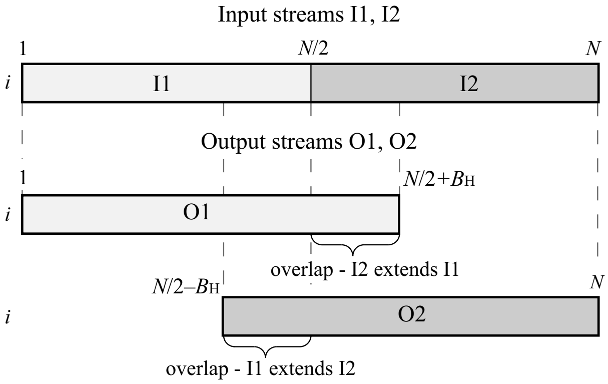


Figure 5.18: Addition of overlap on one image line

processing streams Hi ($i = 1..PD$). The transformation only needs i-th image line to be stored in $\{i \bmod PD\}$-th line buffer. In this manner, the processing streams are properly delayed by N/PD pixels. The output buffer transforms PD processing streams into one fast stream in the opposite way. One can place as many PPUs as desired between these two buffers in a pipeline or other topology.

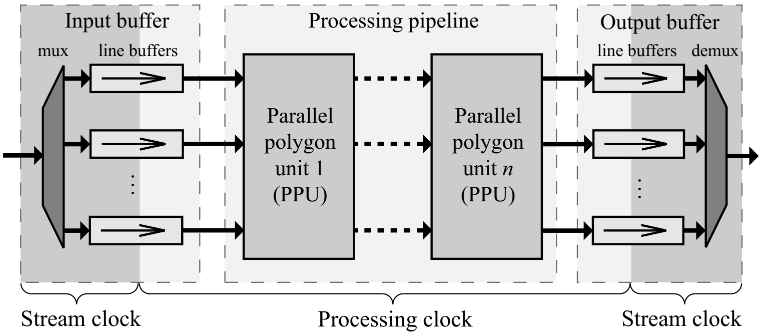


Figure 5.19: Overall architecture of parallel ASF application.

The PPU involves the following limitations on the programmability of parameters: the image size is set before synthesis, and padding sizes $B_{\mathrm{H}}, B_{\mathrm{V}}$ are computed for the maximal allowed SE, specified before the synthesis. The reason is that handling the varying SE and image sizes would introduce an unreasonable hardware overhead of image partition, padding, and overlap features. The SE remains fully programmable.

5.3.4 Conclusions

In the previous paragraphs we have enriched the original implementation of the *Dokládal* algorithm (for horizontal and vertical SE) to support the inclined SEs and called it the Line Unit (LU). Then we have placed four LU units in a pipeline to obtain the polygon unit. The polygon unit processes the image of programmable size in a stream by either erosion or dilation using the polygonal SE of programmable size. All these parameters are programmable until their respective upper bounds that are application-specific and used in the synthesis process as constants affecting the system clock frequency. The memory requirements are very small and the latency is mostly given by the operator latency (given by causality of the SE) and the padding issue.

In order to speed up the computation of a simple polygon unit we have applied the concept of *intra-operator parallelism* that allows us to linearly increase the pixel rate of the simple polygon unit at the cost of increased FPGA area occupation. However, the parallelism combined with padding and overlap issues limits the programmability of the image size, which is an application-specific parameter in the parallel case.

The proposed polygon unit also supports unit concatenation, so many units can be connected one after each other to create a pipeline that computes the whole application such as granulometry or ASF in one image scan.

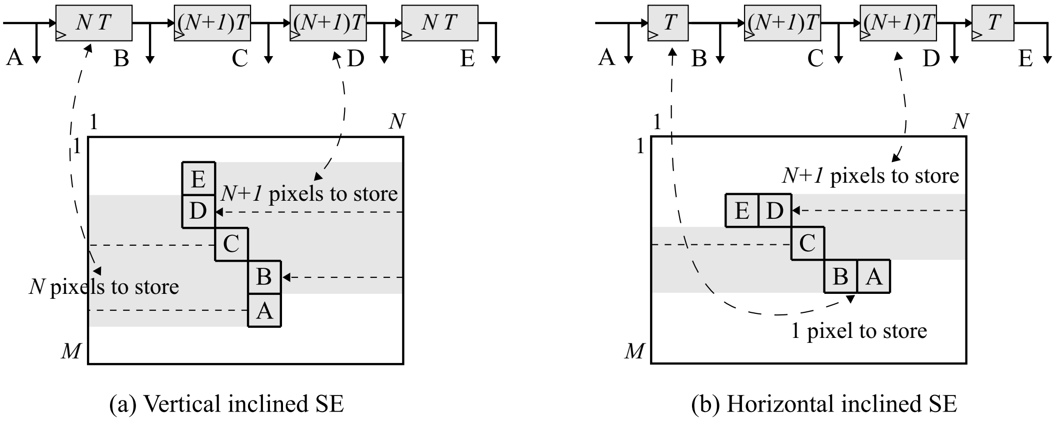
5.4 1-D Synchronous Dilation Architecture

In this section we describe a synchronous dilation architecture supporting long line SEs oriented at an arbitrary angle. This architecture uses basic, well-known principles, such as delay lines and full search for maximum, which has been used in many past implementations, see Section 3.2.3 of the introduction. However, the previous work supports either basic SE shapes (horizontal, vertical lines, rectangles by decomposition) or small 3×3 arbitrary SEs. The methods based on partial-result reuse provide more complex, usually convex, shapes but the programmability of the size and shape of the SE is compromised. The architecture proposed in this section computes dilation by translation-invariant, arbitrary-oriented line SE that has not been reported in the literature yet. The length and orientation of the SE are run-time programmable parameters (until an upper bound given before the synthesis).

The computation of dilation conforms to the definition (2-4) on page 10, so it proceeds in two steps: SE extraction, and maximum search over the extracted SE.

SE extraction is based on delay lines and defines which pixels of an image are at a given time covered by the SE. The SE is considered to be a discrete line of length l oriented at angle α. The origin of the SE is located in the middle of the SE, so l is an odd number; and the SE is symmetric. Examples of line SE extraction for vertical and horizontal inclined SEs are shown in Fig. 5.20. As for the vertical case in Fig. 5.20 (a), any two adjacent pixels of the SE are either N or $N+1$ pixels of datastream apart of each other. For instance, the delay between pixels A and B is equal to N, the delay between C and D is $N+1$. In the case of a horizontal inclined SE in Fig. 5.20 (b), any two adjacent pixels are either 1 or $N+1$ pixels apart. Note that distance from A to B is 1 pixel and distance from C to D is $N+1$ pixels (with respect to the image dataflow). Hence, a chain of delay lines with programmable delay of either 1, N, or $N+1$ pixels extracts an oriented line

SE. Such a programmable delay line is labeled by xT (x stands for $\{1, N, N+1\}$).



(a) Vertical inclined SE

(b) Horizontal inclined SE

Figure 5.20: Extraction of inclined line SEs using programmable delay lines: (a) vertical inclined lines, (b) horizontal inclined SE.

A naive computation of maximum on the extracted SE is carried out by a cascade of dyadic max() operators represented by $\vee$ in Fig. 5.21. Another possible hierarchy of dyadic max() operators is a binary tree, however, a cascade allows for better programmability of the SE length l. The number of max() operators $l-1$ is the same either way (e.g., l=8, the cascade needs $l-1=7$ operators and binary tree needs 4 in the first layer, 2 in the second, and 1 in the last layer, so 7 operators in total). The proposed cascade computes dilation immediately with no additional delay. On the other hand, real hardware implementation would be very slow due to a long critical path from the input through all $l-1$ max() operators to the output.

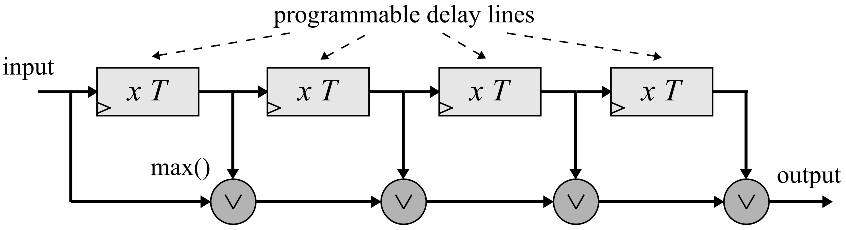


Figure 5.21: Simplified architecture of the naive approach to dilation.

In order to reduce the critical path, we pipeline the cascade by placing a register in front of each max(), see Fig. 5.22 for the architecture. At the same time we must extends delays of the delay lines to counterbalance the delay of inserted registers, for instance by inserting registers behind delay lines as well. The pipelined architecture allows for fast hardware implementation, but also introduced an additional computation latency $l-1$ pixels.

We already mentioned that the cascade is advantageous for programmability of the SE. It is because of the n-th partial results ($n \in \mathbb{N}$, $n < l-1$) of the cascade is equal to dilation by a partial SE containing $n+1$ recent pixels. Therefore, selecting an appropriate partial result is equal to programming the SE length. Instead of

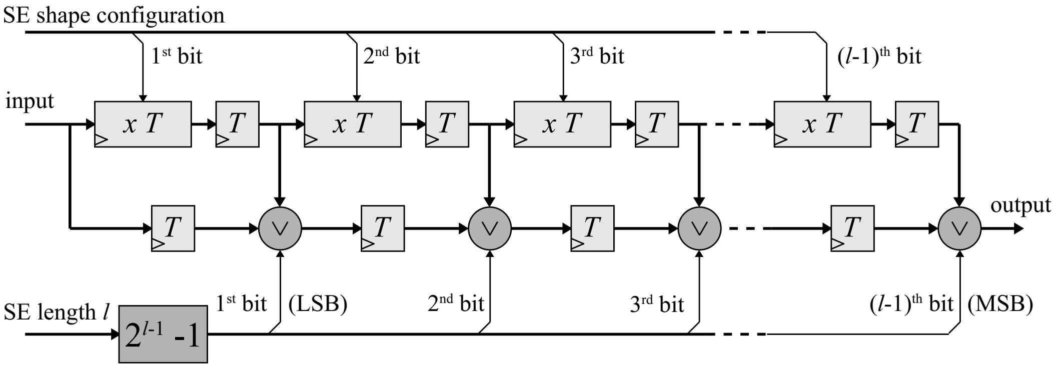


Figure 5.22: Naive architecture supporting programmable oriented line SE.

using a multiplexer, which would need to have $l-1$ inputs and would deteriorate performance (for example $l=31$, 8-bit 31-to-1 multiplexer is very complex circuit), we propose a method of enabling max() operators.

The inner schematic of the max() operator with enable input is displayed in Fig. 5.24 (b). First, the two operands, A from the precedent max() in the cascade and B from a delay line, are compared. The result of comparison is connected to a multiplexer that selects the larger operand. The enable signal when set to 0 overrides the result of the comparison and selects A to be the output regardless the value of B, such as

$$Y = \begin{cases} A & \text{enable} = 0 \\ \max(A, B) & \text{enable} = 1 \end{cases} \tag{5-14}$$

So when the max() is not enabled, it only propagates the result of the precedent max() to the output.

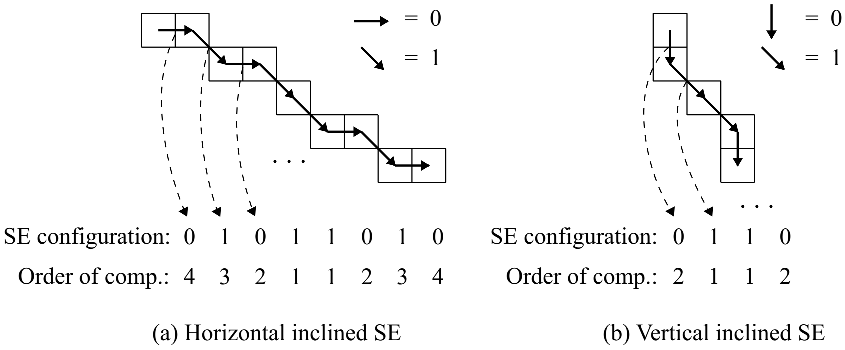

= 0
= 1
SE configuration: 0 1 0 1 1 0 1 0
Order of comp.: 4 3 2 1 1 2 3 4
(a) Horizontal inclined SE
= 0
= 1
SE configuration: 0 1 1 0
Order of comp.: 2 1 1 2
(b) Vertical inclined SE

Figure 5.23: SE shape configuration. Whenever two adjacent pixels belong to the same line (or column), the configuration bit is set to logic 0, otherwise to logic 1. (a) horizontal case, (b) vertical case.

For a given SE length l only first $l-1$ max() operators, which are necessary, are enabled. The enable signal for the n-th max() operator is given by the n-th bit of the binary number $2^{l-1}-1$, where the least significant bit (LSB) is the first bit. For example consider 9-pixel SE $l=9$ (the cascade is designed for $l_{\max} \geq 9$). The enable word $2^{9-1} - 1 = 255_{10} = 0..0\ 1111\ 1111_2$ shows that the first eight operators are enabled and all further operators are disabled. The number of disabled operators depends on length $l_{\max}$ of the cascade.

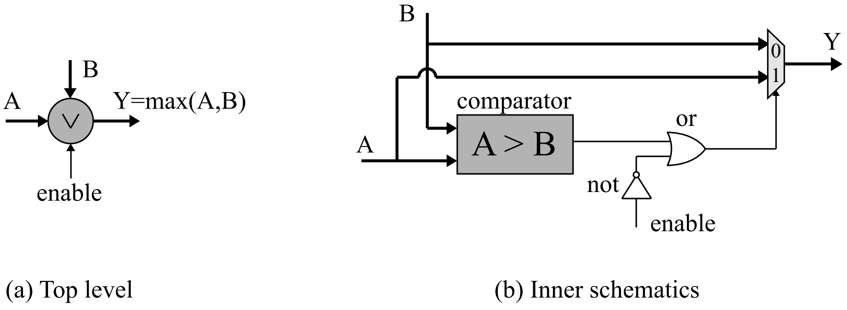


Figure 5.24: Max() operator: (a) top-level symbol, (b) detailed inner view.

In the architecture scheme shown in Fig. 5.22, the shape of the SE is configured by $(l-1)$-bit configuration word. Each bit of this word is connected to one delay line and selects the value of delay. Notice that only two different values of delay are possible for given steepness (either horizontal or vertical). Figure 5.23 (a) displays an example of how the SE shape configuration looks for a horizontal inclined SE. When two adjacent pixels belong to the same line, the delay between them is 1 pixel; when they belong to two different lines, their mutual delay is $N+1$ pixels. We assigned the 1-pixel delay to logic 0, and the delay $N+1$ to logic 1 for the configuration word. The vertical case is treated in a similar way, see Fig. 5.23 (b).

Let us recall that the SE is defined by its length l, orientation α, and that the SE is symmetrical. So we need to compute only a half of the configuration word beginning from the center. The other half is mirrored. An order of the computation is included in Fig. 5.23 as well. The computation itself is carried out in the same way as described in Sections 4.3.1, so by the Bresenham algorithm Alg. 4 on page 56 implemented in very similar way to one shown in Fig. 5.14. The configuration block computes the SE configuration word on basis of the SE length and angular coefficients (*Short_leg*, *Long_leg*).

Memory Requirements

The only memory requirement is made by the set of programmable delay lines that must fit the whole image line of length N. The synchronous architecture has the following memory requirements (considering N×M image, l length of the SE, and *bpp* bits per pixel):

$$M = N(l-1)bpp \quad \text{[bits]} \tag{5-15}$$

Example: suppose a dilation of 8-bit SVGA image (i.e., 800×600=$N \times M$) by an arbitrarily oriented line SE of $l = 31$. The set of delay lines requires (5-15)

$$M = 800(31-1)8 = 192,000 \quad \text{[bits]}$$

This is far below the mere size of the image itself $M_{\text{image}} = 800 \times 600 \times 8bpp \cong 3.66$ Mbits which does not need to be stored at any moment.

5.4.1 Conclusions

In the previous paragraphs we have implemented the synchronous dilation unit based on the traditional SE extraction by delay lines and full maximum search by a chain of dyadic maximum operators. This synchronous unit processes the image of programmable size in a stream by either erosion or dilation using the arbitrary-oriented line SE of programmable size. All these parameters (image size, length and orientation of the SE) are programmable until their respective upper bounds that are application-specific and used in the synthesis process as constants. The memory requirements are very small and the latency is equal to the operator latency (given by causality of the SE).

The proposed unit follows the traditional approach of dilation computation that combined with arbitrary orientation gives us a tool that better handles orientation-sensitive information in images. For example the oriented line dilation can be used to detect the dominant orientation, or when two units are concatenated, they compute 1-D oriented opening suitable for granulometry or detection of the texture orientation. Also, the translation-invariant 1-D SEs can be conveniently used in 2-D SE composition to form exact polygons on the 8-connected grid.

5.5 1-D Opening and Spectrum Architecture

This section describes the hardware implementation of the *streaming peak elimination* algorithm for 1-D oriented opening and pattern spectrum already introduced in Section 4.4 and Alg. 6 on page 61. The proposed architecture computes both opening γ_l^α and $PS(\alpha, :)$ on a horizontal scan ordered data stream of the input image. It consists of two parts (see Fig. 5.25): an opening part that controls the entire algorithm behavior and calculates the opening; and a spectrum accumulation part that stores and accumulates the spectrum.

The heart of the opening part is a sequential algorithm control block. It is a Finite State Machine implementing the conditional behavior of the algorithm (*while*, *if*) and ordering of commands in a similar way as the dilation algorithm was implemented. At first, let us focus on how an input pixel is processed. We assume the horizontal SE orientation, and the Array of queues AQ contains only one queue Q for the sake of simplicity.

In the beginning, the input pixel F is compared with two pixels previously stored in Q by comparators 1–3 in order to reveal a peak value to drop. If a peak is recognized, the controller issues the dequeue and accumulate operations, and reads two other latest pixels in order to iterate the *while* loop one more time. When no peak is found, F along with its reading position rp is pushed into the queue. The outdated pixel is potentially popped by comparator 4, and the value of the oldest

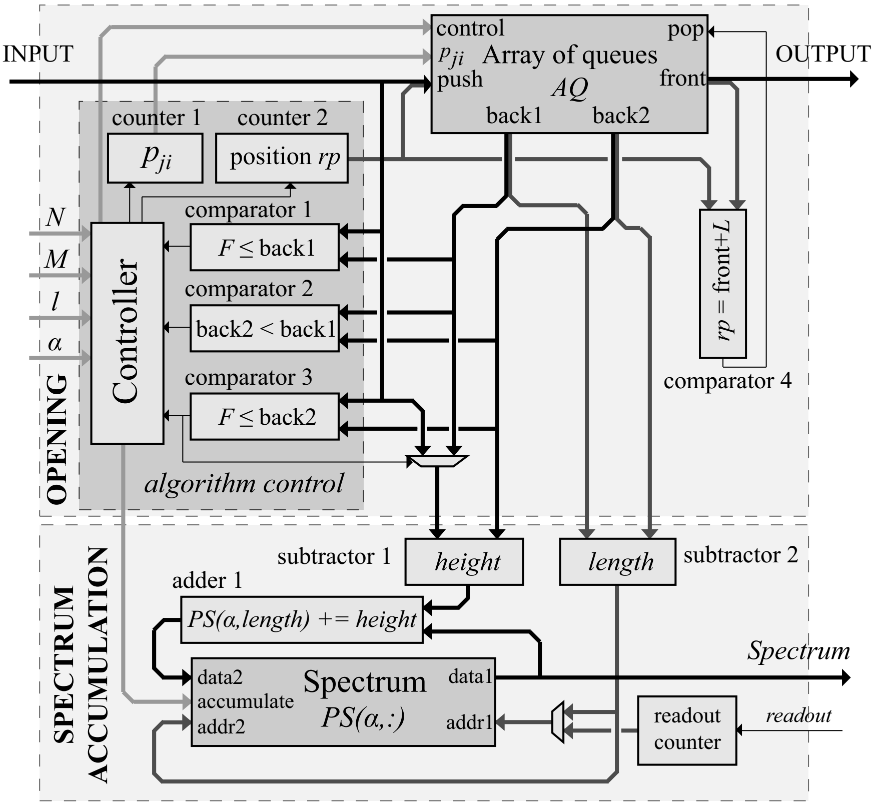


Figure 5.25: Architecture of opening unit with spectrum accumulation.

queue element is the output of opening.

The start and end points of the dropped peak slice are passed to the spectrum accumulation part, which computes *length* and *height* of the peak slice. The accumulation part then retrieves the *length*-th spectrum value $PS(\alpha, length)$, and adds the *height* of the dropped flat zone. As soon as the whole image is processed, the spectrum $PS(\alpha, :)$ can be read out by some simple back-end controller through a standard FIFO interface.

5.5.1 Arbitrary Orientation

The used algorithm supports arbitrary orientation of the SE; and as the horizontal orientation alone limits applicability of the 1-D opening, we are interested in developing the architecture supporting arbitrary orientation as well. In Section 4.4.3 on page 64 we mentioned that one of the best approaches is to partition an image into corridors parallel with the SE and let each corridor have its own designated queue memory Q, one from the array of queues AQ. As the algorithm reads data sequentially in the horizontal scan order, i.e., for most orientations across the corridors, the main task is to determine to which corridor $Q_{ji} = AQ(p_{ji})$ the current pixel at position $[j, i]$ belongs. Then the task of SE orientation reduces to the mere calculation of the select pointer p_{ji} such as (reminder of equation (4-13))

$$p_{ji} = \begin{cases} (i - j\tan(90° - \alpha))\mathrm{mod}(N + l\cos 45°) & \text{if } 45° \leq \alpha \geq 135° \\ (j - i\tan\alpha)\mathrm{mod}(N + l\cos 45°) & \text{otherwise.} \end{cases} \tag{5-16}$$

The p_{ji} computation uses the Bresenham line algorithm Alg. 4 to efficiently handle the $j\tan(90° - \alpha)$ (or $i\tan\alpha$) from (5-16) in hardware, as it was the case of inclined dilation. This term is called *offset* in accordance to the previous terminology. The detailed architecture of the p_{ji} counter from Fig. 5.25 is depicted in Fig. 5.26. The main difference against the former implementation is that the oriented opening demands all the possible angles in contrary to a single orientation for polygons. Therefore, the appropriate *Long_leg* and *Short_leg* constants are read from the memory of coefficients before the computation begins. Also the orientation α is threshold by 90° to determine the fall/rise slope sense. Again the value of *offset* is the initial/reset value of the *pointer* counter, the output of which is the desired value of p_{ji}. Notice that selecting the appropriate queue by p_{ji} completely ensures the image partitioning into corridors, and consequently, orientation α of the SE.

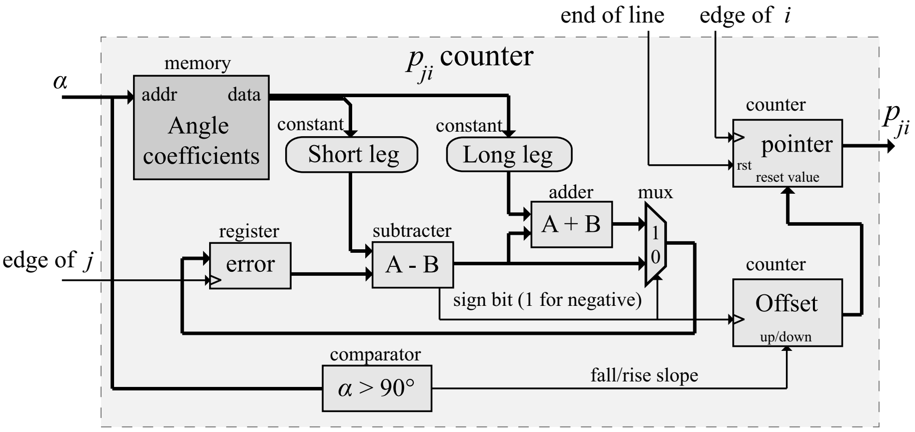


Figure 5.26: Schematic of the *offset* computation using Bresenham algorithm.

Having introduced arbitrary orientation, the Array of queues no longer contains only one queue as for the horizontal SE, but $N + l\cos 45°$ queues instead. It can implement entire AQ in a single dual-port memory because only one Q chosen by p_{ji} is used at a time. Packing queues into one memory eases implementation process and allows the use of either on-chip block RAM or off-chip RAM memory.

The proposed opening unit allows the inter-operator parallelism to be employed when necessary. Thanks to the dataflow processing, a single input data stream is brought to multiple units working in parallel with different parameters, e.g., orientation α. However, some synchronization problems may occur on the output ports due to different latency. We will consider both opening and pattern spectrum separately.

In the case of the patter spectra computation for multiple angles, there is no issue likely to appear. The respective spectra $PS(\alpha_i,:)$ represent relatively small amount of data and they can be read out in any order. Also stalling the read-out process does not really compromise performance. For instance, the complete spectrum $PS(:,:)$ with $\Delta\alpha = 180°/n$ can be calculated in a single image pass using n parallel units connected as in Fig. 5.27 at almost the same time like single spectrum $PS(\alpha_i,:)$.

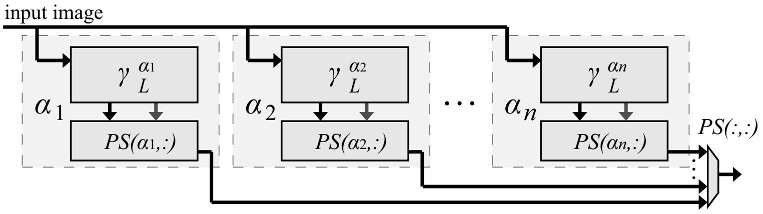


Figure 5.27: Parallel computation of $PS(:,:)$ using n units.

The case of opening is different. The output data contain the entire image in the form of a data stream, which can not be stalled for a long time. One usually does not have sufficient bus bandwidth in order to read out all n output streams independently (except some platforms equipped with DDR memories, and small values of n). Therefore, inter-operator parallelism is supposed to be used rather in applications that allow some further pre-processing before outputting the result in order to decrease the output bandwidth. An illustrative example of such an application is opening χ_l, which runs as (introduced in (2-32))

$$\chi_l(f) = \bigvee_{\alpha_i \in [0,180)} \gamma_l^{\alpha_i}(f). \tag{5-17}$$

In this form χ_l computes the pixel-wise supremum of 180 openings γ_l^{α}, that is for 180 different angles with the orientation resolution $\Delta\alpha = 1°$. According to the associativity of the supremum, instead of first computing 180 openings and then finding the supremum, we clearly can compute only several openings, find the supremum of them that is a partial result $\chi_l^{(j)}$ given as

$$\chi_l^{(j)}(f) = \bigvee_{\alpha_i \in \mathcal{A}_j} \gamma_l^{\alpha_i}(f), \tag{5-18}$$

where $\mathcal{A}_j$ is an n-element set of angles that constitutes the j-th partial opening. We can repeat the computation m times for different orientation (with the same input image, therefore, the input image has to be stored in some memory), and find the supremum of all partial results $\chi_l^{(j)}$, $j \in [1,m]$, afterwards in some post-processing. The post-processing pixel-wise supremum of m images is not particularly computation intensive and can be easily done in either computer or dedicated hardware.

Implementing the equation of $\chi_l^{(j)}$ opening (5-18), two facts should be considered. First, we need to compute n openings $\gamma_l^{\alpha_i}$ with constant length l but with different orientations α_i that causes different latencies of the opening units.

However, as the supremum operation is commutative, we can always sort $\mathcal{A}_j$ such that the delays of openings $\gamma_l^{\alpha_i}$ for all $\alpha_i \in \mathcal{A}_j$ are ordered in an increasing order. The delay of the proposed opening unit is proportional to the vertical projection of the SE $|l \sin \alpha_i|$, so the previous condition can be formulated as $|\sin \alpha_i| \leq |\sin \alpha_{i+1}|$, $i \in [1, n)$. For example, $\mathcal{A}_j = \{0°, 45°, 90°, 135°\}$ should be sorted into $\mathcal{A}_j = \{0°, 45°, 135°, 90°\}$ because $|\sin 135°| < |\sin 90°|$.

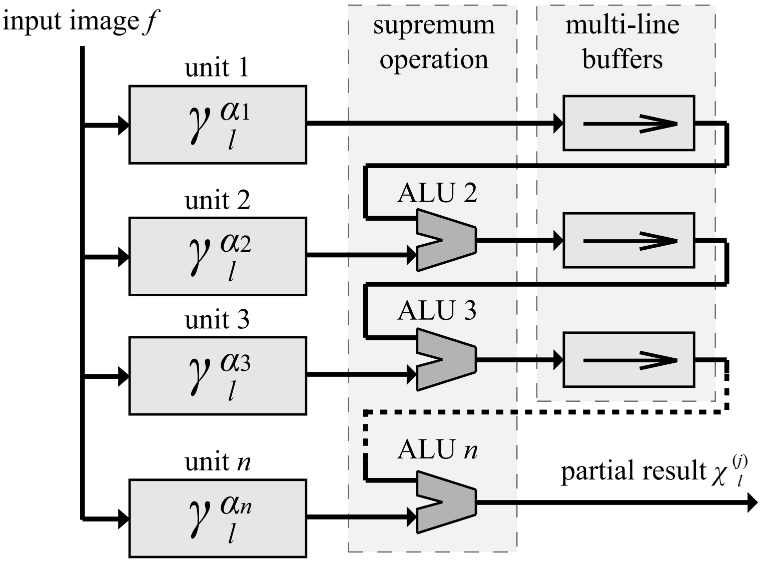


Figure 5.28: Parallel computation of j-th partial result $\chi_L^{(j)}$ using n opening units.

Second, the supremum operator in (5-18) is associative, so it can be implemented as a chain of diadic supremum operators. The basic architecture for computing $\chi_l^{(j)}$ is shown in Fig. 5.28. All n parallel units are fed by the common input image stream. Provided $|\sin \alpha_i| \leq |\sin \alpha_{i+1}|$, the output of unit 1 is connected via a multi-line buffer, which balances the delay difference of units 1 and 2, into ALU 2 that computes the first supremum of $\gamma_l^{\alpha_1}$ and $\gamma_l^{\alpha_2}$. The output of ALU 2 is connected also via a multi-line buffer to ALU 3, and so forth.

Memory Requirements

The proposed architecture has very limited memory consumption. Although the algorithm works with many separated queues, the queues of each 1-D opening unit are merged into a single dual-port memory, mapped side by side in a linear memory space. Every queue has a related pair of front and back pointers which must be retained throughout the entire computation process in the pointer memory. This is the same approach that was chosen in the case of the dilation architecture.

Let l, α denote the length and orientation of the line SE L_l^{α}, and *bpp* bits per pixel of an $M \times N$ image. The memory consumption of purely horizontal, and arbitrary-oriented implementations are, respectively:

$$M_{hor} = l(bpp + \lceil \log_2(l-1) \rceil) \qquad \text{[bits]} \qquad (5\text{-}19)$$

$$M_{orient} = (N + l\cos 45^\circ) \times [l(bpp + \lceil \log_2(l-1) \rceil) + \\ + 2\lceil \log_2(l-1) \rceil]. \qquad \text{[bits]} \qquad (5\text{-}20)$$

Example: Let us compute an opening of 8-bit, SVGA image (i.e., $800 \times 600 = N \times M$) by a SE 41 px long. The computation memory requires (5-19) (5-20)

$$M_{\mathrm{hor}} = (41)(8+6) = 574 \qquad \text{[bits]}$$

$$M_{\mathrm{orient}} = (800+29)(41(8+6)+12) = 485,794 \qquad \text{[bits]}$$

resulting in the maximal consumption of $M_{\max} = M_{\mathrm{orient}} \cong 475$ kbits for the arbitrary-oriented opening. This is once again far below the mere size of the image itself $M_{\mathrm{image}} = 800 \times 600 \times 8bpp \cong 3.66$ Mbits which is never stored. It is worth mentioning that these numbers are effectively equal to the memory consumption of the dilation unit. If we wanted to compute the opening by composition (such as $\gamma_B(f) = \delta_{\widehat{B}}[\varepsilon_B(f)]$, (2-18)), we would have to use two dilation/erosion units, which have together as twice as large memory requirements. In addition, the composition can be used for the translation invariant SEs only (more precisely, it must be possible to have $\widehat{B}$ for any given B), so the discrete-line approach to arbitrary orientation can not be used.

5.5.2 Conclusions

In the previous paragraphs we have implemented the *streaming peak elimination* algorithm in the form of the 1-D oriented opening and pattern spectrum unit. This unit processes the image of programmable size in a stream. It computes opening by the 1-D oriented SE of programmable orientation α and length l and the pattern spectrum $PS(\alpha, :)$ for all possible lengths shorter than l. All these parameters are programmable until their respective upper bounds that are application-specific and used in the synthesis process as constants. The memory requirements are very small and the latency is mostly equal to the operator latency (given by causality of the SE).

The proposed rectangle unit supports simple parallel interconnection so multiple units can work in parallel processing the same input image stream by different operators. This property is convenient for the computation of the full orientation spectrum $PS(:, :)$ or opening $\chi_l(f)$ (5-17), which would need many image scans if computed sequentially.

5.6 Conclusions

In this chapter we have presented the dedicated hardware implementation of the chosen algorithms in the form of dataflow processing units. Their common property is programmability; the image size, SE features, and selected operation are run-time parameters setting up the processing unit according to the application currently in hand. The parameters are bounded by some maximal values, which are passed to the synthesis process to produce an appropriately scaled design. As the size of queue memories is the element most sensible to parameters, the memory consumption seems to be the main consideration when the parameter boundaries are chosen. In addition, once the design is synthesized with given boundaries, a change of parameters results in only a partial use of the allocated memory. Therefore, the boundaries should be kept as small as possible whenever the application allows it.

Another shared property of all proposed units is the dataflow processing. The given operation is applied to the scan-ordered stream of data as it inputs the unit, and the result data leave the unit in the very same order. This property allows for easy serial or parallel interconnection hierarchies, which conveniently enhance performance through so-called *inter-operator parallelism.* Although no coupling elements are essentially necessary (unless stall-free processing), small FIFO memories are used to suppress the data dependency and dataflow stalling.

The architectures providing a dilation by rectangles and polygons were further enhanced by so-called *intra-operator parallelism.* This method instantiates multiple copies of the proposed units in parallel, and in order to keep them all working at the same time, it divides the data stream into multiple slower substreams, each of which is processed by one unit from multiple parallel copies. The division of the data stream can also be seen as division of the image into partitions, a couple of which are processed at a time. This parallel approach is finely scalable, so the design may be adjusted appropriately to meet the application requirements.

The opening and pattern spectrum incorporates only the *inter-operator parallelism* paradigm because the 1-D opening itself is scarcely used alone but, on contrary, in complex applications that comprise many opening operations. Then the broad *inter-operator parallelism* level provides a sufficient way to efficiently exploit the available hardware resources, linearly speeding up the performance of the whole application. Without having implemented it, we allege that the *intra-operator parallelism* can be used with the oriented opening as well if necessary. This case is very similar to the inclined units of polygons, so the whole parallel processing issue can be solved by the same method (i.e., image partition, overlap of partitions, etc.). We did not do so because the *inter-operator parallelism* alone is sufficient and also it is more hardware-efficient than the *intra-operator parallelism.*

All proposed units in the past chapter are supposed to be used as basic bricks to build up larger, more complex architectures. For this reason, two main aspects were especially taken into account to ease the design: (i) dataflow processing that enables easy block interconnection, and (ii) programmability of the main properties. In the following chapters, we will measure some important features and evaluate the aforementioned properties on a set of applications.

6 Implementation Results

Contents

In this chapter, we present implementation results of the proposed architectures by means of timing and FPGA resources. These basic measures may help to easily estimate the system performance and FPGA resources occupation, and to observe how the architectures scale with different parameters. We begin with the rectangular dilation block and its parallel version, the polygonal dilation block and its parallel version, and finally, the opening and pattern spectrum block. All these architectures have been synthesized by the XST tool and targeted to the Xilinx Virtex5 or Virtex6 FPGAs (XC5VSX95T-2, XC5VLX50T-1, or XC6VLX240T-1).

Before looking at the experimentally obtained figures, let us focus on some common properties of both *Dokládal* and *streaming peak elimination* algorithms and discuss their implications on hardware. Both algorithms are queue-based; that means their behavior is mainly based on operations over a queue memory, and these operations take the majority of the algorithm computation time. For this reason, the way how the queues are implemented is important.

As we already know, the queues in each 1-D unit are merged together into a single dual-port memory block in order to allow taking advantage of the Block RAMs (BRAM) available in an FPGA. Although the algorithms were presented as having small memory requirements (compared with other algorithms they are indeed small), their values are rather large from the FPGA resources point of view. For instance, enumerating (5-8) for an SVGA image 800×600 and 31×31 SE, we get the queue requirements of 322,400 bits (24,800×13 bits) in contrast to 36 kbits (2,048×18 bits) of a single BRAM. So, 13 BRAMs ($\lceil 24,800/2,048 \rceil$ using *low power* mapping algorithm of the Xilinx taxonomy) are to be used in practice to fit the queues. The size of this memory and its access time inevitably affects the critical path delay, and therefore, the maximum frequency.

The algorithm FSM and balancing FIFOs are straight-forward synthesized by the XST in the *logic*, i.e., look-up tables (LUT) and registers, and their impact on implementation result is rather less significant. The size of distributed balancing FIFOs varies with $N\sqrt{l}$ and the FSM scales only slightly with the increasing bit-width of the SE size $\lceil \log_2(l) \rceil$, and the image size $\lceil \log_2(N) \rceil \lceil \log_2(M) \rceil$. Assuming some approximation and the fact that FIFOs may be omitted, the amount of necessary logic is proportional to the factor

$$\lceil \log_2(l) \rceil \lceil \log_2(N) \rceil \lceil \log_2(M) \rceil. \tag{6-1}$$

The obtained maximum frequency varies between 150 and 180 MHz in dependence on many factors, such as the supported image size, the speed grade and type of the FPGA, various synthesis tool settings. For the sake of simplicity, we use the frequency of 100 MHz or 125 MHz in all demonstration platforms and we assume the frequency of 100 MHz in the following timing tables.

We evaluated mainly two kind of timing benchmarks, one against varying size of the SE, and one against varying image size. We used natural photos of multiple resolutions as test images: CIF 352×288, VGA 640×480, SVGA 800×600, XGA 1024×768, and 1080p 1920×1080. We report several measures as follows.

- The latency is expressed in a number of image lines. The latency is mostly equal to the operator latency, the further irreducible factor corresponding to dependency of the output on the input. This corresponds to the half-, or full-, height of the SE that we need to wait to have read enough data to compute dilation, or opening, respectively. Note that this abstraction of latency is not pixel-wise precise, the exact latency may vary in order of pixels depending on the image contents.
- The experimental pixel rate $PR_{\text{experimental}}$ gives an average number of clock ticks to process one pixel. It is given by the overall number of clock ticks divided by the image size as

  $$PR_{\text{experimental}} = \frac{T_{\text{proc}}}{M\,N} \qquad [\text{clk/px}]. \tag{6-2}$$

  One can observe that the rate is almost constant w.r.t. both the size of the image and the size of the SE. The slight variation is caused by the bounded support effects verifying the computation complexity per image $\mathcal{O}_{\text{image}}()$.
- The compensated pixel rate $PR_{\text{compensated}}$ eliminates the effects of image boundaries and verifies $\mathcal{O}(1)$. This artificial measure serves for the purpose of constant complexity $\mathcal{O}(1)$ demonstration and is computed as

  $$PR_{\text{compensated}} = \frac{T_{\text{proc}} - 2(l_{\text{right}}M + l_{\text{down}}N)/PD}{M\,N} \qquad [\text{clk/px}] \tag{6-3}$$

- The FPS is the throughput in terms of the number of frames per second computed from the clock frequency and the measured time of processing

  $$FPS = \frac{f_{\text{clk}}}{T_{\text{proc}}} \qquad [\text{frame/s}]. \tag{6-4}$$

- The experimental speed-up shows the performance increase of the parallel architecture over the simple one experimentally measured on a real image. It is computed as

$$\text{Experimental speed-up} = \frac{T_{\text{simple}}}{T_{\text{parallel}}(PD)}, \tag{6-5}$$

where T_{simple} is the processing time of the simple architecture and $T_{\text{parallel}}(PD)$ is the processing time of the parallel architecture of PD degree.

We also measures FPGA resources in terms of a number of used LUTs (look-up tables), registers, and BRAMs (36-kbit block dual-port on-chip RAM memories) for the two aforementioned benchmarks, one against varying size of the SE, and one against varying image size. In this case the current value of either image or SE size is considered to be the upper bound that scales the FPGA implementation. Obviously changing a run-time programmable parameter of a processing unit has no effects on implementation results.

6.1 Rectangle Dilation Unit

Let us begin with the non-parallel rectangular dilation block, the benchmarks of which are outlined in Tables 6.1 and 6.2. The measured latency is equal to the operator latency, which means it is equal to l_{down}, i.e., a half of the rectangular SE height. The $PR_{\text{experimental}}$ shows a small fluctuation with varying both SE and image size as this measure reflects the image boundary effects $\mathcal{O}_{\text{image}}((N + l_{\text{right}})(M + l_{\text{down}}))$. Then the pixel rate is deteriorated by $(l_{\text{right}} + l_{\text{down}})/(M + N)$ fraction. The boundary influence is eliminated in compensated pixel rate $PR_{\text{compensated}}$ that effectively remains constant illustrating thus $\mathcal{O}(1)$.

Concerning the area occupation (see the Xilinx documentation for more details [Xilinx 2009]), the amount of the glue logic is logarithmically dependent on N, M, l according to (6-1); the number of LUTs and registers increases when a power of two of either parameter is exceeded. The number of BRAMs also follows the computed requirements (5-8), but due to the fixed memory geometry (*addresses* × *bits-of-word*) its value is more discrete; it increases in larger steps. For instance, assume VGA image and 31×31 SE. The required memory has 640×31=19,840 addresses and each word is 8+$\log_2$(31)=13 bits wide. Such a memory fits to 13 36-kbit BRAMs, one for each bit of the word. But 13 BRAMs provide up to 36,864 addresses, so the image width can go as high as 36,864/31=1,189 with no increase of BRAM occupation.

At this moment, let us proceed to the parallel version of the rectangular dilation unit, which was presented in Section 5.2.1. Table 6.3 presents the most important measure, the influence of intra-operator parallelism degree PD. The increasing PD has beneficial implications on the pixel rate, the value of which is merely a fraction (divided by PD) of the non-parallel values. The latency remains the same by

Table 6.1: Rectangle unit: timing and area w.r.t. SE size, SVGA image size.

Size of SE (square)	3x3	11x11	21x21	31x31	41x41
Latency [image line]	1	5	10	15	20
$PR_{\text{experimental}}$ [clk/px]	2.344	2.379	2.409	2.440	2.470
$PR_{\text{compensated}}$ [clk/px]	2.338	2.350	2.350	2.352	2.353
Registers	212	232	242	242	252
LUTs	584	761	859	859	953
Block RAMs	2	6	13	13	28

Table 6.2: Rectangle unit: timing and area w.r.t. image, SE = 31x31 square.

Size of image	CIF	VGA	SVGA	XGA	1080p
Latency [image line]	15	15	15	15	15
$PR_{\text{experimental}}$ [clk/px]	2.569	2.484	2.440	2.404	2.391
$PR_{\text{compensated}}$ [clk/px]	2.381	2.375	2.352	2.339	2.348
FPS [frame/s]	384	130	85	51	20.5
Registers	231	237	242	242	253
LUTs	761	853	859	859	1057
Block RAMs	7	13	13	13	26

means of pixel-delay between the input and output streams. The obtained speed-up suggests that the proposed parallelization method has a small overhead, e.g., for PD =6 the obtained experimental speed-up is equal to 5.532 out of possible 6. This overhead is caused by the synchronization of image partitions when the partition edge is encountered. By other words, if one partition takes a longer time to complete due to more complex contents, this partition may stall computation of other partitions slowing down the computation in average. Naturally, the higher the PD, the more partitions are to be synchronized and the greater overhead will degrade performance.

The FPGA area results in Table 6.4 are separated into 2 groups: the operator part (horizontal plus vertical part) and I/O buffers. The area of operator units in terms of registers and LUTs is proportional to PD as well because PD independent units are instantiated in a parallel manner. Although the overall vertical memory requirements (5-8) remain unaffected by PD, the number of occupied RAM blocks slightly increases in practice. That is caused by fixed BRAM memory geometry. It is apparent from the tables that high performance comes at not negligible resources cost. The area of input and output buffers is linear w.r.t. both N and PD since their essential components are line buffers (FIFO memories of N elements). Each one out of PD channels uses its own buffer.

The ultimate timing results (PD=6) versus the image size are listed in Table 6.5.

Table 6.3: Parallel rectangle: timing w.r.t. *PD*. SVGA image, SE = 31x31 square.

Parallelism degree *PD*	1	2	3	4	5	6
Latency [image line]	15	15	15	15	15	15
$PR_{\text{experimental}}$ [clk/px]	2.440	1.264	0.824	0.625	0.505	0.426
Experimental speed-up [-]	1	1.930	2.858	3.770	4.663	5.532

Table 6.4: Parallel rectangle: area w.r.t. *PD*. SVGA image, SE = 31x31 square.

Parallelism degree *PD*	1	2	3	4	5	6
Regs operator	242	650	978	1,280	1,605	1,938
LUTs operator	853	2,138	3,227	3,862	4,875	6,054
Block RAMs	13	13	14	14	18	21
Regs I/O buffers	0	661	969	1,279	1,587	1,896
LUTs I/O buffers	0	1,408	2,086	2,776	3,459	4,135

Table 6.5: Parallel rectangle: timing w.r.t. image size, PD = 6, SE = 31x31 square

Size of image	CIF	VGA	SVGA	XGA	SXGA	1080p
Latency [image line]	15	15	15	15	15	15
$PR_{\text{experimental}}$ [clk/px]	0.443	0.431	0.426	0.426	0.427	0.418
FPS [frame/s]	2075	724	472	290	174	113
Worst-case *FPS* [frame/s]	1915	640	411	246	151	96

It illustrates the high-end performance of the architecture allowing at least 96 fps with 1080p image size in the worst case. The worst-case performance is measured on an artificial saw-shaped image, which is the most unpleasant for the dequeuing step of the algorithm. Data dependency of the algorithm has been discussed in depth in Section 5.1.3. These frame rates remain constant for any morphological serial filter (such as ASF).

In conclusion, the tested rectangle dilation unit is a computation block that can be scaled by means of the image size, SE size, and parallelism degree in order to match the application requirements on the performance and FPGA device. The experimentally obtained results go from 384 fps on CIF images with small FPGA occupation up to 113 fps using 1080p resolution and a lot of FPGA resources (at 100 MHz system clock frequency).

6.2 Polygon Dilation Unit

Essential timing benchmarks of polygon unit PU and parallel polygon unit PPU with respect to the image size and the size of the SE are collected in Table 6.6 and 6.7. The $PR_{\text{experimental}}$ represents the measure of complexity $\mathcal{O}_{\text{image}}((N+W)(M+H))$ (recall that W, H are the projected width and height of L_l^α) and its value varies with $(H+W)/(N+M)$ fraction. In the case of polygons, the influence of the SE size is greater than in rectangles since the SE does not only imply the operator latency, but also the size of padding, which both slow down the computation. These effects are reflected in the FPS measure as well.

The speed-up of PPU against PU represents the acceleration obtained from the parallelization. The difference from the ideal upper limit (speed-up=PD) has two reasons: (i) the overlap of neighboring image partitions, which causes some redundant computation, and (ii) stream switching that needs inter-stream synchronization, which may introduce wait cycles in some partitions. The thickness of the overlap is equal to W (or $2B_{\text{H}}$), so the effect of overlap is proportional to W/N fraction. With increasing image size the acceleration converges towards 6 as the SE size (and consequently the overlap) becomes less significant with regard to the image size.

Table 6.6: Polygons: timing w.r.t. SE size (SVGA image)

SE size [px]	21	31	41	51	61
$PR_{\text{experimental}}$ [clk/px]	2.42	2.46	2.49	2.53	2.58
FPS [frame/s]	85	84	83	82	81
Latency [image line]	10	15	20	25	30

Table 6.7: Polygons: timing w.r.t. image size, SE size = 51 px, PD =6).

Image Size	VGA	SVGA	XGA	1080p
$PR_{\text{experimental}}$ (PU) [clk/px]	2.61	2.53	2.53	2.44
Latency [image line]	25	25	25	25
FPS (PU) [frame/s]	125	82	50	19
FPS (PPU) [frame/s]	599	406	257	105
Experimental speed-up [-]	4.79	4.94	5.1	5.34

Table 6.8 outlines the efficiency of the scalability (that is the parallelism degree PD) in terms of the FPS and speed-up. Not surprisingly, the observed speed-up is somewhat lower than PD. The difference is due to two factors: (i) the overlap, which demands redundant computation, and (ii) the stream switching that needs inter-stream synchronization which may introduce wait cycles.

Table 6.9 reveals the cost of parallelization on FPGA resources in terms of registers, LUTs, and BRAMs of the computational units PPU/PU and the pair of input and output buffers. The claim that the high performance comes at raised

Table 6.8: Polygons: speed-up w.r.t. *PD*, SVGA image, SE size = 31 px.

Parallelism degree *PD*	2	3	4	5	6
FPS [frame/s]	162	234	306	376	441
Experimental speed-up [-]	1.92	2.77	3.62	4.44	5.22

Table 6.9: Polygons: area w.r.t. *PD*, SVGA image, SE size = 91 px.

Parallelism degree *PD*	1	2	3	4	5	6
Registers (P)PU	787	1,644	2,469	3,215	4,019	4,850
LUTs (P)PU	2,656	4,831	7,330	9,301	11,540	14,221
Block RAM (P)PU	39	39	59	42	53	63
Registers I/O buffers	0	251	355	466	590	671
LUTs I/O buffers	0	1,296	1,929	2,545	3,158	3,748

resources cost holds true for polygons as well.

In conclusion, the tested polygon dilation unit is a computation block that can be scaled by means of the image size, SE size, and parallelism degree in order to match the performance and FPGA requirements. The experimentally obtained results go from low-end 125 fps on VGA images with small FPGA occupation up to 105 fps using 1080p resolution and a lot of FPGA resources (at 100 MHz system clock frequency).

6.3 1-D Synchronous Dilation Unit

The timing and FPGA resources benchmarks for the synchronous implementation of the 1-D oriented dilation are outlined in Tables 6.10 and 6.11. The size of the SE does not affect $PR_{\text{experimental}}$ much (there are minor boundary effects according to $\mathcal{O}_{\text{image}}((N+l_{\text{right}})(M+l_{\text{down}}))$), as all the comparisons necessary for a given SE are computed by parallel comparators. Thus, the FPGA resources in terms of registers, LUTs, and BRAMs are proportional to the size of the SE. Also the FPGA resources occupation is quite large in number, it is more than twice as large as occupation of the 2-D rectangular above. Furthermore, two 1-D synchronous units must be used to constitute a rectangle, so the difference is even larger in the 2-D case.

The influence of the image size to benchmarks is the following. The $PR_{\text{experimental}}$ slightly varies as the boundary effects fraction $(l_{\text{right}}+l_{\text{down}})/(M+N)$ changes, the FPS decreases proportionally to the image size. The FPGA resources increase mainly in BRAM memory occupation since longer delay lines are needed for larger images, and the number of comparators is not affected by the image size.

The tested 1-D synchronous dilation unit can be scaled by means of the image and SE size in order to match the performance and FPGA requirements of an application. The performance spans from 920 fps on CIF images up to 47.5 fps

Table 6.10: Oriented dilation unit: timing and area w.r.t. SE length l, $\alpha = 45°$, SVGA image size.

Length of SE	11	21	31	41	51
Latency [image line]	4	8	11	15	18
$PR_{\text{experimental}}$ [clk/px]	1.012	1.021	1.033	1.042	1.054
Registers	518	791	1059	1327	1592
LUTs	1651	1929	2266	2515	2816
Block RAMs	3	5	7	10	12

Table 6.11: Oriented dilation unit: timing and area w.r.t. image, SE l=31, $\alpha = 45°$

Size of image	CIF	VGA	SVGA	XGA	1080p
Latency [image line]	11	11	11	11	11
$PR_{\text{experimental}}$ [clk/px]	1.071	1.041	1.032	1.025	1.016
FPS [frame/s]	920	312	201	124	47.5
Registers	1045	1058	1059	1059	1071
LUTs	1993	2283	2266	2266	2843
Block RAMs	4	7	7	7	14

using 1080p resolution.

6.4 Opening and Spectrum Unit

In the following paragraphs we will focus on experimental results of the proposed opening and pattern spectrum unit implementing the *streaming peak elimination* algorithm. Table 6.12 contains timing and FPGA occupation figures for a diagonal SE ($\alpha = 45°$). As we already know, the latency measure is equal to operator latency, that is to the vertical projection of the SE times the width of the image. The pixel rate $PR_{\text{experimental}}$ follows the complexity $\mathcal{O}_{\text{image}}((N+W)(M+H))$ with H, W be the projected lengths of the SE $W = l\cos\alpha, H = l\sin\alpha$. The constant complexity is degraded by a factor of $(W + H)/(N + M)$. The deviation of the pixel rate may seem to be larger in value than in the case of the *Dokládal* algorithm for dilation. That is caused by longer lengths of the SE chosen for the opening benchmark. The effect of 255 px SE to 800×600 image is evident. The computed $PR_{\text{compensated}}$ eliminates the empty iterations inferred by latency and verifies $\mathcal{O}(1)$. We used formula (6-6) and slightly modified it to

$$PR_{\text{compensated}} = \frac{T_{\text{proc}} - 2(WM + HN + WH)}{M\,N} \qquad \text{[clk/px]}. \qquad (6\text{-}6)$$

The FPGA area occupation scales logarithmically with the SE size as expected in terms of both logic and block RAMs.

Table 6.12: Opening: timing and area w.r.t. SE, SVGA image, $\alpha = 45°$.

Length of SE	15	31	63	127	255
Latency [image lines]	10	21	44	89	180
$PR_{\text{experimental}}$ [clk/px]	2.350	2.453	2.602	2.900	3.485
$PR_{\text{compensated}}$ [clk/px]	2.287	2.323	2.334	2.343	2.298
FPS [frame/s]	88	85	80	72	60
Registers	321	350	395	473	473
LUTs	1612	1643	1734	1840	2052
Block RAM	6	13	28	60	128

Table 6.13: Opening: timing w.r.t. α, SVGA image, $l = 63$ px.

Orientation α [°]	0	15	30	45	60	75	90
FPS [frame/s]	84.6	81.7	80.2	79.4	79.35	79.8	81.2
Latency [image lines]	0	16	31	44	54	60	62

The last benchmark captured in Table 6.13 and in Fig. 6.1 concerns the relation of the FPS and the orientation of the SE α. The variation of FPS for different angles stems from the complexity $\mathcal{O}_{\text{image}}((N+W)(M+H))$ whence values of $W = l\cos\alpha, H = l\sin\alpha$ highly depend on α. The FPS deviation is also proportional to the length of the SE l, so the variance of FPS for $l = 31$ px is $\Delta FPS(l = 31) = 3.9$ frame/s and for $l = 63$ px $\Delta FPS(l = 63) = 5.4$ frame/s. The mean value of FPS drops down with l as the borders introduces more empty cycles, the number of which is $NH + MW + HW = Nl\sin\alpha + Ml\cos\alpha + l^2\sin\alpha\cos\alpha$.

6.5 Comparison of the Proposed Implementations

At this section we will compare the proposed architectures. The comparison of architectures scaled for the SVGA image and the SE that fits 31×31 bounding box are outlined in Table 6.14. This comparison is intended to collect the performance and FPGA area measures of all architectures with the same parameters that may better show their characteristics and the prospective use. However, we are aware of the fact that each architecture uses a different SE and it addresses another application field.

When considering rectangle and polygon units, we can see in Table 6.14 that they both achieve almost the same performance of 84–85 fps. Their LUTs and registers is however quite distinct, the polygon unit consumes almost three times more logic than rectangle. The reason is that the polygon unit contains 4 computation units instead of only 2 and dedicated blocks for the padding. In the case of parallel architectures with $PD = 6$, the FPGA resources consumption of the parallel polygon unit is twice as large for the same reasons. The performance of the parallel polygon unit (441 fps) is a little bit smaller than for parallel rectangle unit (472

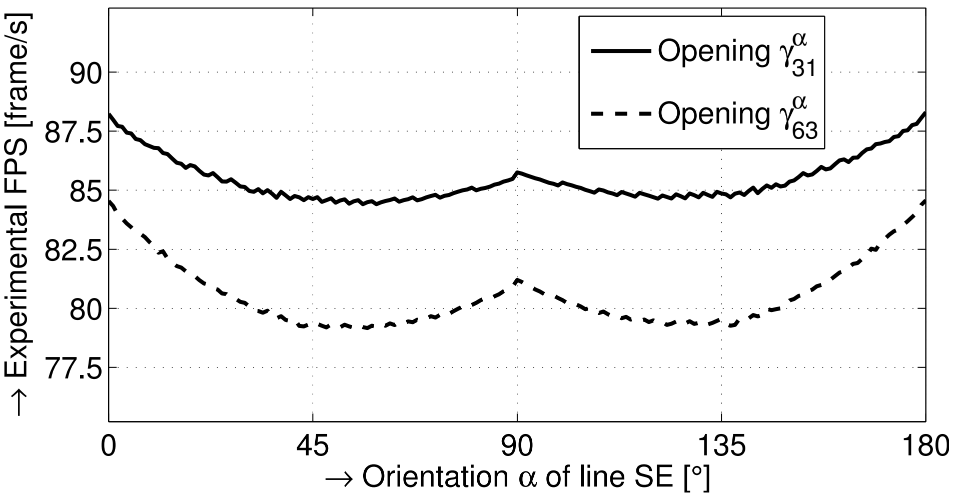


Figure 6.1: Performance of FPGA opening and *PS* block w.r.t. α, SVGA image.

Table 6.14: Comparison of the proposed architectures

	FPS [frame/s]	$PR_{\text{exper.}}$ [clk/px]	LUTs	Registers	BRAMs
Rectangle	85	2.44	859	242	13
Polygon	84	2.460	2,507	773	18
Parallel rectangle	472	0.426	6,054	1,938	21
Parallel polygon	441	0.474	13,070	4,402	27
1-D Synchronous	201	1.033	2,266	1,059	7
1-D Opening and *PS*	85	2.453	1,643	350	13

fps) that suggests that parallelization causes greater overhead for polygons. It has been expected since polygons need the padding and overlap issue to be handled.

The 1-D synchronous architecture may be compared with the rectangular unit (its results are mainly given by the vertical unit). The main advantage of the synchronous architecture is it higher performance 200 fps derived from $PR_{\text{exper.}} \approx 1$ whereas architectures using *Dokládal* algorithm have $PR_{\text{exper.}} \approx 2.45$. Also the memory requirements are in favor of the synchronous unit. Even though the dependence of the dilation on input pixels tells both architectures to store the same amount of pixels (HN in the horizontal scan, H is the projected height of the SE), the *Dokládal* algorithm needs more memory space to store one pixel because it stores the position of the pixel as well. On the other hand, the synchronous architecture consumes much more FPGA logic due to $l-1$ comparators are necessary for the l-pixel SE. The logic occupation issue becomes significant with large SEs.

The performance of the 1-D opening and pattern spectrum unit is equivalent to the rectangle and polygon units (non-parallel). The reason is similarity of both *Dokládal* and *streaming peak elimination* algorithms and their implementations based on the FSM and Array of queues. The comparison of BRAMs between opening and rectangle units shows that opening can be computed within the same memory space of 13 BRAMs as dilation (in the rectangle dilation only the 1-D vertical dilation needs BRAMs).

Bearing in mind that opening can be computed as a concatenation of erosion and dilation, two 1-D synchronous units may replace the opening unit. Two synchronous units would be faster, 201 fps instead of 85 fps, but their FPGA requirements would be much greater. Also the direct pattern spectrum computation would not be possible.

6.6 Comparison with Existing Implementations

At this point, we are interested in comparing the proposed architectures with other recently published hardware implementations of mathematical morphology. We include in the comparison the following architectures labeled hereafter by the name of the first author: [Clienti 2008a], [Chien 2005], [Déforges 2010]. We summarize a description of these solutions below; for more details see Chapter 3 or respective publications. This comparison presented here has a rather illustrative purpose because the respective designs are not entirely equivalent; they used different technologies and FPGAs. Unfortunately, we can only compare the figures communicated in the literature as the source codes are intellectual properties of the authors or laboratories.

Clienti proposed to use many neighborhood processors optimized for an arbitrary SE within a small bounding box 3×3 px, which are interconnected in a partially reconfigurable pipeline of 16 processors. Chien developed an ASIC chip for 5×5 disc SE providing good performance for this fixed SE. Both previous approaches utilize the homothecy to obtain larger SEs, which, however, limits the choice of resulting shapes of the SE and performance. On the other hand, one Déforges's unit supports various 8-convex SEs in one scan. As mentioned in the state of the art, the programmability of the modules, namely the possibility to control the SE shape after the synthesis and the FPGA occupation figures were not communicated, so we will have to adopt some assumptions on these issues in the later part.

Table 6.15 outlines the comparison of architectures chiefly in terms of the degree of parallelism, supported shape of the SE, computational performance, and processing rate. Looking at the performance column, we can observe that Clienti achieved the best throughput of 400 Mpx/s followed by our rectangular dilation, polygonal dilation, Chien, and Déforges. However, such a straightforward comparison may be misleading since both Clienti and Chien used only very small, elementary SEs in contrast to Déforges and us whose implementations support SEs of significantly larger sizes. Recall that the former implementations obtain large SEs from small SEs by homothecy, that is by multiple application of the given operation.

As for the 1-D opening and synchronous architectures, there is no implementa-

Table 6.15: Comparison of architectures for mathematical morphology

	Morpho. operator	Parallel degree	Supported SE	Perf. [Mpx/s]	f_{max} [MHz]	PR [clk/px]
Clienti	δ,ε	4	arbitrary 3×3	400	100	0.25
Chien	δ,ε	1	disk 5×5	190	200	1.052
Déforges	δ,ε	1	arbitrary 8-convex	50	50	1
Parallel rectangle	δ,ε	6	rectangle	234	100	0.426
Parallel polygon	δ,ε	6	polygon	218	100	0.467
1-D synchronous	δ,ε	1	oriented line	193	200	1.033
1-D opening	γ,φ,PS	1	oriented line	38.4	100	2.453

tion supporting the arbitrary-oriented line SE to our knowledge. Notice that none of the three aforementioned architectures is capable of attaining the oriented line SE. They only provide diverse non-rectangular, 8-convex shapes (convex according to the used 8-connectivity); but the oriented discrete line is not in general a convex shape and can not be obtained by composition from some elementary SEs using homothecy. However, we believe that the Déforges architecture can be modified in order to support oriented line SEs. By simple reconnecting the top-level entities *Memory module* and *Max extraction* (see [Déforges 2010]), we can move from the originally used 8-convex SE decomposition to naive implementation of line SEs at zero additional cost. This hypothetical architecture computes the 1-D oriented line dilation at the same pixel rate 1 clk/px like our 1-D synchronous architecture (the difference in the system frequency is probably given by used FPGAs).

The modified Déforges for line SEs can also compute the 1-D arbitrary-oriented opening by composition $\gamma_B = \delta_{\widehat{B}}\varepsilon_B$. This opening remains invariant to the translation, but on the other hand it does not behave correctly near image boundaries (see Fig. 4.14). The performance of such opening is obviously a half of the dilation performance because architecture must be used twice. So, comparing both feasible architectures, our design computes γ_L^α with moderate 1.14–1.7 speed-up depending on the length of the SE. For the sake of simplicity, we considered the performance of Déforges to be fixed at 25 Mpx/s, even thought it shall decrease due to the image boundaries as well. The comparison is much more favorable for us in the pattern spectrum case. Using the conventional way (2-30) that sums up a residue of two openings for each element of the PS, as much as $l_{\max}$ openings are needed to obtain the entire PS. As our architecture provides PS in a single image pass in the exactly same time, the speed-up is further multiplied by $l_{\max}\times$ factor. So, for instance, the speed-up for $PS(\alpha,:)$, $l_{\max}$=15 is equal to 27.2×, and for $l_{\max}$=255 the speed-up reaches 294×.

6.6.1 Comparison Using Alternating Sequential Filters

In the following paragraphs, we will compare the morphological implementations on alternating sequential filters (ASF). We have learned recently that Clienti and Chien proposals are especially optimized for small SEs and they use homothecy for large ones. As we and Déforges provide large SEs directly, it may be worth to compare the implementations using an application that involves both small and large SEs. Such an example of a widely used, but simple at the same time, application is an ASF.

From the introduction we know that a λ-order ASF (referred to as ASF$^\lambda$) is composed of the sequence of λ closings and λ openings with the increasing SE size, such as the s^{th} stage ($s \in \mathbb{N}; s \leq \lambda$) uses the SE of width $2s+1$.

$$\begin{aligned}\text{ASF}^\lambda &= \gamma^\lambda \varphi^\lambda \gamma^{\lambda-1} \varphi^{\lambda-1} \dots \gamma^1 \varphi^1 \\ &= \delta_{\widehat{B}_\lambda} \varepsilon_{B_\lambda} \varepsilon_{\widehat{B}_\lambda} \delta_{B_\lambda} \delta_{\widehat{B}_{\lambda-1}} \varepsilon_{B_{\lambda-1}} \dots \varepsilon_{\widehat{B}_1} \delta_{B_1}\end{aligned}$$

The initial number of morphological operators 4λ can be reduced using the associativity property of dilation and erosion. Hence, every two consecutive dilations or erosions may be merged into one to obtain only $2\lambda+1$ operators, such as

$$\text{ASF}^\lambda = \delta_{\widehat{B}_\lambda} \varepsilon_{B_\lambda \oplus \widehat{B}_\lambda} \delta_{B_\lambda \oplus \widehat{B}_{\lambda-1}} \dots \varepsilon_{B_1 \oplus \widehat{B}_1} \delta_{B_1}. \tag{6-7}$$

Hereafter, we consider as example an ASF of 6-th order, such as ASF^6 = $\delta_{13\times13} \varepsilon_{25\times25} \dots \varepsilon_{5\times5} \delta_{3\times3}$. This filter consists of 13 morphology operations in the reduced form.

Now let us focus again on the architectures in question and properties of the hardware systems they were proposed to comprise. The properties as well as the estimated performance are gathered in Table 6.16. In [Clienti 2008b] the authors published a hardware system containing 16 elementary 3×3 processors in a single FPGA. It can be simply computed that it will require 6 image scans to apply the entire ASF^6. [Chien 2005] described an ASIC chip containing one computation core leveraging the PRR principle, therefore, as much as 45 scans are to be performed. In the case of [Déforges 2010] where the proposed principle of SE decomposition was implemented as a computation unit, neither the FPGA occupation nor the possibility of using multiple instances in a single chip was communicated. Therefore, we consider two boundary cases: (a) only one unit fits the FPGA, so 13 image scans are needed; and (b) the entire ASF^6, i.e., 13 processing units, fits the FPGA, and a single image scan is sufficient. Our architecture using either rectangular or polygonal SEs with $PD = 6$ allows fitting of the entire ASF^6 resulting in one image scan as well.

From the estimated performance results for the ASF^6 in Table 6.16 we observe that the high use of homothecy tends to increase the number of necessary image scans. Indeed, all Clienti, Chien, and Déforges (a) whose solutions are efficient for

Table 6.16: Comparison of hardware systems on ASF

	Hardware System		Application Example ASF^6		
	Number of units	Supported image	Image scans	Perf. [Mpx/s]	Latency [im. lines]
Clienti	16	1024×1024	6	66.7	$5M + 84$
Chien	1	720×480	45	4.22	$44M + 84$
Déforges (a)	1	512×512	13	3.8	$12M + 84$
Déforges (b)	13	512×512	1	50	84
Parallel rectangle	13	1024×1024	1	213	84
Parallel polygon	13	1024×1024	1	185	84

small SE sizes and short concatenations become more or less penalized for longer concatenations; their performances drop down with the higher numbers of necessary image scans. On the other hand, Déforges (b) and our work, which does not need more than one image scan, attain the high performance for ASF^6 comparable to the performance of a single computation unit.

In addition, a large number of image scans has negative influence on latency and the design of the hardware system in general. Between two consecutive scans the data are read/written from/into the memory that significantly increases latency by the entire frame with each additional image scan. The image storage also involves a hard requirement on the hardware system design because some off-chip memory is to be used to accommodate the intermediate result image. The dense memory traffic to/from the off-chip memory might presents a complication in course of the design of an application platform.

6.7 Conclusions

This chapter dealt with the FPGA prototype results of the proposed architectures. We reported the timing measures, such as latency, experimental and compensated pixel rate, FPS throughput, etc., and the FPGA resources measures in terms of LUT, register, and BRAM consumption. This set of measures was performed for each architecture with different values of the programmable parameters. So we have observed the scalability of the architectures with respect to the varying size of the image and the SE. In the case of FPGA resources measures, the current value of the programmable parameters was considered to be the upper bound that scales the FPGA implementation (obviously changing a programmable parameter of a processing unit has no effects on implementation results).

We can see from the benchmarks that all the proposed architectures have some common properties. First of all, the pixel rate is almost independent of the size of the SE that is very beneficial for both serial and parallel interconnection of the units. For instance, when we use a pipe of many operators with different SE sizes, they are all computed at the same high pixel rate. The operator with a large SE does not slow down (or even stall) the operator with a small SE. Also the minimal latency mainly given by operator latency eases attaining the high performance of the whole application.

The ASF^6 comparison suggests that for long concatenations and large SEs our architectures achieve better throughput than any other hardware architecture to date. It is caused by avoiding the use of homothecy to obtain large SEs, so a large SE does not inevitably degrade the speed via many image scans, and thanks to the efficient $\mathcal{O}(1)$ algorithm the small FPGA occupation allows instantiation of many computation units in an FPGA. We have implemented a set of programmable IP blocks usable in a large scale of industrial systems running under severe timing constraints satisfying up to 100 MHz 1080p FullHD requirements.

7 Applications

Contents

In this chapter we present two applications of the proposed processing units. The main purposes are to provide examples of applications that beneficially take advantage of implementation in dedicated hardware, and also to prove that the proposed processing units are useful in applications of different context and complexity.

The first application deals with integration of the processing units into the multi-purpose platform called FREIA (FRamework for Embedded Image Applications [FREIA 2011]). The goal of the FREIA platform addresses the most computation performance demanding applications by proposing a platform that leverages multiple architectures of image processing.

The second application ([Bartovsky 2011c]) comes from the other group of applications suitable for dedicated hardware—low-power embedded application. In this case, we propose a low-complexity architecture for the three-class classification of particles recorded by the Timepix detector based on the shape of particle traces.

7.1 FREIA Platform

The FREIA project intends to improve different image processing accelerator architectures so as to address a larger set of applications and to support application portability to future accelerators. The expected results of the FREIA project are a new image processing platform based on a common interface for the improved image processing accelerator architectures, implemented in an FPGA, to reduce the application development cost by hiding the target architecture without sacrificing performance. The implemented architectures share common accelerator interfaces to make application portable and to make partial dynamic reconfiguration of the accelerator possible. The second goal of the FREIA project is to deliver optimization tools that cope with these architectures.

The main objective of this application is to integrate the processing units proposed in the manuscript into the generic FREIA platform, which already contains two accelerators, namely TER@PIX ([Bonnot 2008]) and SPoC ([Clienti 2008a]). The TER@PIX SIMD accelerator contains a large number of processing elements, each of which access a small window, and therefore, it is considered to be the middle-grained architecture. The SPoC pipeline of elementary neighborhood processors applies the given operator on the whole image, hence it is the coarse-grained architecture. Pixel level architectures would be fine grained in the FREIA terminology. Our proposed processing units are also coarse-grained, but provides morphological operations by large neighborhoods that brings a significant performance gain for the applications using large neighborhoods over the other technologies of FREIA.

7.1.1 Top-level Platform Description

From the top-level viewpoint, two slightly different platforms have been designed for FREIA. The main difference between them is the way how image data are transferred between processing units and the DDR memory that serves as an image storage. The first platform uses central DMA peripheral that issues DMA transfers via the main PLB (Peripheral Local Bus). The second platform uses dedicated VFBC (Video Frame Buffer Controller) channel. Since the latter achieved faster image transfers by order of magnitude, we will focus only on the VFBC case hereafter.

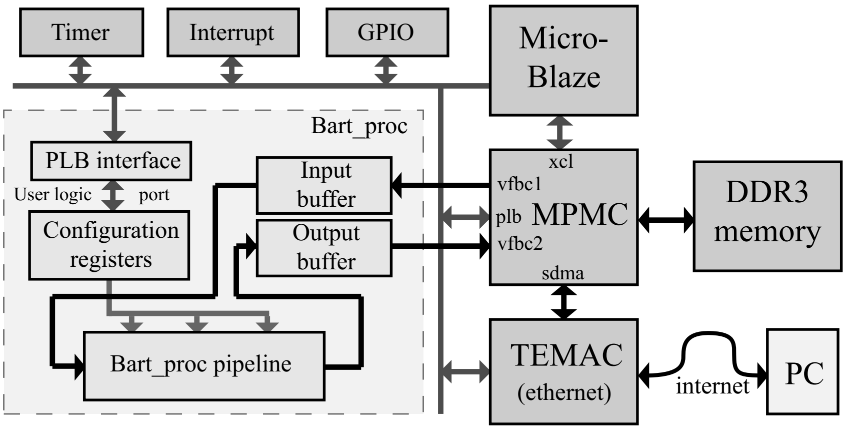


Figure 7.1: Top-level platform architecture. Black lines denote image data transfers, gray lines denote configuration and control.

The top-level platform architecture is displayed in Fig. 7.1. The platform consists of two main parts: (i) peripheral containing a couple of the proposed processing units called Bart_proc (highlighted by a grey rectangle and further described in Section 7.1.2 below), and (ii) the embedded MicroBlaze processor environment, the purpose of which is to provide the Bart_proc peripheral with proper configuration and image data. The MicroBlaze environment is composed of the following units:

- MicroBlaze processor: is an embedded processor soft core reduced instruction set computer (RISC) optimized for implementation in Xilinx FPGAs. It executes the application code which is stored in the DDR memory (access via MPMC by XCL dedicated bus). It handles TEMAC ethernet data transfers, sets up and controls Timer, Interrupt controller, GPIO, DMA, as well as it writes configuration of Bart_proc into Configuration registers and controls its behavior. It does neither access nor change image data.
- MPMC (Multi-port Memory Controller): provides various means of access to the off-chip DDR memory, such as XCL for MicroBlaze, PLB for peripherals, SDMA for TEMAC, or VFBC for Bart_proc.
- TEMAC (Tri-Mode Ethernet Media Access Controller): provides a control interface and registers for a hard silicon Ethernet MAC core. The internal control registers are accessible via PLB bus, whereas the received/sent data are transferred by LocalLink bus to the SDMA port of the MPMC. Its functionality is managed by lightweight TCP/IP protocol stack running on MicroBlaze.
- GPIO (General-purpose Input/output): provides a simple visual aid feature via LEDs.
- Interrupt controller: handles interrupt requests from Timer and TEMAC.
- Timer: provides real-time measuring feature and defines timing for TEMAC.

For any image processing platform, managing and transferring image data is an usual challenge due to an image contains a large amount of data. Let us see how the FREIA platform handles images. Paths used for image data transfers are shown as black lines in Fig. 7.1. Prior to any computation, an image to process is transferred from a personal computer (PC) to the FPGA via the internet (TCP/IP protocol using TEMAC core and lightweight TCP/IP stack). The received image is stored in the DDR.

Then the image is transferred from the DDR memory into the Bart_proc and backward via a pair of dedicated VFBC channels. VFBC is a feature of the MPMC, which allows us to read and/or write image data (in general any sequential data) to the DDR at very high speed. The VFBC standard provides the FIFO-like dataflow control (full, almost full, empty, almost empty flags), so the data stream can be stalled by either endpoint if necessary. On the side of Bart_proc the image is buffered by a pair of buffers. Their main purpose is to balance the unlike throughput and data bus widths of both units (Bart_proc uses 1-pixel data bus, i.e., chiefly 8-bit, whereas VFBC uses data bus 32 bits wide), and to allow the MPMC to transfer data in bursts. Burst transfers are more favorable for avoiding congestion of the DDR memory.

7.1.2 Bart_proc Peripherals

From the MicroBlaze viewpoint, the Bart_proc is a peripheral connected to the PLB bus. In order to comply with PLB requirements and constraints, the Bart_proc is interfaced via the standard PLB interface IP block provided by Xilinx, which translates read/write PLB transaction (or DMA bursts) to user-logic register or memory accesses. Then Configuration registers are accessible on respective addresses for PLB masters (MicroBlaze and DMA if used). As mentioned before, the configuration is written/read by MicroBlaze and image data via VFBC channels. The image transmission is full-duplex as we use two half-duplex VFBC channels.

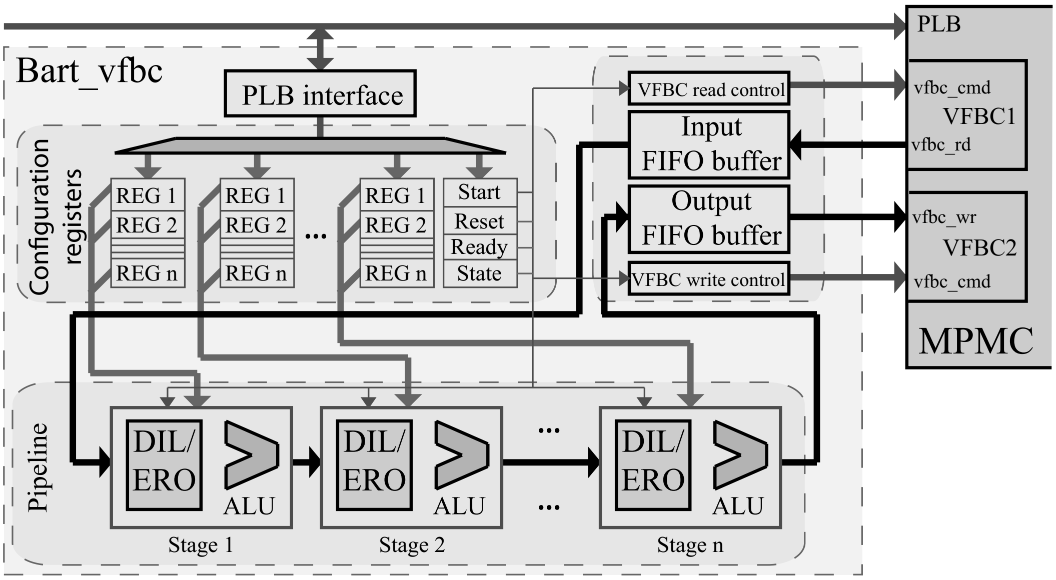


Figure 7.2: Detailed view of Bart_proc architecture. Black lines denote an image data bus, gray lines denote configuration and control.

The Bart_proc contains three parts: (i) configuration registers, data buffers, and a processing pipeline. The configuration registers is a set of registers that store the necessary configuration for all the processing units and for the control purpose. Notice that there is one bank of registers for each stage of the processing pipeline and that the values of registers are directly connected to the processing pipeline. So programming configuration registers takes effect immediately. In order to facilitate the process of programming the registers, incoming values can be broadcasted among all banks. The values of control registers, such as Start, Reset, etc., are distributed among all units.

The processing pipeline is a sequence of elementary stages each of which possesses one DIL/ERO unit and ALU. For further information about the particular interconnection and implementation see Section 7.1.3 below. The pipeline accepts one data stream (8-bit data bus, 1-bit acknowledge, 1-bit FIFO full) at the input port, applies a sequence of morphological operations according to the content of the configuration registers, and provides one data stream (of the same data-width) at the output port.

The image data are transferred to and from memory via a pair of VFBC channels. One VFBC channel consists of three FIFO-like (hence simplex) buses, read, write, and command. The command bus is used to set up the VFBC channel, i.e., the desired direction, the address and size of the image, etc. Once the channel is configured, the MPMC manages the image transfer using either bus, thus in a half-duplex way. For this reason, two parallel VFBC channels are used to achieve the full-duplex communication. The VFBC read control and VFBC write control blocks write the configuration to respective VFBC channels.

The following Bart_proc peripherals were implemented. They differ only in used architecture for DIL/ERO processing units:

- Rectangular dilation architecture, see Section 5.2.
- Rectangular parallel dilation architecture, see Section 5.2.1.
- Polygonal dilation architecture, see Section 5.3.
- Polygonal parallel dilation architecture, see Section 5.3.3.
- 1-D naive dilation architecture, see Section 5.4.

Hereafter, we restrict our description on the rectangular parallel architecture only.

7.1.3 Bart_proc Pipeline

The Bart_proc pipeline (see Fig. 7.3) is a core part that performs the given morphological computation. It consists of several equal processing stages connected one after each other into a pipeline. The heart of each stage is the DIL/ERO morphological unit that computes dilation or erosion by a given SE. The input data to this unit are selected by MUX IN (input multiplexer) from either input image or output of the preceding stage. The output of DIL/ERO unit is connected to ALU, MUX OUT (output multiplexer) and MUX GL (global output multiplexer). The MUX OUT selects the proper output of a stage among DIL/ERO and ALU. ALU's second operand is the output of previous DIL/ERO unit. ALU also measures a sum of the entire image that can be useful for granulometry. The select configuration for all the multiplexers is stored in the configuration registers. Note that each interconnection link is composed of 8-bit data bus, 1-bit data acknowledgement, and the backward 1-bit FIFO full flag, which ensures no data loss when any FIFO is full.

DIL/ERO performs morphological dilation or erosion by flat SE of programmable size (maximum size depends on used architecture) and position of the origin. The DIL/ERO contains balancing FIFO at the input and controls data-flow via fifo_full signal, so data streams may be stalled if necessary. The dilation/erosion computation can be turned off by bypass feature. Then the computation memory changes into a large FIFO that can be used to balance dataflows in certain operations, such as top-hat, gradient, and so forth.

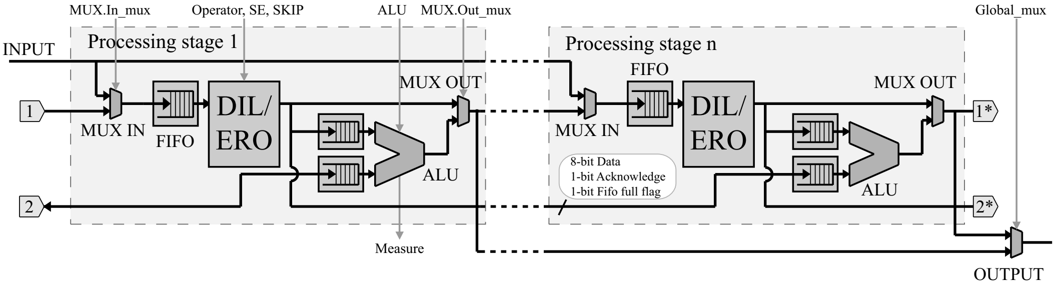


Figure 7.3: Bart_proc pipeline architecture.

ALU (Arithmetic Logic Unit) performs simple arithmetic operations of two operands, each of which can be configured as either ALU input signals or constant. The supported operations are as follows: no operation; bit-wise logic NOT, AND, OR, XOR; saturated addition and subtraction; maximum and minimum. The operand select, operation code, and the value of constant are stored in the configuration registers. The ALU contains balancing FIFO at each input and controls dataflow via fifo_full flag in the same manner like DIL/ERO that ensures no data loss in the pipelined communication.

7.1.4 FREIA Interface

The FREIA platform on the FPGA board is connected with the PC via 100-Mbit ethernet to transfer the image data and configuration. The FREIA can work as either a server or a client. The main difference is which part decides what application is to be done, and consequently defines the configuration.

In the client mode, the FREIA platform itself runs the application that has to coded as a function called by the MicroBlaze processor. The images are read from the pc-side image server (PC-IS) and the result image is sent back to the PC-IS. The PC-IS is a simple C-code server that only listens to image send or receive requests and handles image transfers. Since the application is completely managed by the Microblaze, the configuration is incorporated in its code that has to be reprogrammed every time any modification to the application is to be applied. By other words, the FREIA computation is not controllable until it is stopped, reprogrammed, and started again.

In the server mode, the FREIA platform works as a server. This server listens to the incoming connections. When some PC-side client connects to the server, the client determines what action is to be done and sends a proper instruction. The most commonly used instructions are: write an input image, read an output image, write the Bart_proc configuration registers and initiate processing. Then the whole application to run (operation, size of the SE, images) is programmed in the pc-side client, the FREIA server only passes the configuration created in the client to the Bart_proc peripheral, and reads and writes images to the DDR memory.

The pc-side client is a counterpart to the FREIA server. The pc-client defines the whole application and image transfers. The pc-client application may proceed in the following fashion:

- Send an input image from the PC to the FREIA platform.

  ```
  put_image(SOCKET, InputImage);
  ```

- Set the morphological function. Now we configure stages of the processing pipeline by the desired morphological operation and the size of the SE. In the example below, we set the first stage to dilation, and the second stage to erosion, both by predefined SE1 size of the SE.

  ```
  dilation(&HW, CONFIGURATION, 1, SE1);
  erosion(&HW, CONFIGURATION, 2, SE1);
  ```

- Send the configuration and initiate the computation.

  ```
  send_configuration(SOCKET, CONFIGURATION, &HW);
  ```

- Receive the output image from the FREIA platform.

  ```
  get_image(SOCKET, OutputImage);
  ```

In such a way, the whole application to be executed in the FREIA platform is created using a simple C-code library on the PC side. This really simplifies the usage of the FREIA hardware accelerator for application engineers since it hides the complex hardware concerns behind a couple of library calls.

7.1.5 FREIA Performance Evaluation

In this section the performance results of the Bart_proc integrated in FREIA platform are evaluated. Tables 7.1 and 7.2 overview the most important parameters and performance results of the respective benchmarks for both server and client modes. We used different image sizes and degrees of parallelism (*PD*, see Section 5.2.1) and measured the time needed for an image transfer from PC to DDR memory via ethernet and the time consumed by processing the image, which is read from the DDR, and the result is stored in the DDR, too. The operations used in benchmarks are erosion and dilation by SE up to 61×61, closing and opening by the SE up to 31×31, and gradient by the SE up to 31×31. According to constant complexity of the algorithm and its implementation, the performance does not change with respect to the size of the SE, but slightly, randomly varies due to DDR memory accesses. The platform has been targeted to Virtex-5 XC5VSX95T-2 FPGA at clock frequency 125 MHz.

In the case of using DMA instead of VFBC, the description of which we have omitted, the processing throughput is saturated at 16.7 Mpx due to the DMA transfers take place on the shared PLB peripheral bus. The performance of this platform

Table 7.1: Performance benchmarks of client mode

SE	Type of platform	Supported Image	Parallel degree	Time for tr. PC->DDR	Time for processing	Processing throughput
Rectangle	DMA	512×512	1	176 ms	15 ms	16.7 Mpx/s
Rectangle	DMA	512×512	2	176 ms	15 ms	16.7 Mpx/s
Rectangle	VBFC	1024×768	1	301 ms	15.3 ms	51.4 Mpx/s
Rectangle	VBFC	512×512	2	176 ms	2.7 ms	92.6 Mpx/s
Rectangle	VBFC	1920×1080	6	1060 ms	7.5 ms	276 Mpx/s

Table 7.2: Performance benchmarks of server mode (all VFBC)

SE	Supported Image	Parallel degree	Time for tr. PC->DDR	Time for processing	Processing throughput
Rectangle	1024×768	1	324 ms	17 ms	46.2 Mpx/s
Rectangle	1920×1080	6	981 ms	8.5 ms	244 Mpx/s
1-D orient.	896×672	1	318 ms	6.8 ms	88.5 Mpx/s

Table 7.3: Comparison with other FREIA architectures

Architecture	Frequency	Gradient perf.	Pixel rate
SPoC	300 MHz	291 Mpx/s	1.03 clk/px
TER@PIX	150 MHz	257 Mpx/s	0.78 clk/px
Bart_proc	125 MHz	276 Mpx/s	0.45 clk/px

is below a half of performance of a stand-alone processing unit and obviously are not affected by the parallel degree.

The platform using VFBC is capable of exploiting the whole computational power of the Bart_proc. The ultimate benchmark for parallel degree 6 achieves 276 Mpx/s in the client mode and 244 Mpx/s in the server mode that conforms to 133 (117) FullHD frames per second. However, high performance of the Bart_proc is deteriorated by a slow ethernet image transfer.

As it can be seen from the table, the time for an image transfer is in orders of magnitude greater then the time consumed by processing, and is therefore the main bottleneck of these platforms. It is caused by the relatively slow lightweight software TCP/IP stack, which runs on MicroBlaze. In the future work, this TCP/IP stack shall be replaced by other means of suitable communication standard, e.g., ad-hoc UDP hardware stack, or PCI-Express bus.

Table 7.3 outlines the comparison of the proposed Bart_proc accelerator against the two other architectures already implemented for the FREIA platform: TER@PIX ([Bonnot 2008]) and SPoC ([Clienti 2008a]). Both these architectures supports also convolution, correlation, geodesic reconstruction along with mathematical morphology. We used the gradient operator by 3×3 SE for the comparison since it is supported by all three architectures without performance deterioration due to multiple image scans. The result shows that all three architectures achieves comparable performance, however, our Bart_proc needs lower frequency, which is beneficial in the context of low-power embedded devices.

7.2 Classification of Particles Recorded by the Timepix Detector

In this application (published in [Bartovsky 2011c]) we proposed an image processing approach to the classification of particles recorded by Timepix based on the shape of traces, using only a few basic morphological operations. This method implemented in an FPGA achieves performance and latency allowing for high acquisition rate. When embedded with Timepix, it can beneficially analyze radioactive fluxes of unknown sources and spectra.

The Timepix device [Llopart 2007] is a new generation of CMOS pixel detectors usable in a large scale of applications; from astronomical observations, X-ray fluorescence imaging to event reconstruction in physical numerous experiments (i.e., analyze radioactive fluxes from unknown radioactive sources) [Jakubek 2011].

The particles recognition requires to identify and analyze the *trace*, representing the particle's "signature", left by the particle whenever it strikes the Timepix detector. The different particles leave differently shaped traces in dependence on the type of the particle, its energy and incidence angle. Consequently, the shape and the energy deposited alongside every track can be used for identification of the particle [Bouchami 2011]. The goal of this application is to propose a classification method for automated event observations. Clearly, such a method should be scalable with respect to particle classes as well as its implementation should be very fast.

7.2.1 Classification Using Morphological Characteristics

The Timepix device records a sequence of gray-valued images $I\colon\ D \times t \to V$. The support $D \subset \mathbb{Z}^2$ is a rectangular 256×256 raster. The images are scalar-valued with the set of values V coded in 14 bits with positive integer values from [0, 16383]. In the following, one *cluster* denotes a connected component of non-zero pixels. One cluster corresponds to the trace left by one particle (or more particles, if they overlap). In this work, though, we suppose that one cluster is left by only one particle.

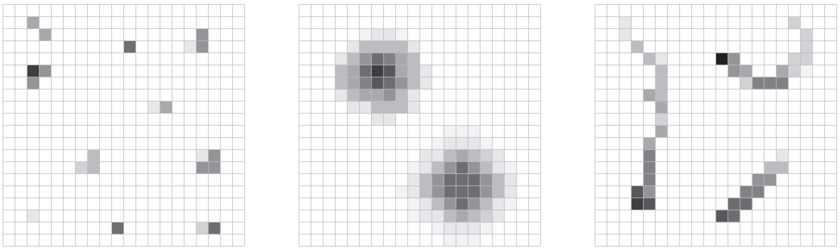
The set of all traces observed at time t is defined as $CC\{(x, y) \mid I(x, y, t) > 0\}$, where CC denotes the connected components in a set obtained with 8-connectivity. Each connected component can be associated with descriptors allowing to classify the particles into different classes. Examples of such descriptors are the area, the projected and unrolled length, the skeleton, the geodesic diameter, the circularity, the tortuosity, etc., see [Soille 2003].

The descriptor-based classification methods are very precise; their drawback though is the computing complexity. They require the computation of connected components, labeling, skeletonization and reconstruction, even before the descriptors can be computed. Even if optimized [Matas 2008], the skeletonization and reconstruction are iterative, with data-dependent computational intensity. Such properties infer high memory requirements, undefined latency, and slow computation inapplicable in high-frame-rate applications.

It is clear that the efficiency of the image processing bounds the sampling frequency of the image acquisition. At the same time, a high sampling frequency limits the probability of overlapping traces. Therefore, we present a fast classification method based only on two descriptors: thickness and projected length. These descriptors are composed of the morphological dilation, erosion, and simple arithmetic operators, thereby avoiding all iterative, costly algorithms. We propose a modular and programmable hardware implementation, too.

7.2.2 Method Description

In this study, we consider three main classes of traces called blobs, dots and tracks, see Fig. 7.4. These names correspond to the nuclear physics terminology used in [Bouchami 2011] and [Jakubek 2011]. The dots are generated by, e.g., low-energy electrons or photons. The blobs are left by α or heavy ions. And the linear or curly tracks are produced by minimum ionizing particles or electrons.



(a) dots (b) blobs (c) tracks

Figure 7.4: Examples of traces deposited by different particles.

7.2.2.1 Residual Approach to Particle Classification

Consider a family of shapes Ξ and an image I^{Ξ} containing objects from Ξ. The shape $\xi_i \in \Xi$ can be extracted from I^{Ξ} by opening γ_{ξ_i}

$$I^{\Xi'} = \gamma_{\xi_i} I^{\Xi} \tag{7-1}$$

where $\Xi' = \Xi \setminus \{\xi_i\}$, and $I^{\Xi'} = I^{\Xi} - I_i^{\xi}$.

This type of opening is commonly considered as algebraic opening. If Ξ is ordered, the shapes $\{\xi_i\}$ can be extracted one by one. This approach proceeds in a few steps each of which recognizes and extracts one type of particles retaining the other particles intact in the residual image. The following step extracts another type of particles and so on.

The algebraic opening γ_{ξ_i} from (7-1) can be constructed by morphological opening by reconstruction using the following steps:

1. Marker selection. It selects particles according to some criterion. A marker image $m : \mathbb{Z}^2 \to \mathbb{R}$ is commonly an image containing non-zero values intersecting the marked objects, and zero elsewhere. In the following, m^{ξ_i} will be used to mark objects of the shape ξ_i.

2. Object reconstruction recovers from the marker m the original values and shape from f. It is based, in general, on the geodesic dilation of m under f, $m < f$,
$$\delta^f(m) = \delta(m) \wedge f \tag{7-2}$$
hence from, by iteration
$$(\delta^f)^n(.) = \delta^f[(\delta^f)^{n-1}(.)] \tag{7-3}$$
we obtain the reconstruction
$$\mathcal{R}^f(m) = \lim_{n\to\infty} (\delta^f)^n(m) \tag{7-4}$$

Here we have a family of shapes $\Xi = \{\alpha, \gamma, \varepsilon\}$. The process of separation based on a cascade of openings is a binary decision tree classifier, see Fig. 7.5. First, we extract the thick dots (alpha particles, referred to as A), second, the thin tracks (electrons, referred to as E). Finally, the last residual image will contain the dots (gamma particles, referred to as G) only.

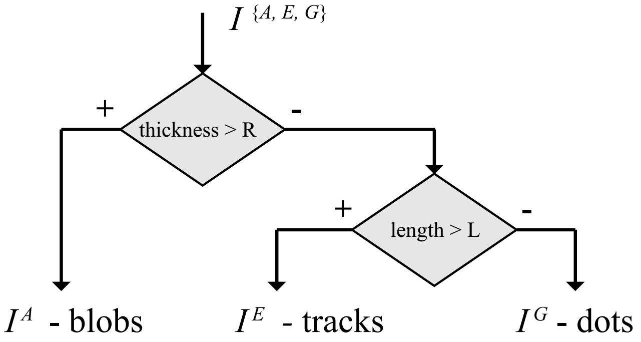


Figure 7.5: Flowchart of residual approach. $I^{\{A,E,G\}}$ denotes the input image, and I^A, I^E and I^G the result images.

The reconstruction is an iterative process based on the geodesic dilation (7-2) with unitary geodesic ball as structuring element. We will show that these stages can be approximated by a concatenation of basic morphological operators, erosion/dilation, and simple arithmetical operations. It can be computed only in one scan of the input image and "on the fly" without intermediate memory.

7.2.2.2 Method Implementation

As indicated above, the extraction of a shape ξ_i is done by algebraic opening γ_{ξ_i}, constructed by morphological opening by reconstruction. Recall that the reconstruction is iterative process, iterated until idempotence. Given the restricted and known family of shapes, we can approximate the reconstruction by only one geodesic dilation. Hence, the stages consist of the following steps.

1. Extraction of the marker of shape ξ by a morphological opening
$$m^A = \gamma_{B^A} I \tag{7-5}$$

2. Geodesic dilation (approximating the reconstruction) of the marker under the image I
$$m' = \delta_B^I(m^\xi) \tag{7-6}$$

3. Extraction of the image I^ξ containing the ξ-shaped objects
$$I^\xi = \begin{cases} I & \text{if } m' > 0 \\ 0 & \text{elsewhere} \end{cases} \tag{7-7}$$

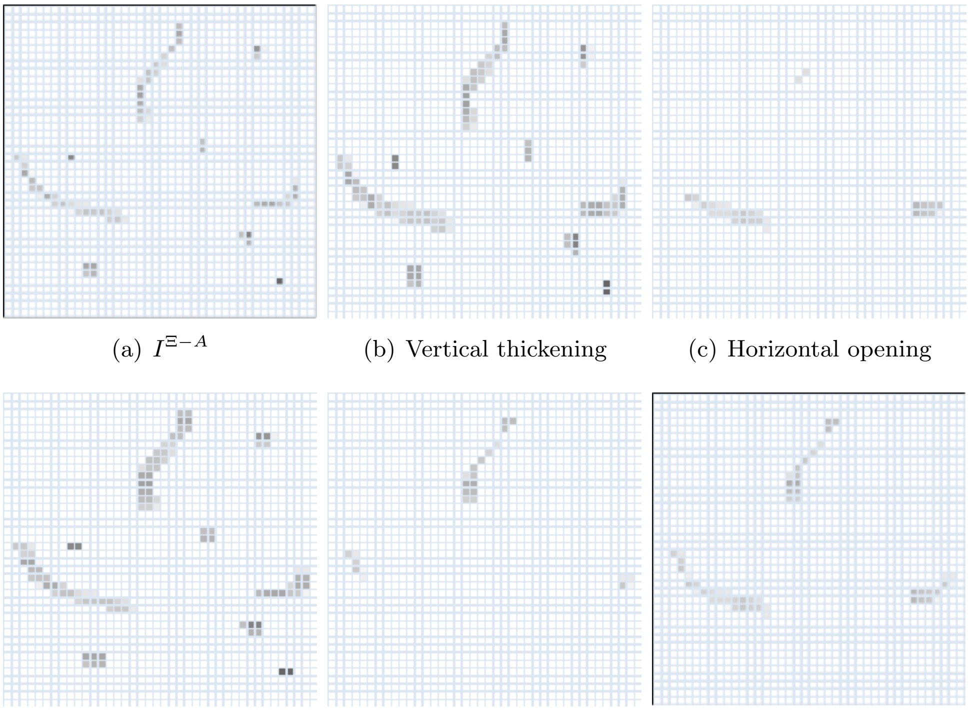
Based on this scheme, the particle classification is done in the following order. Refer to Table 7.4 for parameters of the structuring elements. The reconstruction step uses alike structuring element B for both shapes α and ε.

1. *Blobs* - First we obtain I^A from the initial image $I^{\{A,E,G\}}$. The residual image is $I^{\{E,G\}} = I^{\{A,E,G\}} - I^A$.

2. *Tracks* - Second, we obtain I^E. The usual morphological approach to detect curvilinear objects is to use the supremum of openings γ_{B^φ} by a rotating linear segment B^φ, oriented in φ. It is well known that a supremum of openings is also an opening.
$$\gamma = \bigvee_{\varphi \in \Phi} \gamma_{B_\varphi} \tag{7-8}$$
The tracks are thin, curvilinear, oriented in arbitrary angle. This requires a fine angular sampling of Φ resulting in a high computational cost.

   Here, to limit the number of discrete angles $\varphi \in \Phi$, we thicken the tracks by a dilation perpendicular to the opening. This allows to obtain satisfactory results with only two discrete angles, horizontal and vertical $\Phi = \{H, V\}$ (see Fig. 7.6 for illustration). Hence, using (7-8) for γ_{B^ξ} in (7-5), with $\xi = E$, we obtain
$$m^E = \bigvee_{\varphi = H,V} \varepsilon_{B^E_\varphi} \delta_{B'_\varphi} I^{\{E,G\}} \tag{7-9}$$
where $B_V = \mathrm{rot}(B_H)$, the copy of H rotated by 90^o, for both B^E and B'.

3. *Dots* - Finally, the residuum image $I^G = I^{\{E,G\}} - I^E$ contains the dots, i.e., the gamma particles.



(a) $I^{\Xi-A}$ (b) Vertical thickening (c) Horizontal opening

(d) Horizontal thickening (e) Vertical opening (f) m^E

Figure 7.6: Illustration of curvilinear objects detection in $I^{\Xi-\alpha}$.

7.2.2.3 Experimental Results

To evaluate the proposed classification method, we have performed a statistical measuring of the traces' dimensions on randomly selected images of the Timepix database. The results suggest that the diameter of blob traces is at least 4 px. Therefore, the thickness criterion R equal to 4 (accords to B_α=$[4,4]$) identifies the blobs.

On the other hand, the dot traces fit inside 2×2 bounding box. So the length parameter L equal to 3 (see B_H=$[1,3]$) separates tracks from dots. The approx-

Table 7.4: Structuring element parameters.

Class	Blobs	Tracks	
	A	E	
Marker selection	B^A	B^E_H	B'_H
	[4,4]	[1,3]	[2,1]
Approximation of reconstruction	B		
	[3,3]		

[H,W] denote the height and the width of a rectangular structuring element.

imated reconstruction uses in both cases the SE of the marker's erosion plus one pixel in all directions. Such a SE has the minimal surface necessary for the proper recovery of the original shape (cf. Table 7.4).

The computed confusion matrix, see Table 7.5, allows to appreciate the performance. The resulting errors are mainly due to: i) the border effects: particles touching the image border are sometimes misclassified, ii) the limit cases: the proposed method only approximates (with rectangular SE) the measurements of the particle trace thickness and projected length. The result misclassification of the method is below 7% of particles (each type of particles considered separately). Notice that this error remains within the error interval of much more sophisticated methods implemented in [Holy 2006], [Bouchami 2011] and optimized for Medipix/Timepix data.

Table 7.5: Confusion matrix computed for 100 images randomly selected from the database.

Input class Number of particles	Blobs 418	Tracks 4627	Dots 12906
431 blobs classified as	418	13	0
4920 tracks classified as	0	4614	306
12600 dots classified as	0	0	12600

Notice that the proposed method can be used to further analyze the three main classes by splitting them into sub-classes. The sub-classes are defined by the purpose of the particular physical measurement. We can illustrate this idea on the example of sorting the blobs with respect to their thickness. It requires to apply several consecutive blobs classification procedure with varying R. Another example could be rough sorting of the track impact angles. The principle is to refine the angular sampling of Φ in (7-8).

7.2.3 Hardware Architecture

The overall architecture of the proposed particle classification is displayed in Fig. 7.7. It consists of a several Recognition Units (RU), Control Unit, and optional Visualization Memory if results are to be displayed. The classification of a given type of particles is carried out in one RU block. The RU performs three tasks as described in the previous section: (i) the marker creation, (ii) the reconstruction, and (iii) the residual image. The RU outputs two images, I^{ξ} containing classified particles, and the residual image $I^{\Xi'}$ containing other particles.

In applications that need more types of particles to be recognized multiple RUs are instantiated in a pipeline (Fig. 7.7). It allows us to classify all types of particles concurrently on time-shifted data, thus using inter-operator parallelism. The residual image of an RU is taken as an input by the following RU.

The control unit provides both controls and programmable parameters for each

RU. The classified particles I^{ξ_i} of any RU can be either read by a further block (RU, output, image compression, etc.) or stored in the global visualization memory.

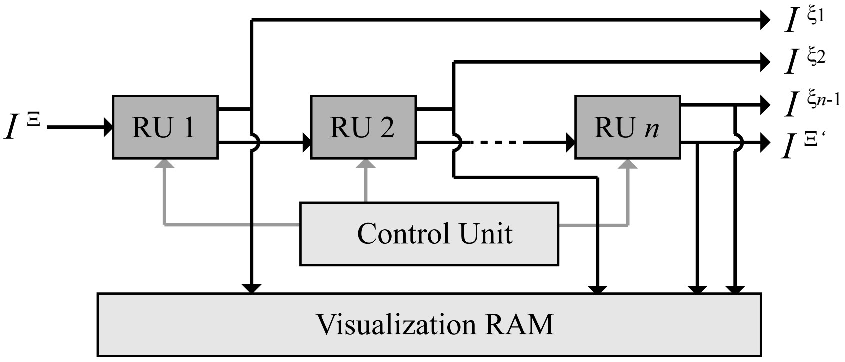


Figure 7.7: Overview of the proposed hardware implementation.

Recognition Unit

The internal structure of the RU is shown in Fig. 7.8. First of all, the marker image m^{ξ} is to be created. It is done by processing the image according to (7-5). Both φ = H and V in the supremum $\bigvee$ in (7-9) are independent and therefore separated in two parallel branches. Each branch computes one erosion and one dilation using two Mathematical Morphology Blocks (MMB1-4). The marker is completed from the parallel branches in Arithmetic Logic Block 1 (ALB1) that performs the $\bigvee$.

In the second step, the marker m^{ξ} is used in the approximated particle reconstruction. It consists of the marker dilation (MMB5) followed by threshold operation defined in (7-7) with the input image. The result image I^{ξ} containing only the desired particles is obtained through comparison with the RU input image, see (7-7). Both previous operations are evaluated in ALB2.

Finally, the RU input image is split into two output images; I^{ξ} with classified particles, and the residual image $I^{\Xi'}$. This step is carried out in ALB2 as well. The FIFO memory connected between the input image and ALB2 must be sized properly to compensate the delay of the branch containing MMB{1:5}. For instance, let us consider that MMB{1:5} infer total delay of 5 image lines due to δ, ε intrinsic latency. The intrinsic latency is unavoidable and defined by dimensions of B. Hence, the FIFO must be capable of storing at least 5 image lines as well.

The MMB performs either morphological dilation or erosion on an input image by the structuring element B. It implements the rectangular dilation unit from Section 5.2. The ALB is intended to perform several arithmetic operations. Besides the reconstruction and the residue process described above, basic arithmetic operations as min(), max(), <, >, >0, <0, addition, or subtraction can be selected.

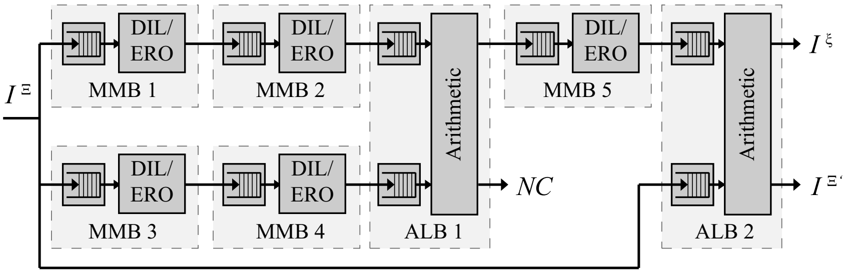


Figure 7.8: Internal structure of the RU. I^{ξ} contains classified particles, $I^{\Xi'}$ is the residual image.

Demonstration

The demonstration of particle traces classification into three types was implemented, see Fig. 7.9. It instantiates two RUs; the first I^A classifies blobs using the m^A marker image outputting dots and tracks in the residual image. The residual image from RU1 is read by the second RU2 that uses the marker m^E to classify tracks I^E. Hence, the residual image of RU2 $I^{\Xi'} = I^G$ contains dots only. All three outputs are stored in on-chip Visualization Memory and displayed on a screen.

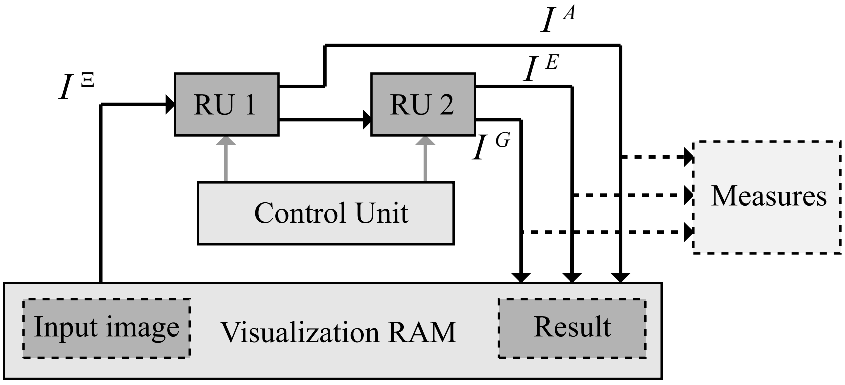


Figure 7.9: Overview of application that classifies dot, blob, and track particle traces.

7.2.3.1 Implementation Results

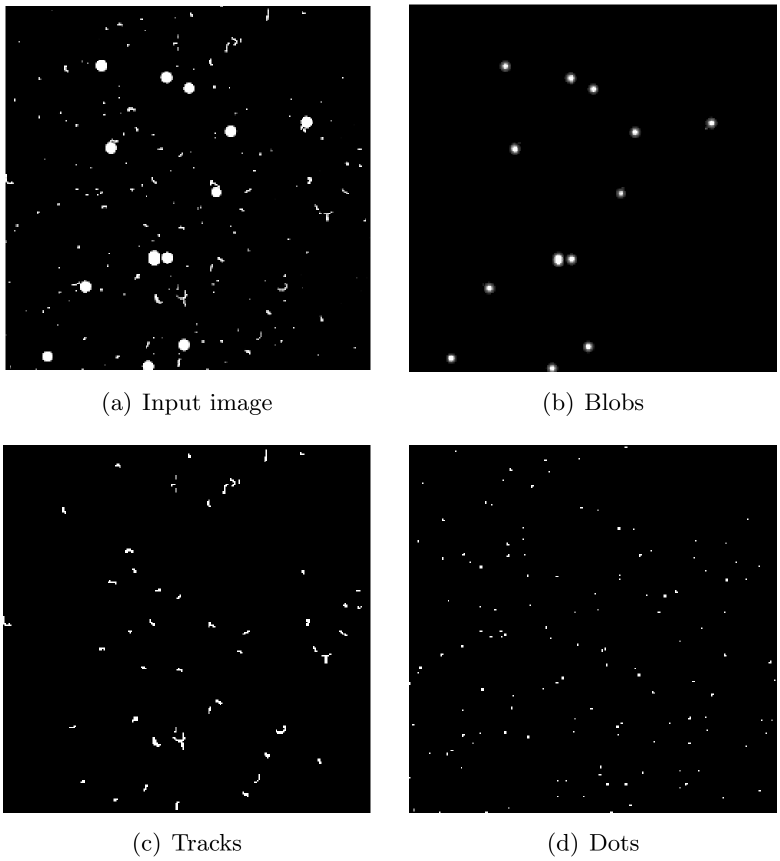
The proposed demonstration has been targeted to the Xilinx Virtex-5 FPGA (XC5VLX50T-1). The design (without the optional visualization) occupies the following hardware resources: 1405 registers, 4495 LUTs, and 9 36-kbit on-chip block RAMs.

The time benchmarks of the proposed design were performed on a set of Timepix images, each containing a mixture of all three kinds of particles. The results are outlined in Table 7.6. All Timepix images were processed in almost the same time with minimal differences, so we use the average value. The worst case presents

Table 7.6: Timing results of the classification.

Image type	Time [ms]	Latency [μs]	Rate [fps]
Best case	1.352	31.54	739
Worst case	1.678	39.4	594
Average Timepix	1.356	31.6	738

the lowest granted stream performance obtained on the most unpleasant gray-level image (artificial image containing monotonous gradient) whereas the best case conforms to the constant image. One can see that processing of Timepix image is very close to the best case since the Timepix image contains many zero-valued areas. The classification of a typical Timepix image is shown in Fig. 7.10. The input image in Fig. 7.10 (a) contains particles of all three kinds. The contrast of the input image was enhanced to make all the particles visible; the energy of blobs is few times greater then the energy of other particles. The images containing each kind of particles can be seen in Fig. 7.10 (b–d).



(a) Input image

(b) Blobs

(c) Tracks

(d) Dots

Figure 7.10: Example of obtained results: a) experimental input image (with enhanced contrast), b) classified as blobs, c) classified as tracks, d) classified as dots

This application illustrates usability of the proposed processing units for classification of particles in dedicated hardware, which can be considered as a typical embedded application. It processes the input image in a stream inferring minimal latency. We achieved very high performance rate of 738 frames per second thanks to the streaming pipeline structure. The high frame rate allows the Timepix detector to acquire images with a high sampling frequency, reducing thus the appearance of overlapping particles that can not be classified.

7.3 Conclusions

In this chapter we have illustrated usability of the proposed processing units in two applications of diverse purposes.

First, we have integrated all the proposed processing units into the FREIA platform. As the FREIA platform is build around a MicroBlaze processor, the new accelerator is designed as a peripheral connected to the processor and the memory by separated buses to attain a high image data throughput. For the most parallelized unit dilation by the square SE 61×61 is computed at performance of 276 Mpx/s, even for high-definition 1080p images.

In the second part, we presented the method of particle traces classification using the filter-based morphological markers instead of descriptors based on connected components, which are very computation intensive. Implemented in hardware, the classification recognizes three main types of traces: dots, blobs, and tracks; and can be naturally extended. The proposed architecture achieves very good, real-time performance 738 fps exceeding the current read-out capability of the Timepix measure device (90 fps).

8 General Conclusions and Perspectives

Contents

In this thesis, we discussed implementation of fundamental morphological filters with large structuring elements in the dedicated hardware. The major contributions of the thesis can be divided into three parts: algorithms, hardware implementation, and applications.

Algorithms

At first, we have reviewed the literature of existing algorithms usable for morphological filters. We have discussed that the queue-based algorithms seem to be suitable for hardware implementation thanks to the sequential access to data, minimal latency, and small memory requirements. These three properties combined allow for a simple and efficient concatenation of operators, which is necessary in order to obtain more complex operators. From the existing algorithms we have chosen the *Dokládal* algorithm ([Dokládal 2011]) due to its support of non-causal SEs for erosion and dilation over the algorithm by [Lemire 2006]. We have described this algorithm in detail in Chapter. 4 and enriched the family of supported SEs by inclined lines, which can form regular polygons. The computation of dilation by inclined line SEs is done along inclined discrete lines determined by the Bresenham line algorithm [Bresenham 1965]. The same approach was used in algorithms by [Soille 1996] or [Morard 2011] with the difference that we preserve sequential access to data, a crucial property for hardware implementation.

As the first main contribution of the thesis, we have proposed an original algorithm for arbitrary-oriented 1-D opening and pattern spectrum. This queue-based algorithm has constant complexity (i.e., the computation time is independent of the SE size), sequential access to input and output data, minimal latency, and small memory requirements. Such properties suggest that this opening algorithm should also allow for efficient and powerful hardware implementation like the *Dokládal* algorithm. Although 2-D opening is almost exclusively computed as a concatenation of erosion and dilation, using a dedicated algorithm for 1-D opening is justifiable. The proposed one-scan algorithm computes the opening with lower latency and smaller memory requirements than an erosion-dilation concatenation. Moreover, our algorithm computes the pattern spectrum in a single image scan, which is traditionally obtained iteratively as a residue of openings.

Despite the fact that our algorithm is tailored for the dedicated hardware, we have evaluated performance benchmarks on the GPP and GPU platforms in order to draw a comparison between the algorithms. The results have shown that two algorithms with fewer comparisons per pixel (but with less regular access to data and large memory requirements) performs better than our solution on the GPP. However, the GPU benchmarks have indicated that our algorithm outperforms any other 1-D opening algorithm on the GPU. This is not surprising as we know that regular access to data and small memory requirements help to ameliorate the parallelism (and hence the speed-up) of the GPU computation.

Hardware Implementation

Chapter 5 has described the hardware implementation of the selected efficient algorithms for dilation and opening, the fundamental operators of morphological filtering. The operations by different structuring elements has been implemented in a form of the following programmable processing units: rectangle unit, polygon unit, and 1-D opening and pattern spectrum unit. These processing units have some common properties:

- The processing time is linear with respect to the image size and independent of the SE size.
- The latency is mostly equal to the operator latency inferred by the size of the used SE. The memory requirements are small and proportional to the size of the used SE.
- The processing unit uses strictly sequential access to data at all algorithm levels. This property enables the application to eliminate any intermediate data storage in order to form compound operators; the processing units can be simply concatenated one after each other.
- Two levels of parallelism: (i) inter-operator parallelism in serial concatenations $\zeta = \delta\varepsilon \ldots \delta\varepsilon$, allowing to run all these atomic δ and ε operators simultaneously, and (ii) intra-operator parallelism in every atomic dilation/erosion. The intra-operator parallelism uses the principle of fast stream decomposition into several slower streams processed by multiple units in parallel without altering the sequential access property.
- The operation-specific parameters, i.e., the image size, the SE features, erosion/dilation select, are run-time programmable up to some specified upper bound at the beginning of processing each frame.

The proposed architectures serve as basic building blocks to be used for the construction of more complex operators such as ASF, granulometries, etc., with the same properties and performance. From the application point of view, a simple dilation or erosion does not really represent computing difficulties to either existing implementation mentioned in the state of the art. Neither large SEs represent an impassable obstacle; one can devise a pipe of processing elements (e.g., neighborhood processors) long enough to do the computation. However, the complex operators will eventually require intermediate image storage, and consequently loss of performances, or need an excessively long pipe, which will fit only the targeted application. It will lack the flexibility needed for hardware accelerators with a priori unknown specifications of applications. In our proposition, the main advantage against the existing architectures reside in the combination of ability to implement large SE without decomposition, programmable SE size and shape, and reduced length of the pipe. These advantages come naturally out in more challenging applications, such as ASF, as we have shown in the comparison using ASF filters.

The performance obtained on an FPGA are approaching the 100 fps on HDTV 1080p standard for dilation and 80 fps using SVGA image resolution for 1-D opening. These performances allied to the programmability are extremely interesting. They open the accessibility of advanced morphological operators in industrial systems running under severe time constraints, such as on-line production control, aging material defectoscopy, etc.

Applications

In Chapter 7, we have utilized the proposed processing units in two applications of different context and complexity. At first, we have integrated our proposition into the FREIA platform, which had already contained SPoC and TER@PIX architectures, to address the most performance-demanding applications. Our contribution enriched the capability of the FREIA platform by computing large SEs in a single scan, so with better efficiency and performance.

The second application using our proposed units was from the context of low-power embedded systems. Its main purpose was to classify the particles recorded by the Timepix particle detector based on the shape of traces using only few basic morphological operators to fit low-power FPGA devices. The classification has exploited the scalability of the processing units, which keeps hardware implementation resource-efficient even for small sizes of images and SEs, so the whole classification could run in parallel achieving high frame rate of 738 fps that allows for analysis of radioactive fluxes of unknown sources and spectra.

8.1 Perspectives

The issues discussed in the thesis present many avenues of research for the future work. We will deal with the perspectives of algorithms and dedicated hardware implementation below.

Algorithms

Concerning our contribution to the field of algorithms for mathematical morphology, i.e., the original algorithm for 1-D arbitrary-oriented opening and pattern spectrum, there is a wide perspective of investigation the feasibility of the pattern spectrum operator on further applications, such as material characterization or crack detection [Obara 2007]. Another interesting aspect of the pattern spectrum algorithm may be the computation of local granulometries [Vincent 2000]. The local granulometries is useful for statistical methods to classify or to segment textures.

The proposed arbitrary-oriented opening algorithm has been implemented on a GPU [Karas 2012b] with a significant speed-up up to 50× over the GPP platform. We expect that the pattern spectrum extension of the algorithm should bring the comparable speed-up and allow for the real-time computation of the whole pattern spectrum with 180 directions.

Hardware Implementation

The rectangle and polygon dilation processing units proved to be useful for the vision applications in Chapter 7. However, these applications take advantage of simple interconnection, which may not be sufficient for general-purpose morphology computing platforms. Then some higher-level adaptable interconnection should be used, for instance an adaptable ring architecture [Ngan 2011]. Such a computing system would combine the high performance of the proposed units with high polyvalence of the adaptable ring.

The 1-D synchronous arbitrary-oriented dilation can be modified in two different ways. First, we can obtain a unit that does not compute only one dilation by l-pixel SE, but computes a vector of dilations by all possible SEs of lengths from 1 to l pixels at the same time. Such a vector may be very useful as a vector of features for statistical learning methods, see for example [Cord 2007]. The second branch of the prospective development of the synchronous architecture aims at arbitrary-shaped SEs using the SE decomposition into vertical cords similar to the [Urbach 2008] algorithm. As the synchronous architecture is capable of computing dilations by SEs of different lengths concurrently, it can be used to compute a vector of vertical line dilations for each column. Then in the second stage, the partial results of vertical dilations are combined together to form the arbitrary-shaped SE.

The 1-D arbitrary-oriented opening and pattern spectrum unit uses only inter-operator parallelism to achieve high performance arguing that this parallelism is able to exploit the full capabilities of dedicated hardware via instantiating many units in parallel, which is common in many applications, e.g., χ_l opening and pattern spectrum. However, in the cases where a small number of units (or even only one unit) are necessary, the overall application performance may be unsatisfactory. Then the performance of the opening unit can be ameliorated by intra-operator parallelism by partitioning the image into vertical stripes in the same manner like in the case of inclined line dilation units of parallel polygons, discussed in Section 5.3.3.

Publications

1. J. Bartovsky, P. Dokladal, E. Dokladalova, and V. Georgiev. Parallel implementation of sequential morphological filters. *Journal of Real-Time Image Processing.* 2011 (accepted, available online, doi:10.1007/s11554-011-0226-5).

2. P. Karas, V. Morard, J. Bartovsky, T. Grandpierre, E. Dokladalova, P. Matula, and P. Dokladal. GPU implementation of linear morphological openings with arbitrary angle. *Journal of Real-Time Image Processing.* 2012 (accepted, available online, doi:10.1007/s11554-012-0248-7).

3. J. Bartovsky, P. Dokladal, E. Dokladalova, M. Akil, and M. Bilodeau. Real-time streaming implementation of morphological filters with polygonal SE. *Journal of Real-Time Image Processing.* (accepted on June 2012, to appear).

4. J. Bartovsky, P. Dokladal, E. Dokladalova, and M. Bilodeau. One-scan algorithm for arbitrarily oriented 1-D morphological opening and slope pattern spectrum. In ICIP 2012, USA, October 2012.

5. J. Bartovsky, E. Dokladalova, P. Dokladal, and M. Akil. Efficient FPGA architecture for oriented 1-D opening and pattern spectrum. In ICIP 2012, USA, October 2012.

6. J. Bartovsky, D. Schneider, E. Dokladalova, P. Dokladal, V. Georgiev, and M. Akil. Morphological classification of particles recorded by the Timepix detector. In ISPA 2011, Croatia, September 2011.

7. J. Bartovsky, P. Dokladal, E. Dokladalova, and M. Bilodeau. Fast sequential algorithm for 1-D morphological opening and closing. In ISMM 2011, Italy, July 2011.

8. J. Bartovsky, P. Dokladal, E. Dokladalova, and V. Georgiev. Stream implementation of serial morphological filters with approximated polygons. In ICECS 2010, Greece, December 2010.

9. V. Kraus, M. Holik, J. Bartovsky, V. Georgiev, J. Jakubek, and D. Schneider. Space weather monitor based on the Timepix single particle pixel detector. In TELFOR 2010, Serbia, November 2010.

10. J. Bartovsky, E. Dokladalova, P. Dokladal, and V. Georgiev. Pipeline architecture for compound morphological operators. In ICIP 2010, Hong Kong, September 2010.

Bibliography

[Adams 1993] R. Adams. *Radial decomposition of discs and spheres.* CVGIP Graphical models and image processing, vol. 55, no. 5, pages 325–332, 1993. (Cited on page 53)

[Bagdanov 2002] A. D. Bagdanov, and M. Worring. *Granulometric analysis of document images.* In International Conference on Pattern Recognition, pages 468–471, 2002. (Cited on page 2)

[Bartovský 2010] J. Bartovský, E. Dokládalová, P. Dokládal, and V. Georgiev. *Pipeline architecture for compound morphological operators.* In International Conference on Image Processing, pages 3765–3768, Sept. 2010. (Cited on page 77)

[Bartovský 2011a] J. Bartovský, P. Dokládal, E. Dokládalová, and M. Bilodeau. *Fast streaming algorithm for 1-D morphological opening and closing on 2-D support.* In ISMM 2011, volume 6671 of *LNCS*, pages 296–305. Springer, July 2011. (Cited on pages 27, 29, 30, and 41)

[Bartovský 2011b] J. Bartovský, P. Dokládal, E. Dokládalová, and V. Georgiev. *Parallel implementation of sequential morphological filters.* Journal of Real-Time Image Processing, pages 1–13, 2011. (Cited on page 77)

[Bartovsky 2011c] J. Bartovsky, D. Schneider, E. Dokladalova, P. Dokladal, V. Georgiev, and M. Akil. *Morphological classification of particles recorded by the timepix detector.* In 7th ISPA 2011, pages 343 –348, sept. 2011. (Cited on pages 125 and 133)

[Bartovský 2012a] J. Bartovský, P. Dokládal, E. Dokládalová, and M. Bilodeau. *One-scan algorithm for arbitrarily oriented 1-D morphological opening and slope pattern spectrum.* In International Conference on Image Processing, Sept. 2012. (Cited on page 41)

[Bartovský 2012b] J. Bartovský, P. Dokládal, E. Dokládalová, M. Bilodeau, and M. Akil. *Real-time implementation of morphological filters with polygonal structuring elements.* Journal of Real-Time Image Processing, 2012. (Cited on page 78)

[Bartovský 2012c] J. Bartovský, E. Dokládalová, P. Dokládal, and M. Akil. *Efficient FPGA architecture for oriented 1-D opening and pattern spectrum.* In International Conference on Image Processing, Sept. 2012. (Cited on page 78)

[Beucher 1992] S. Beucher, and F Meyer. *The morphological approach of segmentation: the watershed transformation.* In Dougherty E. (Editor), Mathematical Morphology in Image Processing, pages 433–481, 1992. (Cited on page 2)

[Beucher 1995] S. Beucher, R. Peyrard, M. Bilodeau, and M. Gauthier. *Road monitoring and obstacle detection system by image analysis and mathematical morphology.* In European Automobile Engineers Cooperation Conference, pages 1–12, 1995. (Cited on page 3)

[Bonnot 2008] P. Bonnot, F. Lemonnier, G. Edelin, G. Gaillat, O. Ruch, and P. Gauget. *Definition and simd implementation of a multi-processing architecture approach on fpga.* In Design, automation and test in Europe, pages 610–615. ACM, 2008. (Cited on pages 126 and 132)

[Bouchami 2011] J. Bouchami, and *et al. Measurement of pattern recognition efficiency of tracks generated by ionizing radiation in a medipix2 device.* Nucl. Instrum. Meth. A, vol. 633, Supplement 1, no. 0, pages S187 – S189, 2011. (Cited on pages 133, 134, and 138)

[Brambor 2006] J. Brambor. *Algorithmes de la morphologie mathématique pour les architectures orientées flux.* PhD thesis, School of Mines Paris, Centre de Morphologie Mathématique, Sept. 2006. ID:5758. (Cited on page 29)

[Bresenham 1965] J. E. Bresenham. *Algorithm for computer control of a digital plotter.* IBM Systems Journal, vol. 4, no. 1, pages 25–30, 1965. (Cited on pages 5 and 143)

[Chien 2005] S.-Y. Chien, S.-Y. Ma, and L.-G. Chen. *Partial-result-reuse architecture and its design technique for morphological operations with flat structuring elements.* Circuits and Systems for Video Technology, IEEE Transactions on, vol. 15, no. 9, pages 1156 – 1169, sept. 2005. (Cited on pages 33, 34, 119, and 121)

[Clienti 2008a] Ch. Clienti, S. Beucher, and M. Bilodeau. *A system on chip dedicated to pipeline neighborhood processing for mathematical morphology.* In EURASIP, editeur, EUSIPCO 2008, Lausanne, August 2008. (Cited on pages 32, 119, 126, and 132)

[Clienti 2008b] Ch. Clienti, M. Bilodeau, and S. Beucher. *An efficient hardware architecture without line memories for morphological image processing.* In ACIVS '08, pages 147–156, Berlin, Heidelberg, 2008. Springer-Verlag. (Cited on pages 35 and 121)

[Clienti 2009] Ch. Clienti. *Architectures flot de données dédiées au traitement d'images par Morphologie Mathématique.* PhD thesis, School of Mines Paris, Centre de Morphologie Mathematique, Sept. 2009. ID:5758. (Cited on pages 29 and 30)

[Coltuc 1997] D. Coltuc, and I. Pitas. *On fast running max-min filtering.* IEEE Transactions on Circuits and Systems II, vol. 44, no. 8, pages 660 –663, aug 1997. (Cited on page 33)

[Cord 2007] A. Cord, D. Jeulin, and F. Bach. *Segmentation of random textures by morphological and linear operators.* In 8th ISMM, pages 387–398, Oct. 2007. (Cited on pages 2 and 146)

[CUDA 2012] CUDA. *nVidia cuda documentation.* `http://www.nvidia.com/object/cuda_home_new.html`, 2012. (Cited on page 30)

[Déforges 2010] O. Déforges, N. Normand, and M. Babel. *Fast recursive grayscale morphology operators: from the algorithm to the pipeline architecture.* Journal of Real-Time Image Processing, pages 1–10, 2010. 10.1007/s11554-010-0171-8. (Cited on pages 34, 119, 120, and 121)

[Diamantaras 1997] K. I. Diamantaras, and S. Y. Kung. *A linear systolic array for real-time morphological image processing.* J. VLSI Signal Process. Syst., vol. 17, no. 1, pages 43–55, 1997. (Cited on page 37)

[Dokládal 2011] P. Dokládal, and E. Dokládalová. *Computationally efficient, one-pass algorithm for morphological filters.* Journal of Visual Communication and Image Representation, vol. 22, no. 5, pages 411–420, 2011. (Cited on pages 5, 24, 29, 41, and 143)

[FREIA 2011] FREIA. *Framework for embedded image applications.* `http://freia.enstb.org`, 2008–2011. (Cited on pages 6 and 125)

[Fulguro 2010] Fulguro. *Fulguro documentation.* `http://fulguro.sourceforge.net`, 2010. (Cited on page 29)

[Gil 1993] J. Gil, and M. Werman. *Computing 2-d min, median, and max filters.* IEEE Trans. Pattern Anal. Mach. Intell., vol. 15, no. 5, pages 504–507, 1993. (Cited on page 22)

[Gil 2002] J. Gil, and R. Kimmel. *Efficient dilation, erosion, opening, and closing algorithms.* IEEE Trans. PAMI, vol. 24, no. 12, pages 1606–1617, 2002. (Cited on page 23)

[Gokhale 2010] M. B. Gokhale, and P. S. Graham. Reconfigurable computing: Accelerating computation with field-programmable gate arrays. Springer Publishing Company, Incorporated, 1st édition, 2010. (Cited on page 4)

[Gorpas 2009] D. Gorpas, and D. Yova. *Image segmentation for biomedical applications based on alternating sequential filtering and watershed transformation.* In Proc. SPIE, volume 7370, 2009. (Cited on page 2)

[Hauck 2007] S. Hauck, and A. DeHon. Reconfigurable computing: The theory and practice of fpga-based computation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. (Cited on page 4)

[Hedberg 2009] H. Hedberg, P. Dokládal, and V. Öwall. *Binary morphology with spatially variant structuring elements: Algorithm and architecture.* IEEE Transactions on Image Processing, vol. 18, no. 3, pages 562–572, 2009. (Cited on page 38)

[Heijmans 1997] H.J.A.M. Heijmans. *Composing morphological filters.* IEEE Trans. Image Processing, vol. 6, no. 5, pages 713 –723, may. 1997. (Cited on page 2)

[Holy 2006] T. Holy, J. Jakubek, S. Pospisil, J. Uher, D. Vavrik, and Z. Vykydal. *Data acquisition and prococessing software package for medipix2.* Nuclear Instruments and Methods in Physics Research, vol. 563, pages 254–258, 2006. (Cited on page 138)

[Ikenaga 2000] T. Ikenaga, and T. Ogura. *Real-time morphology processing using highly parallel 2-D cellular automata CAM2.* Image Processing, IEEE Transactions on, vol. 9, no. 12, pages 2018 – 2026, dec 2000. (Cited on page 37)

[ITU-R 2012] ITU-R. *Parameter values for ultra-high definition television systems for production and international programme exchange.* http://www.itu.int/rec/R-REC-BT.2020-0-201208-I/en, 2012. (Cited on page 1)

[Jakubek 2011] J. Jakubek. *Precise energy calibration of pixel detector working in time-over-threshold mode.* Nucl. Instrum. Meth. A, vol. 633, Supplement 1, no. 0, pages S262 – S266, 2011. (Cited on pages 133 and 134)

[Karas 2010] P. Karas. *Efficient computation of morphological greyscale reconstruction.* In MEMICS'10 – Selected Papers, volume 16 of *OpenAccess Series in Informatics (OASIcs)*, pages 54–61, Dagstuhl, Germany, 2010. (Cited on page 30)

[Karas 2012a] P. Karas, and J. Bartovsky. *CUDA-based linear openings.* http://sourceforge.net/p/linearopenings, Jan. 2012. (Cited on pages 67 and 73)

[Karas 2012b] P. Karas, V. Morard, J. Bartovský, T. Grandpierre, E. Dokládalová, P. Matula, and P. Dokládal. *GPU implementation of linear morphological openings with arbitrary angle.* Journal of Real-Time Image Processing, 2012. (Cited on pages 2, 30, 72, 73, and 146)

[Klein 1972] J.-C. Klein, and J. Serra. *The texture analyser.* J. of Microscopy, vol. 95, pages 349–356, 1972. (Cited on page 32)

[Klein 1989] J.C. Klein, and R. Peyrard. *Pimm1, an image processing ASIC based on mathematical morphology.* In ASIC Seminar and Exhibit, pages P7 – 1/1–4, sep 1989. (Cited on page 32)

[Knuth 1976] D. E. Knuth. *Big omicron and big omega and big theta.* SIGACT News, vol. 8, no. 2, pages 18–24, April 1976. (Cited on page 21)

[Knuth 1997] D. E. Knuth. The art of computer programming, volume 1 (3rd ed.): fundamental algorithms. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997. (Cited on page 21)

[Leiss 2007] E. L. Leiss. A programmer's companion to algorithm analysis. Taylor & Francis Group, LLC, Boca Raton, FL, USA, 2007. (Cited on page 21)

[Lemire 2006] D. Lemire. *Streaming maximum-minimum filter using no more than three comparisons per element.* CoRR, vol. abs/cs/0610046, 2006. (Cited on pages 24, 29, 49, and 143)

[Lemonnier 1995] F. Lemonnier, and J.-C. Klein. *Fast dilation by large 1D structuring elements.* In Proc. Int. Workshop Nonlinear Signal and Img. Proc., pages 479–482, Greece, Jun. 1995. (Cited on page 23)

[Llopart 2007] X. Llopart, R. Ballabriga, M. Campbell, L. Tlustos, and W. Wong. *Timepix, a 65k programmable pixel readout chip for arrival time, energy and/or photon counting measurements.* Nucl. Instrum. Meth. A, vol. 581, no. 1-2, pages 485 – 494, 2007. (Cited on page 133)

[Malamas 2000] E. N. Malamas, A. G. Malamos, and T. A. Varvarigou. *Fast implementation of binary morphological operations on hardware-efficient systolic architectures.* J. VLSI Signal Process. Syst., vol. 25, no. 1, pages 79–93, 2000. (Cited on page 37)

[Mamba 2012] Mamba. *Mamba documentation.* `http://cmm.ensmp.fr/Micromorph`, 2012. (Cited on page 29)

[Maragos 1989] P. Maragos. *Pattern spectrum and multiscale shape representation.* IEEE Trans. Pattern Anal. Mach. Intell., vol. 11, no. 7, pages 701–716, 1989. (Cited on pages 2 and 16)

[Maragos 2005] P Maragos. *Morphological filtering for image enhancement and feature detection.* Image & Video Processing Book (2nd ed.), pages 135–156, 2005. (Cited on page 2)

[Marion 2004] V. Marion, O. Lecointe, C. Lewandowski, J-C. Morillon, R. Aufrere, B. Mercotegui, R. Chapuis, and S. Beucher. *Robust perception algorithm for road and track autonomous following.* In Unmanned ground vehicle technology Conference, pages 55–66, 2004. (Cited on page 3)

[Matas 2008] P. Matas, E. Dokládalová, M. Akil, T. Grandpierre, L. Najman, M. Poupa, and V. Georgiev. *Parallel algorithm for concurrent computation of connected component tree.* In ACIVS 2008, volume LNCS 5259, pages 230–241, 2008. (Cited on pages 26 and 134)

[Matheron 1975] G. Matheron. Random sets and integral geometry. Wiley New York, 1975. (Cited on pages 1, 2, 10, and 16)

[MATLAB 2012] MATLAB. *Matlab documentation.* http://www.mathworks.com/products/matlab, 2012. (Cited on page 29)

[Mealy 1955] G. H. Mealy. *A method for synthesizing sequential circuits.* Bell Systems Technical Journal, vol. 34, pages 1045–1079, sept. 1955. (Cited on page 78)

[Menotti 2007] D. Menotti, L. Najman, and de Albuquerque A. *1D Component tree in linear time and space and its application to gray-level image multithresholding.* In Proceedings of 8th ISMM, volume 1, pages 437–448. INPE, 2007. (Cited on page 26)

[Moore 2007] J. A. Moore, K. A. Pimbblet, and M. J. Drinkwater. *Mathematical morphology: Star/galaxy differentiation & galaxy morphology classification.* Publications of the Astronomical Society of Australia, vol. 23, no. 4, pages 135–146, 2007. (Cited on page 2)

[Morard 2011] V. Morard, P. Dokladal, and E. Decenciere. *Linear openings in arbitrary orientation in O(1) per pixel.* In Acoustics, Speech and Signal Processing (ICASSP), pages 1457–1460, may 2011. (Cited on pages 5, 27, 30, 69, and 143)

[Morph-M 2012] Morph-M. *Morph-M documentation.* http://cmm.ensmp.fr/Morph-M, 2012. (Cited on page 29)

[Ngan 2011] N. Ngan. *Etude et conception d'un reseau sur puce dynamiquement adaptable pour la vision embarquee.* PhD thesis, Paris-Est, ESIEE Paris, Dec. 2011. (Cited on page 146)

[Normand 2003] N. Normand. *Convex structuring element decomposition for single scan binary mathematical morphology.* In Discrete Geometry for Computer Imagery, volume 2886 of *LNCS*, pages 154–163. Springer Berlin, Heidelberg, 2003. (Cited on pages 13 and 34)

[nVidia 2012] nVidia. *nVidia GPU programming guide.* http://developer.nvidia.com/nvidia-gpu-programming-guide, 2012. (Cited on pages 30 and 73)

[Obara 2007] B. Obara. *Identification of transcrystalline microcracks observed in microscope images of a dolomite structure using image analysis methods based on linear structuring element processing.* Computers and Geosciences, vol. 33, pages 151–158, 2007. (Cited on page 146)

[Octave 2012] Octave. *Gnu octave documentation.* http://www.gnu.org/software/octave, 2012. (Cited on page 29)

[OpenCL 2012] OpenCL. *OpenCL documentation.* `http://www.khronos.org/opencl`, 2012. (Cited on page 30)

[OpenCV 2012] OpenCV. *OpenCV documentation.* `http://opencv.willowgarage.com`, 2012. (Cited on pages 29, 30, and 49)

[Pecht 1985] J Pecht. *Speeding-up successive minkowski operations with bit-plane computers.* Pattern Recognition Letters, vol. 3, no. 2, pages 113 – 117, 1985. (Cited on page 22)

[Pitas 1989] I. Pitas. *Fast algorithms for running ordering and max/min calculation.* Circuits and Systems, IEEE Transactions on, vol. 36, no. 6, pages 795 –804, June 1989. (Cited on page 33)

[Salembier 1998] P. Salembier, A. Oliveras, and L. Garrido. *Anti-extensive connected operators for image and sequence processing.* IEEE Trans. on Image Proc, vol. 7, no. 4, pages 555–570, 1998. (Cited on page 26)

[Serra 1982] J. Serra. Image analysis and mathematical morphology, volume 1. Academic Press, New York, 1982. (Cited on page 10)

[Serra 1988] J. Serra. Image analysis and mathematical morphology, volume 2, theoretical advances. Academic Press, London, 1988. (Cited on pages 1, 14, and 16)

[Serra 1989] J. Serra. *Toggle mappings.* From pixels to features, pages 61–72, 1989. (Cited on page 2)

[Serra 1992] J. Serra, and L. Vincent. *An overview of morphological filtering.* Circuits Syst. Signal Process., vol. 11, no. 1, pages 47–108, 1992. (Cited on pages 1 and 2)

[Soille 1996] P. Soille, E. J. Breen, and R. Jones. *Recursive implementation of erosions and dilations along discrete lines at arbitrary angles.* IEEE Trans. Pattern Anal. Mach. Intell., vol. 18, no. 5, pages 562–567, 1996. (Cited on pages 5, 24, 49, 57, 69, and 143)

[Soille 2003] P. Soille. Morphological image analysis: Principles and applications. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003. (Cited on pages 10, 12, and 133)

[Sternberg 1986] S. Sternberg. *Grayscale morphology.* Comput. Vision Graph. Image Process., vol. 35, no. 3, pages 333–355, 1986. (Cited on page 14)

[T. Randen 2012] T. Randen. *Brodatz Textures.* `http://www.ux.uis.no/~tranden/brodatz.html`, 2012. (Cited on page 74)

[Urbach 2004] E. R. Urbach, J. B. T. M. Roerdink, and M. H. F. Wilkinson. *Connected rotation-invariant size-shape granulometries.* In International Conference on Pattern Recognition, pages 688–691, 2004. (Cited on page 2)

[Urbach 2008] E. R. Urbach, and M. H. F. Wilkinson. *Efficient 2-D grayscale morphological transformations with arbitrary flat structuring elements.* IEEE Trans. Image Processing, vol. 17, no. 1, pages 1 –8, jan. 2008. (Cited on pages 25, 49, 69, and 146)

[Van Droogenbroeck 1996] M. Van Droogenbroeck, and H. Talbot. *Fast computation of morphological operations with arbitrary structuring elements.* Pattern Recogn. Lett., vol. 17, no. 14, pages 1451–1460, 1996. (Cited on page 25)

[Van Droogenbroeck 2005] M. Van Droogenbroeck, and M. J. Buckley. *Morphological erosions and openings: Fast algorithms based on anchors.* J. Math. Imaging Vis., vol. 22, no. 2-3, pages 121–142, 2005. (Cited on pages 27, 49, 50, and 69)

[van Herk 1992] M. van Herk. *A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels.* Pattern Recogn. Lett., vol. 13, no. 7, pages 517–521, 1992. (Cited on page 22)

[Vapnik 1995] V. N. Vapnik. The nature of statistical learning theory. Springer-Verlag New York, Inc., New York, NY, USA, 1995. (Cited on page 2)

[Velten 2004] J. Velten, and A. Kummert. *Implementation of a high-performance hardware architecture for binary morphological image processing operations.* In Circuits and Systems, 2004. MWSCAS '04. The 2004 47th Midwest Symposium on, volume 2, pages II–241 – II–244 vol.2, 25-28 2004. (Cited on page 32)

[Vincent 2000] L. Vincent. *Granulometries and opening trees.* Fundam. Inform., vol. 41, no. 1-2, pages 57–90, 2000. (Cited on page 146)

[Wei 2007] Z. Wei, Y. Hua, S. Hui-sheng, and F. Hong-qi. *X-ray image enhancement based on multiscale morphology.* In Bioinformatics and Biomedical Engineering, pages 702–705, july 2007. (Cited on page 2)

[Wilkinson 2008] M.H.F. Wilkinson, Hui G., W.H. Hesselink, J.-E. Jonker, and A. Meijster. *Concurrent computation of attribute filters on shared memory parallel machines.* IEEE Trans. Pattern Anal. Mach. Intell., vol. 30, no. 10, pages 1800 –1813, oct. 2008. (Cited on page 26)

[Xilinx 2009] Xilinx. *Virtex-5 family documentation.* `http://www.xilinx.com/support/documentation/virtex-5.htm`, 2009. (Cited on page 111)

[Xu 1991] J. Xu. *Decomposition of convex polygonal morphological structuring elements into neighborhood subsets.* IEEE Trans. Pattern Anal. Mach. Intell., vol. 13, no. 2, pages 153–162, 1991. (Cited on pages 13 and 53)

[Zhang 2011] X. W. Zhang, G. Thibault, and E. Decenciére. *Application of the morphological ultimate opening to the detection of microaneurysms on eye* *fundus images from a clinical database.* In International Congress for Stereology, Beijing, China, October 2011. (Cited on page 2)

[Zhuang 1986] X Zhuang, and R. M. Haralick. *Morphological structuring element decomposition.* Computer Vision, Graphics, and Image Processing, vol. 35, no. 3, pages 370–382, 1986. (Cited on page 13)