HAL
open science

Parallel Implementation of Sequential Morphological Filters

Jan Bartovsky, Petr Dokládal, Eva Dokladalova, Vjaceslav Georgiev

To cite this version:

Jan Bartovsky, Petr Dokládal, Eva Dokladalova, Vjaceslav Georgiev. Parallel Implementation of Sequential Morphological Filters. Journal of Real-Time Image Processing, 2011, to appear (-), pp.1-13. 10.1007/s11554-011-0226-5 . hal-00786367

HAL Id: hal-00786367

https://hal.science/hal-00786367

Submitted on 8 Feb 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

JRTIP manuscript No.
(will be inserted by the editor)

Jan Bártovský · Petr Dokládal · Eva Dokládalová · Vjaceslav Georgiev

Parallel Implementation of Sequential Morphological Filters

Received: date / Revised: date

Abstract Many useful morphological filters are built as more or less long concatenations of erosions and dilations: openings, closings, size distributions, sequential filters, etc.

An efficient implementation of such concatenation would allow all the sequentially concatenated operators run simultaneously, on the time-delayed data. A recent algorithm (see below) for the morphological dilation/erosion allows such inter-operator parallelism.

This paper introduces an additional, intra-operator level of parallelism in this dilation/erosion algorithm. Realized in a dedicated hardware, for rectangular structuring elements with programmable size, such an implementation allows to obtain previously unachievable, real-time performances for these traditionally costly operators. Low latency and memory requirements are the main benefits when the performance is not deteriorated even for long concatenations or high-resolution images.

Keywords Mathematical Morphology, Serial Filters, Real-Time Implementation, Dedicated Hardware

1 Introduction

Mathematical Morphology is very popular, self-contained, image processing framework providing a complete set of tools from filtering, multi-scale image analysis to pattern recognition. It has been used in a number of applications, including the biomedical and medical imaging, video surveillance, industrial control, video compression, stereology, or remote sensing ([8, 16, 18, 19]).

In image-interpretation applications requiring a high correct-decision liability, one often use robust multi-criteria and/or multi-scale analysis. It generally consists of a serial concatenation of alternating atomic operators dilation and erosion with a progressively increasing computing window, the so-called structuring element (SE). Its examples include:

- *Alternate Sequential Filters (ASF)* - that are concatenations of openings and closings with a progressively increasing structuring element, useful for multi-scale analysis [19, 20].
- *Size distributions* - (aka granulometries) are concatenations of openings allowing measuring the size distribution in a population of objects [14, 17, 28].
- *Statistical learning* - a selected set of morphological operators ζ_i can be separately applied to an image f. Then for every pixel $f(x,y)$, the vector of values $\zeta_i(f)(x,y)$ can serve as a vector of descriptors for the pixel-wise learning and classification [4].

Although built from basic blocks (the dilation and erosion), these operators are costly due to the number of iterations. The real-time capabilities (i.e., low latency) are even more difficult to achieve due to the sequential data dependence and high memory requirements.

The recently introduced algorithm for the dilation and erosion [7] shows how to handle efficiently the implementation of such concatenations. It enables an inter-operator level of parallelism where all the sequentially concatenated operators can run simultaneously, on time-delayed data. Obviously, it is fully exploited only if the algorithm is implemented in an adequate hardware (HW).

In this paper, we propose a HW implementation of the original algorithm and we introduce an additional, intra-operator level of parallelism. Such an implementation allows obtaining previously unachievable, real-time performances for these traditionally costly operators. Section 2

J. Bártovský and E. Dokládalová
Laboratoire Informatique Gaspard Monge
CNRS-UMLV-ESIEE (UMR 8049)
University Paris-Est
93162 Noisy-le-Grand Cedex, France
E-mail: {bartovsj,dokladae}@esiee.fr

P. Dokládal
Center of Mathematical Morphology (CMM)
Mines ParisTech
77305 Fontainebleau Cedex, France
E-mail: petr.dokladal@mines-paristech.fr

V. Georgiev
Faculty of Electrical Engineering
University of West Bohemia
30614, Pilsen, Czech Republic
E-mail: georg@kae.zcu.cz

discusses the state of the art of existing dilation/erosion algorithms and concludes by the novelties presented in this paper. Section 3 and 4 recall the definitions and algorithmic principles and Section 5 illustrates the functional scheme of a sequential HW implementation. Then, Section 6 introduces the parallel implementation allowing obtaining an additional performance increase. Finally, Section 7 presents results obtained on FPGA.

2 Fast Implementations of Morphological Filters

During the last decades, propositions of optimized implementations concentrated on the efficiency of computing the dilation and erosion. The majority of authors measures this efficiency as a number of comparisons per pixel. Nevertheless, the minimization of comparisons can result in high memory requirements. It can even penalize the execution time since the overall latency issues are neglected.

For the following, we define as *operator latency* the latency introduced by the dependence of the result on future data samples. For example the max filter $y_i = \max(x_{i-2}, x_{i-1}, \ldots, x_{i+2})$ has operator latency 2. We define as *algorithm latency* any additional latency introduced by the algorithm, e.g., the necessity to perform a reverse scan on data. The latency is a time-less measure expressed in a number of data samples.

Note that there is also an additional delay, called *computing latency*, induced by the time needed to compute the result after all data are available. It is a platform dependent measure, independent of two previous latency definitions. In the example above, the polyadic max can either be executed sequentially on sequential machines, or in parallel on the dedicated hardware.

Then the overall latency of the system is the sum of these three terms.

2.1 Algorithmic Advances

The most efficient dilation/erosion algorithms are based on the SE decomposition to a set of basic, more easily optimized shapes, see [22, 30, 31]. A special attention is paid to 1-D algorithms obtaining a significant gain in the overall performance.

The most popular 1-D algorithm is called HGW (published by van Herk [26], and Gill and Werman [9]). The computation complexity per pixel is $\mathcal{O}(1)$, i.e., is independent of the SE size. Nonetheless, the algorithm requires two scans: forward and reverse. Lemonnier [12] proposes to identify local extrema and propagate their values as long as it is covered by the SE. Again, forward and reverse scans are needed. Notice that in 2-D the reverse scan of the vertical component multiplies the algorithm latency by a factor of the image width.

Lemire [11] proposes a fast, stream-processing algorithm for causal linear SE. It runs also on floating-point data, has low memory requirements and zero algorithm latency. However, an intermediate storage of local maxima results in a random access to the input data. This problem is solved in Dokladal and Dokladalova [7] using the strictly sequential access to the data. It allows the real on-the-fly computing and has zero algorithm latency.

A different approach represents the algorithm proposed by Buckley and Van Droogenbroeck [25]. It detects the anchors – the portions of the signal unaffected by the operator – and updates only the parts to be modified by the operation. It has zero algorithm latency. However, the algorithm uses a histogram which makes memory requirements dependent on the number of gray levels.

Recently, Urbach and Wilkinson [24] propose an algorithm for arbitrary shaped 2-D flat SEs based on the computation of multiple horizontal linear SEs for every pixel and storing them in a look-up table. The result is then computed by taking the maximum from the intermediate values (stored in the look-up table) corresponding to the shape of the SE. The horizontal linear SE can be computed with one of the above mentioned 1-D algorithms.

2.2 Implementations

In the beginning of the 70's, Klein and Serra [10] propose a texture analyzer for linear and rectangular SE by decomposition based on the delay-line concept. More recently, Velten and Kummert [27] propose also a delay-line based architecture supporting arbitrary-shaped SEs. However, the complexity being quadratic $\mathcal{O}(W^2)$ (W denotes the length of the SE), it becomes penalizing for large SEs. In Chien *et al.* [2], the authors show how to reduce the number of redundant comparisons within large SEs by merging adjacent smaller SEs. The complexity becomes $\mathcal{O}(\lceil \log_2(W) \rceil)$ with identical memory requirements.

Clienti *et al.* [3] proposes a highly parallel morphological System-on-Chip. It is based on a set of neighbourhood processors optimized for 3×3 SE, interconnected in a partially configurable pipeline. Larger SE is obtained by homothecy (see Basic Notions below) requiring to instantiate a deep pipe of these processors or multiple image scans.

A similar approach has been published by Deforges *et al.* [5]. Based on Xu's [30] decomposition combined with a stream implementation, the authors propose a pipeline architecture composed of the elementary parametrizable blocks. It handles an arbitrary convex shape of structuring element in only one scan of the input image. However, using large SE will require the proportional increase of the atomic HW resources, concatenated in a deep pipe. The principal limitation comes from a limited programmability of such a pipe.

To complete this brief survey, we can also cite the systolic architectures proposed by Diamantaras and Kung [6], Malamas *et al.* [13] or Shih *et al.* [21] for gray-scale or binary morphology. Their common inconvenience is the need of an intermediate storage for 2-D structuring element and a long response time of the system.

2.3 Novelty of this Paper

All previous algorithms optimize the dilation/erosion algorithm, rather than the entire operator chain. The performance will inevitably decrease for more complex applications with long loops (iterations, idempotence) or concatenations.

Consider some serial morphological filter $\zeta = \delta\varepsilon \ldots \delta\varepsilon$ (with δ and ε standing for dilation and erosion, see Section 3 below for details). If the atomic operators δ and ε use sequential access to data then the entire ζ can run on pipelined, time-delayed data. If the atomic algorithms δ and ε - in addition - have zero algorithm latency, then the entire chain ζ inherits the same properties: sequential data access and zero algorithm latency. This is an interesting property, since computing ζ suddenly becomes very efficient: in stream, with only the (further irreducible) operator latency of ζ.

In comparison with the preceding state of the art, the Dokladal [7] algorithm extends the possibility to implement erosion/dilation filters with the arbitrarily large, 2-D SE in only one scan over the image, with the minimal algorithm latency and memory requirements. If implemented in a dedicated hardware, we can obtain the same features even for the implementation of long concatenations ζ.

This paper starts from the sequential HW implementation of the Dokladal algorithm (published in Bartovsky *et al.* [1]). It describes more deeply the implementation features and optimization techniques. It shows how to exploit the inter-operator parallelism in ζ. Additionally, it introduces another intra-operator parallelism in the computation of the 2-D erosion/dilation. The 2-D erosion/dilation is implemented as a run-time programmable block. The operation (erosion or dilation), the size and the origin of the SE can be modified on-the-fly between two frames.

Several such blocks concatenated in a pipeline allow obtaining previously unachievable, real-time performances for operators in the form of ζ. We can reach almost 100Hz HDTV 1080p performance, independent of the length of ζ.

3 Basic principles

Let $\delta_B, \varepsilon_B \colon \mathbb{Z}^2 \to \mathbb{R}$ be a dilation and an erosion on grey-scale images, parametrized by a structuring element B, assumed rectangular, flat (i.e., $B \subset \mathbb{Z}^2$) and translation-invariant, defined as

$$\delta_B(f) = \bigvee_{b\in B} f_b \tag{1}$$

$$\varepsilon_B(f) = \bigwedge_{b\in \widehat{B}} f_b \tag{2}$$

The hat $\widehat{\ }$ denotes the transposition of the structuring element, equal to the set reflection $\widehat{B} = \{x \mid -x \in B\}$, and f_b denotes the translation of the function f by some vector b. The SE B is equipped with an origin $x \in B$. Below, $B(x)$ denotes B placed with its origin at x.

The implementation of (1) and (2) consists of searching the extremum of f within the scope of B

$$[\delta_B(f)](x) = \max_{b\in B} [f(x-b)] \tag{3}$$

$$[\varepsilon_B(f)](x) = \min_{b\in B} [f(x+b)] \tag{4}$$

The dilation and erosion by convex structuring elements verify the homothecy. Let B be some convex structuring element, and rB the change of scale of B, with $r > 1$, $r \in \mathbb{Z}$. Then for the dilation we have

$$\delta_{rB} f = \underbrace{\delta_B \ldots \delta_B}_{r-\text{times}} f. \tag{5}$$

The homothecy allows obtaining large-size dilations by repeating several times the dilation by a small SE.

Combinations of dilations and erosions form other operators. The basic concatenation products are opening $\gamma_B = \delta_B\varepsilon_B$ and closing $\varphi_B = \varepsilon_B\delta_B$. From here we can form the 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 Alternate Sequential Filters (ASF), originally proposed by Sternberg [23], and studied in Serra [19], Chapter 10. In general, it is a family of operators parametrized by some $\lambda \in \mathbb{Z}^+$, obtained by the alternating concatenation 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 $\xi = \varphi$:

$$ASF^\lambda = \gamma^\lambda \varphi^\lambda \ldots \gamma^1 \varphi^1 \tag{6}$$

starting with a closing, and

$$ASF^\lambda = \varphi^\lambda \gamma^\lambda \ldots \varphi^1 \gamma^1 \tag{7}$$

starting with an opening.

The second application example is the size distribution of a population of objects [14, 17, 28]. One way to compute them is the residue from a sequence of openings

$$sd(\lambda) = \|\gamma^\lambda - \gamma^{\lambda-1}\| \tag{8}$$

The following section briefly recalls the principles of the used algorithm, [7]. It starts by the 1-D dilation algorithm, followed by the principle of separation of the n-D dilation into perpendicular 1-D computations, preserving the stream aspects at all levels.

4 Algorithm Description

4.1 1-D Dilation Algorithm

The algorithm principles and properties have been originally published in [7]. We briefly recall the main important principles for HW implementation.

For some 1-D input signal $f \colon 1 \ldots N \to \mathbb{R}$, the algorithm[1] computes the value $\delta_B f(wp) = f(rp)$. The SE B, $B \subset \mathbb{Z}$,

[1] See Appendix for the 1-D dilation pseudocode

is a line segment, containing its origin, and not necessarily symmetric. Consequently, the size of B is given by the span from the centre to the left and to the right, $SE1$ and $SE2$. The length of B is $SE1+SE2+1$. The coordinates wp and rp stand for the current *writing* and *reading* positions.

The algorithm uses a FIFO queue Q. The queue supports operations *push*, *pop* and *dequeue* (modifying the FIFO's content) and queries *front* and *back*. The input signal f is read sequentially. A newly read value $f := f(rp)$ is inserted in the FIFO queue as a pair $\{f, rp\}$, the sample f and reading position rp (code line 3). In this pair, one can independently access either the value or the position by indexing. For example the last stored element's value can be accessed (without dequeing it) by a query Q.back()[1].

The algorithm does not store non decreasing intervals (see [7] for details and proof). The values that appear to belong to increasing or constant intervals are dequeued (code lines 1-2). Consequently, the values stored in the queue are always ordered in a decreasing order.

The old values, uncovered by the SE, are retrieved from the queue (code lines 4-5). The result of the dilation $\delta_B f(x)$ is read at the front of the queue (code line 7). The result becomes available as soon as enough input data have been read, otherwise the output is empty (code line 9).

4.2 2-D Dilation Algorithm

The separability of n-D morphological dilation into lower dimensions is a well known property. For example, 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. Then the dilation by a rectangle R can be computed by concatenation of two perpendicular 1-D dilations

$$\delta_R = \delta_V \delta_H. \tag{9}$$

The sequential access to the data in 1-D makes that two perpendicular 1-D computations can be assembled into 2-D with sequential access at both levels, 2-D and 1-D, for both input and output data. There is no additional latency and no intermediate storage (the data are pipelined).

See the example of dilation by a rectangle $R{=}H \oplus V$ of an $N \times M$ image f, Fig. 1. The image is sequentially read in the raster-scan mode, line by line from left to right. The various indices rp and wp denote *reading* and *writing positions*, respectively, for the segments H and V, and the rectangle R. The computation is illustrated for column i and line j, i.e., the result $\delta_R f(i, j)$ is to be written at wp_R.

The computation of $\delta_R = \delta_V \delta_H$ decomposes as follows: The current reading position of δ_R coincides with rp_H, that is $rp_R = rp_H$. The result of the horizontal dilation, at wp_H, is immediately read by the vertical dilation in the respective column, that is $wp_H = rp_V$. The result of the vertical dilation δ_V is written at the writing position wp_R, i.e., $\delta_V = wp_R$.

Notes:
- The rp_r and wp_r run over the image in the raster scan mode. The distance between rp_R and wp_R is the (further irreducible) operator latency.
- There is one instance of the horizontal dilation running at the current line j, and N instances of vertical dilation, i.e., one per each column.

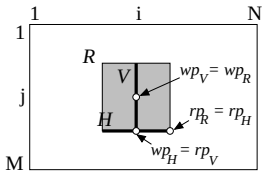


Fig. 1 Decomposition of dilation by a rectangle R into two 1-D dilations by segments H and V, see (9).

5 Sequential Hardware Implementation

In this section, we firstly describe in details the implementation of the 1-D dilation in a basic block that (thanks to the separability) can be used as a building brick in any dimensional system. We illustrate this below on a 2-D dilation or erosion. Secondly, we show how the intra-operator parallelism can be introduced to increase the performance.

5.1 1-D Algorithm Implementation

The 1-D algorithm presented in Section 4 is a system with sequential behavior. It contains a *while* loop that can not be unrolled (uncertain number of iterations). The common way to implement such a system is the Mealy Finite-State Machine (FSM, see [15]). The FSM issues all the necessary operations over the memory as well as it controls the input and output data-flow. It consists of 2 states $\{S1, S2\}$.

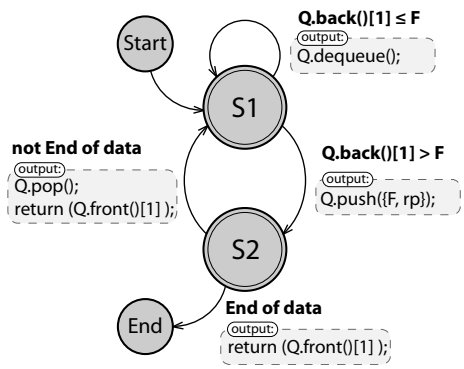


Fig. 2 State diagram of the 1-D Algorithm FSM. State transition conditions are typeset in bold; the output signals are given in a shadow bounding box.

The $S1$ state *Dequeues all useless values*. It is a data dependent stage of the algorithm as it dequeues an a priori unknown amount of pixels. This is represented in the code by

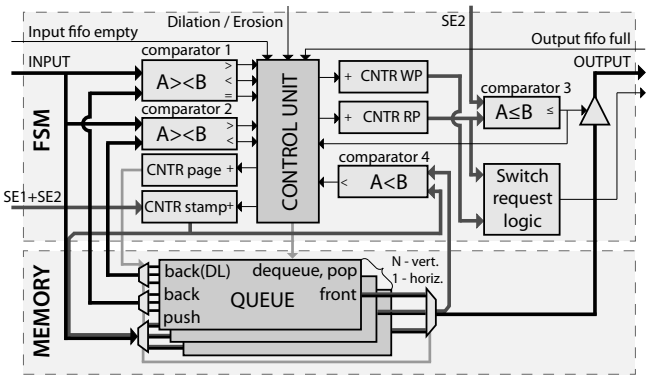


Fig. 3 Overview of implemented 1-D architecture. The FSM part manages computation, memory part contains the data storage–queues

the *while* statement (code line 1). Consequently, its computation time varies from 1 to the SE size clock cycles in the worst case when all the previously stored pixels are unnecessary.

The *Enqueue current sample* signal (code line 3) is issued upon the transition from $S1$ to $S2$.

The $S2$ state handles the 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 executed in parallel. Consequently, the execution of $S2$ takes only one clock cycle.

5.2 1-D Block Architecture

The HW implementation can be separated into 2 areas (Fig. 3), the FSM part and the memory part. The FSM manages entire computing procedure and temporarily stores values in the memory part. The memory instantiates one FIFO queue in the case of horizontal direction (horizontal scan) and N FIFO queues in the vertical case (N is the image width). The queues are addressed by a modulo N Page counter (active in the case of vertical direction).

The Control unit is a sequential circuit that manages the state transitions. It increments the rp, wp and manages the Page and position Stamp counter appropriately. The Control unit also performs the queue memory operations and handles the backward full flags used for data-flow control.

Principle

In the beginning of $S1$, the last queued pixel is invoked by Back() operation from the queue and fetched to the Comparator 1. The pixel value is compared with the value of the current sample. Notice that the comparator evaluates all three possible relations ($>$,$<$,$=$) at the time, for both dilation and erosion. 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 stamp and enqueued (line 3).

The $S2$ invokes the oldest queued pair {pixel, stamp} by Front() operation. The read pixel is a correct result if rp has already reached or exceeded the $SE2$ parameter. This output allowing condition (line 6) is checked by Comparator 3. The deletion of outdated values is performed by comparing the current value of the reading-position stamp with the rp value of the oldest pair. Notice that the deletion has no impact on the output dilation value because Pop() operation (lines 4 and 5) issued by the Control unit has an effect only with the next clock edge.

The switch request logic is used only in the parallelized version of the architecture, see Section 6. It is a simple block containing several comparators which generate a signal with the last output value of each parallel segment. Its purpose is to inform the switch connected to the output that the end of the segment has been reached and the following segment is to be processed.

The entire set of parameters, i.e., SE dimensions and selection of the morphological function, is run-time programmable at the beginning of the line for 1-D, and of the frame for the 2-D implementation, respectively. In addition, no further controller is needed; the internal behavior is driven only by the regular scan order data-flow.

5.2.1 Reducing the impact of data-dependency

Hereafter, we briefly describe two techniques brought to the system for higher throughput and lesser area occupation.

Number of dequeue steps

The data-dependent number of dequeue steps (below denoted by $Steps$) has an unpleasant consequence on the HW design: longer balancing FIFOs (see Fig. 4), lower data throughput. For HW design it is important to minimize the worst case upper bound $Steps_{orig}{=}SE-1$.

The number of stored pixels is within $[1,SE[$. Suppose that we are to dequeue D pixels. We know that the pixels are queued in a strictly decreasing order. Thus, if the DL-th pixel ($DL<D$) can be dequeued then also all previous pixels can be dequeued. This can be done at the same time. Hence, the worst-case number of dequeue steps is

$$Steps = \max_{D<SE}(D \operatorname{div} DL + D \operatorname{mod} DL) \tag{10}$$

where D denotes the number of pixels to be dequeued and div and mod the integer division and the remainder operations. D can be regarded as a uniformly distributed, random variable $D \in [1,SE[$. Then we need to find the optimal DL that minimizes $Steps$ (Eq. 10) for all D such as

$$DL_{optim} = \arg\min_{DL<D} \max_{D<SE}(D \operatorname{div} DL + D \operatorname{mod} DL) \tag{11}$$

The optimal DL_{optim} brings us the minimal number of dequeue steps $Steps_{optim}$

$$Steps_{optim} = \min_{DL<D} \max_{D<SE}(D \operatorname{div} DL + D \operatorname{mod} DL) \tag{12}$$

Table 1 exemplifies, for some *SE* widths, the original and reduced number of dequeue Steps, obtained with optimal *DL*. Notice that more than one optimal *DL* can exist.

The *SE* is user programmable. DL_{optim} also is programmable, though it is useless to make it accessible to the user; it can instead be read from a LUT for every given user-specified SE.

Table 1 Optimal dequeue length *DL*, original and reduced number of dequeue steps for selected SE widths

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

Pixel addressing

The absolute pixel addressing in the queues can be advantageously replaced in the HW by using the modulo addressing. Instead of the absolute reading position rp, we use the relative modulo position $stamp = rp \bmod SE$. The pixels are enqueued by $Q.push(f, stamp)$ (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' one ($stamp = Q.front()[2]$).

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

5.3 2-D Dilation Implementation

Recall that dilation is separable into lower dimensions, Eq. 9. The dilation by a rectangle can be implemented using two 1-D dilation blocks, Fig. 4.

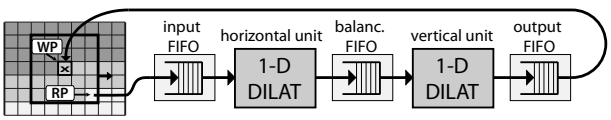


Fig. 4 2-D implementation is composed of 1-D blocks for respective directions.

The computing latency of the dilation varies per each pixel. In order to preserve the input/output stream flow, one needs to compensate the different latencies by insertion of balancing FIFOs. The FIFO fills when the preceding block outputs data faster than the subsequent block can read. The depth of this FIFO directly defines the upper bound of the system latency of the 2-D block.

Obviously, the FIFOs should be as small as possible. The necessary depth infers from the dequeuing worst case

$$F_{input} = \frac{Steps_{hor} + 2}{StreamRate} - 1 \tag{13}$$

$$F_{balance} = N\left(\frac{Steps_{ver} + 2}{StreamRate} - 1\right) \tag{14}$$

where $Steps_{hor}$ and $Steps_{ver}$ are numbers of the dequeue steps in horizontal and vertical directions (12).

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.

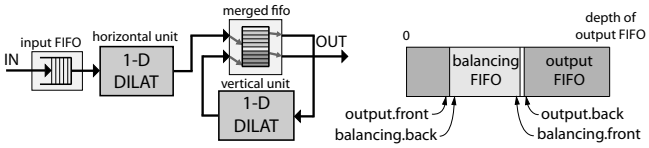


Fig. 5 Merged FIFO replaces the balancing and output FIFOs to reduce memory requirements.

The output and balancing FIFOs can be merged (see Fig. 5) into one memory thanks to the following properties: 1) the vertical unit reads exactly one pixel from 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, 2) 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 it). Merging both FIFOs reduces the memory to approximately one half. The result memory (see Fig. 5) has two pairs of standard FIFO ports, but it contains only one dual-port RAM.

5.4 Clock rate

The overall average clock rate stays in the interval from 2 clock cycles per pixel in the best case, up to 3 clock cycles per pixel in the worst case. The current rate between 2 and 3 clock cycles per pixel is data dependent.

A temporarily worst case arrives whenever a monotonously decreasing signal is followed by a high value. This makes a number of samples to be dequeued at the time (code lines 1 - 2, and the S1 state of the FSM), and the computing latency temporarily increases. However, the average computing latency remains unchanged, compensated by the fact that during the entire monotonous decrease of the signal no values have been dequeued. Therefore, the average clock cycles per pixel rate remains constant.

5.5 Memory requirements

The memory requirements of the 2-D architecture consist of horizontal and vertical computation-involved memories and two balancing FIFOs, defined by (13) and (14).

In the vertical case, the algorithm uses a several 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. 6) since only one queue is accessed at the time (the others are idle). A single memory block also allows using an off-chip memory.

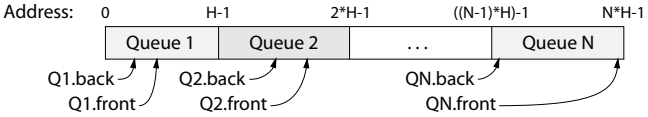


Fig. 6 Vertical Queues are mapped into linear memory space side by side. The front and back pointers are stored at separated memory.

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 modified 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 contribution per 2-D unit is given by:

$$M_{hor} = W(bpp + \lceil \log_2(W-1) \rceil) \quad \text{[bits]} \qquad (15)$$

$$M_{ver} = N(H(bpp + \lceil \log_2(H-1) \rceil) + 2\lceil \log_2(H-1) \rceil) \quad \text{[bits]} \qquad (16)$$

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

Example: Consider a dilation of 8bpp, SVGA image (i.e., 800×600=$N \times M$) by a square, 31x31 SE.

The computation (the queues) requires (15) and (16)

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

and

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

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

The input and the balancing FIFOs require (13) and (14)

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

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

6 Parallel Hardware Implementation

This section develops and implements the concept of the previously mentioned *intra-operator parallelism* in the dilation/erosion operator. Its main objective is to increase the throughput while maintaining the beneficial properties of the proposed algorithm, namely the sequential data access and minimal algorithm latency as much as possible.

The principle 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). Considering that the input data are fetched line by line, we propose a solution minimizing the waiting-for-data periods of all units.

The image partition for 2-D dilation conforms to the intersection of two horizontal and vertical partitions (Fig. 7). Its granularity is determined by the PD. The horizontal partition (partition of image among horizontal units) is interleaved, whereas the vertical units use the partition into compact blocks.

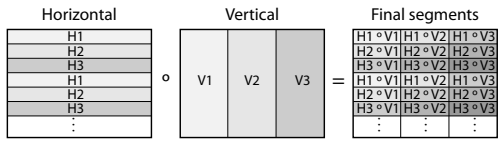


Fig. 7 Example of image partition for *PD*=3: image is divided horizontally line by line and into *PD* equal stripes in a vertical direction. The final image partition is obtained by intersection.

During the parallel processing the computation runs simultaneously at multiple segments of the image, see Fig. 8. These segments must belong to different columns and lines,i.e., must be placed on a diagonal.

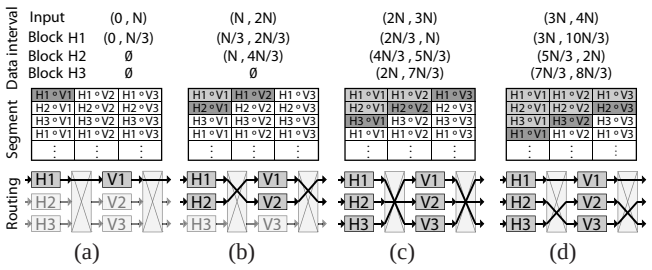


Fig. 8 Image partitioning and switch routing in parallel processing for *PD*=3. Decomposed in time - (a) beginning of processing, (b .. d) after kN pixels, k=1..3. The shading denotes the state : Dark Gray - being computed, Light Gray - already computed, White - waiting.

The input data rate can be theoretically *PD*-times faster than the computational throughput of one unit. Therefore, each image line needs to be buffered in a line buffer. The line buffers are filled at the external (fast) pixel rate and read by the internal *PD*-times slower rate.

Figure 8 gives an example for $PD = 3$. 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. 8(a). After *N* received pixels, the output of H1 is connected to V2 which belongs to output line 1. Since the H1 left V1 and line 2 is read, the H2 can start processing second line feeding V1 connected to output line 2, see Fig. 8(b). When the 2N input pixel is received, the H1 connects to V3, H2 connects to V2 and H3 connects to V1, see Fig. 8(c), and so on.

6.1 Architecture

The parallel architecture depicted in Fig. 9 contains four separable generic parts scalable by $n \equiv PD$: input buffer, horizontal and vertical parts and output buffer. The input buffer is mainly composed of the 1-to-*n* multiplexer and *n* line buffers (we omitted the control logic). It divides the fast input stream into *n* (*n*-times slower) streams processed by computational units as described above. The output buffer composes *n* 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.

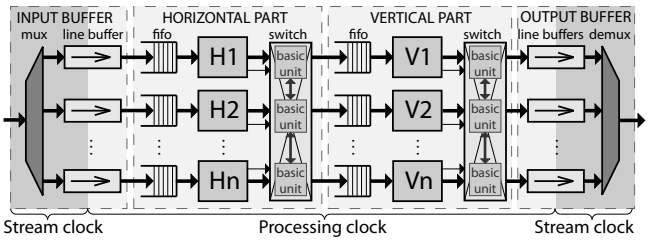


Fig. 9 Overview parallel 2-D architecture. The horizontal and vertical stages can be instantiated several times between input/output buffers to create compound operators.

Both horizontal and vertical parts instantiate *n* balancing FIFOs, *n* horizontal or vertical units, and one switch that manages the interconnection. Each horizontal unit along with the front-end FIFO conforms to Section 5.2.

The width of the processing area proportionally affects both vertical memories, see (14) and (16). 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 *n*. Contrarily, the memory requirements of every vertical part is divided by *n* because it processes only a fraction of the original image width. The area of the FSM of vertical units increases linearly with *n*.

6.2 Switching

The routing of the computation units is handled by a switch block. Every switch contains *n* 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 *n* interconnection channels. Notice that they are bidirectional: forward data and backward FIFO full flag. As described in Fig. 8, the output switching of all input ports is circular, i.e., V1, V2 ... Vn, 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 unit is available.

The moment when to switch a given port is provided by the preceding unit's *Switch Request* logic. It generates a request every time it crosses the border of adjacent segments. If the desired unit is free, the switch reconnects the channel. If not, the switch sets high the FIFO full flag of requesting unit to stall it until the desired destination unit is freed and the channel can be established. All the channels are switched independently so stalling one unit does not affect the others.

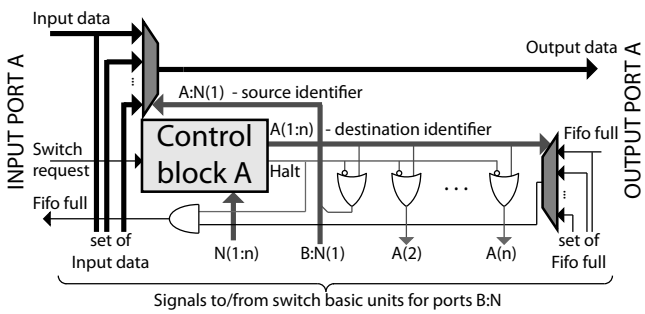


Fig. 10 Basic unit of the switch. Every switch contains *n* basic units for a correct routing between *n* input/output ports.

Figure 10 depicts the basic unit of the switch for one pair of input/output ports referred to as A. For *n* pairs of ports this circuitry is instantiated *n*-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.

7 Experimental results

The proposed 2-D stream processing architectures have been implemented in VHDL, and targeted to the Xilinx Virtex5 FPGA (XC5VSX95T-2) using the XST synthesis tool. The processing clock frequency is 100 MHz. Notice that the queues are gathered in a block RAM memory, and thus its access time augments the critical path delay.

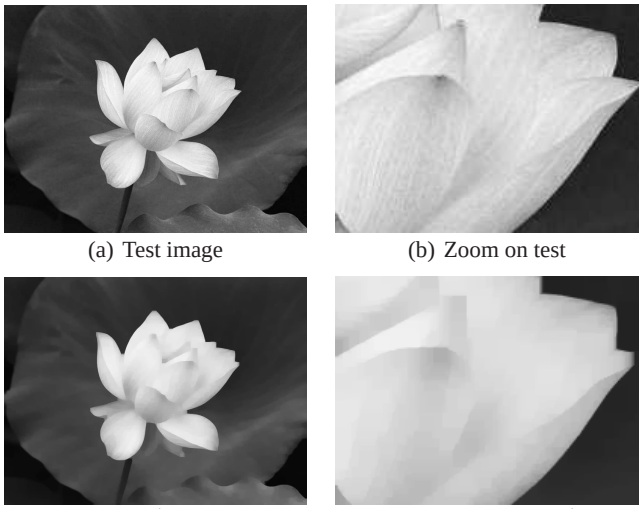
The measured performance for non-parallel architectures (PD=1) in terms of overall latency, clock cycles per pixel and FPGA area are given by Tables 2 and 3.

Table 2 Timing and area vs. SE, SVGA image size, PD=1.

Size of SE (sq.)	3x3	11x11	21x21	31x31	41x41
Latency [clk]	1908	9474	18888	28351	37969
Av. rate [clk/px]	2.344	2.356	2.360	2.361	2.361
Registers	212	232	242	242	252
LUTs	584	761	859	859	953
Block RAMs	2	6	13	13	28

Table 3 Timing, frame rate and area w.r.t. image, SE = 31x31 square, PD=1.

Size of Image	CIF	VGA	SVGA	XGA	1080p
Latency [clk]	12826	23465	28351	37472	69548
Av. rate [clk/px]	2.371	2.376	2.361	2.383	2.368
Experimental FPS	384	130	85	51	20.5
Worst-case FPS	319	106	68	41	16
Registers	231	237	242	242	253
LUTs	761	853	859	859	1057
Block RAMs	7	13	13	13	26



(a) Test image (b) Zoom on test

(c) ASF^4 filtered (d) Zoom on ASF^4

Fig. 11 (a) Experimental 800×600 lotus image. (b) Zoom on the fine veinous texture disadvantageous for the algorithm. (c) Result of ASF^4 filter, see Eq. 6 or 7. (d) Zoom on the ASF^4 filtered image.

One can observe that the overall latency is factor of the SE size, the image width (both caused by operator latency) and the pixel rate (computing latency). The average pixel rate (AR) remains constant (Table 2). The average pixel rate can be expressed by (17) and the stream frame-per-second (FPS) ratio by (18). T_{proc} is overall time consumed by processing and f_{clk}= 100 MHz is clock frequency of computation units.

$$AR = \frac{T_{proc} - 2SE_2(N+M+SE_2)/PD}{NM} \quad \text{[clk/px]} \quad (17)$$

$$FPS = \frac{f_{clk}PD}{ARNM + 2SE_2(N+M+SE_2)} \quad \text{[fr/s]} \quad (18)$$

M, N denote the width and height of the image, SE_2 denotes the width of the structuring element from the origin rightwards.

Concerning the area occupation (see the Xilinx documentation [29]), the number of registers is quasi-constant; the number of LUTs and BRAM blocks increases linearly with the SE and image sizes (Table 3). Although the vertical memory (size is given by (16)) is packed into the RAM block, the amount of the used memory always exceeds the theoretical value. It is caused by a different memory organizations; e.g., the required word is 13 bits whereas available memories are of width 36 bits and its fractions.

The experimental frames-per-second (FPS) rate is obtained on a natural test image (see Fig. 11). The worst-case FPS is a theoretical worst-case performance of the system expected on the synthetic saw-shaped data.

Table 4 presents the relative speed-up of the parallel architecture vs. the intra-operator parallelism PD. In terms of overall latency and average processing rate, the processing domain clock cycle is considered as a reference unit. Note that the latencies of parallel versions are merely fractions (divided by PD) of non-parallel values.

Table 4 Timing vs. degree of intra-operator parallelism PD. SVGA image, SE = 31x31 square.

PD	2	3	4	5	6
Latency [clk]	14243	9561	7244	5818	4893
Av. rate [clk/px]	1.220	0.824	0.625	0.505	0.426
Exp. speed up	1.938	2.869	3.785	4.682	5.554

Table 5 Area vs. degree of intra-operator parallelism PD. SVGA image, SE = 31x31 square.

PD	2	3	4	5	6
Registers	650	978	1280	1605	1938
LUTs	2138	3227	3862	4875	6054
Block RAMs	13	14	14	18	21
Reg. buf	661	969	1279	1587	1896
LUTs buf	1408	2086	2776	3459	4135

The FPGA area results, Table 5, are separated into 2 groups: the area of computing parts and buffers. The area of input and output buffers is linear w.r.t. both N and PD since their essential components are PD line buffers (FIFO memories with independent ports of N elements). The area of the operator units in terms of Slice registers and LUTs is proportional to PD as well because n independent circuits are instantiated in a parallel manner. Although the overall vertical memory requirements remain unaffected by PD, practically the number of occupied RAM blocks slightly increases. It is caused by a different memory organization.

Table 6 Timing and frame rate vs. image size, PD = 6, SE = 31x31 square

Size of Image	CIF	VGA	SVGA	XGA	SXGA	1080p
Latency [clk]	2208	3996	4893	6390	7391	11641
Av. rate [clk/px]	0.443	0.431	0.426	0.426	0.427	0.418
Experimental FPS	2075	724	472	290	174	113
Worst-Case FPS	1915	640	411	246	151	96

The ultimate timing results (*PD*=6) versus the image size are listed in Table 6. It illustrates the real performance of the architecture. It allows to achieve at least 96 fps with 1080p image size (full HD TV image size).

The worst case occurs on artificial saw-shaped image with no constant plateaus. Such an image infers the maximal number of algorithm's while-loop iterations. The best case fps (not mentioned in the table) is obtained with a constant image. A real, unfiltered image containing textures or random noise achieves performance somewhere between best and worst cases. For instance at 1080p, the worst case is 96 fps, best case 140 fps, achieved experimental performance is 113 fps.

This frame rate remains constant for any morphological serial filter (such as ASF). Obviously, the FPGA area increases accordingly to the size of the ASF. The implementation is eased by the fact that one can use an off-chip memory.

7.1 Comparison with existing HW implementations

Table 7 presents a comparison with other recent architectures. The table is divided into three sections. The processing unit section presents the features of a single 2D computational unit. The second part the HW specifications, and the third part the performance on a given application, an ASF filter.

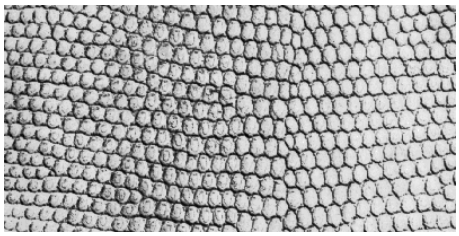
One can see that Clienti [3] offers a high throughput for small 3×3 rectangular SEs. Similarly, the Chien ASIC chip [2] provides very reasonable performance on small SEs. On the other hand, Déforges [5] directly offers large, non-rectangular, convex SE, but with a lower processing rate. The programmability is not mentioned, namely, the possibility to control the SE shape after the synthesis is not clear.

Although all these solutions are efficient for small SE sizes or short concatenations, they become more or less penalized for longer filters. This issue is illustrated in an Example Application, Table 7. It estimates the performances on a five-stage $\mathrm{ASF}^5 = \varphi_{11\times11}\gamma_{11\times11} \ldots \varphi_{3\times3}\gamma_{3\times3}$. Decomposed into a sequence of dilations and erosions, it can be realized as $\mathrm{ASF}^5 = \varepsilon_{11\times11}\delta_{21\times21} \ldots \varepsilon_{5\times5}\delta_{3\times3}$. Notice that it makes use of a progressively increasing SE. On neighborhood processors, large SE can be obtained using the homothecy Eq. 5. The Clienti SPOC instantiates 16 of 3×3 processing units. Hence, the ASF^5 will require 5 image scans with the entire image necessarily buffered in the memory. Chien also uses the homothecy. This deteriorates the throughput.

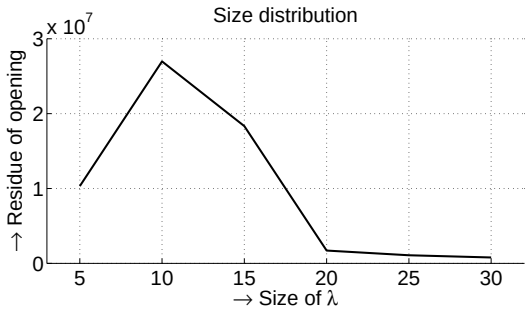
One could immediately figure out to instantiate a longer pipe in order to reduce the number of image scans. Alas, a long, fixed-length pipe lacks the flexibility. Consider another application for the illustration of the problem: the size distributions, exemplified by Fig. 12. Contrarily to ASF, the size distributions are often sampled sparsely, the SE increments by more than one and, at the same time, one often goes to much larger SE sizes. Every opening $\{\gamma_{B_i}\}$ in (8) needs to be output and stored in the memory to compute the subtraction. For small sizes, a long pipeline is underused and the workload of the processing units unbalanced, whereas for large λ one may still need several image scans.

For example, for sizes $\lambda = 5, 10, 15, 20, 25$, as in Fig. 12, the Clienti SPOC will require 7 image scans. The 16 processor pipe is underused for $\lambda = 5, 10, 15$, whereas it will require 2 scans for $\lambda = 20, 25$.

Our processing unit with programmable SE size avoids using the homothecy. This allows optimal workload distribution over the entire pipe, so important for processing large images in real-time systems.



(a) Example of a texture



(b) Size distribution sd(λ)

Fig. 12 The size distribution of the texture grain.

8 Conclusions

This paper describes an efficient implementation of serial morphological filters with flat, rectangular structuring elements of arbitrary size. The efficiency is obtained through the following properties:

– The computational complexity is linear w.r.t. the image size and independent of the SE size.

– The overall latency is mostly equal to the latency of the operator, inferred by the size of the used structuring element.

Table 7 Comparison of several FPGA and ASIC architectures concerning morphological dilation and erosion

	Processing unit					HW System		Example Application ASF[5]	
	Parallel degree	Supported SE	Throughput [Mpx/s]	f_{max} [MHz]	Clock rate [clk/px]	Number of units	Supported image	Image scans	FPS
Clienti [3]	4	arb. 3x3	403	100	0.25	16*	1024x1024	5	80
Chien [2]	1	disk 5x5	190	200	1.052	1	720x480	27	21.5
Déforges [5]	1	arb. convex	50	50	1	1*	512x512	11	17.2
This paper	6	rectangles	234	100	0.426	11*	1920x1080	1	113

* Number of available stages varies with size of used FPGA

– It uses strictly sequential access to the data at all algorithm levels.
– Low memory consumptions (far below the size of the image) allow embedding on a single chip complex operators able to process large images.
– Two levels of parallelism: i) the inter-operator parallelism in serial concatenations $\zeta = \delta\varepsilon \ldots \delta\varepsilon$, allow running all these atomic δ and ε operators simultaneously, and ii) the intra-operator parallelism in every atomic dilation/erosion. The intra-operator parallelism is scalable (tested up to six) and allows the decomposition of fast streams into several slower streams processed in parallel without altering the streaming property of the system.

The architecture serves as a basic building block to be used for construction of more complex operators such as ASF, granulometries, etc., with the same properties and performance. The performances obtained on an FPGA are approaching the 100Hz HDTV 1080p standard. These performances are far above what has been reported in the literature. 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. The number of examples includes the on-line production control, aging material defectoscopy, etc., wherever one requires processing of high resolution images and low latency.

Appendix: The 1-D Dilation Pseudocode

Algorithm 1: df←1D_DILATION (rp, wp, f, SE1, SE2, N)

Input: rp, wp - reading/writing position; f - input signal value $f(rp)$; SE1, SE2 - SE size towards left and right; N - length of the signal; Q - a FIFO-like queue
Result: output signal value $\delta_B f(wp)$

```
while Q.back()[1] ≤ f do
    Q.dequeue() ;                    // Dequeue useless valuess
Q.push({f, rp}) ;                    // Enqueue the current sample
if wp - SE1 > Q.front()[2] then
    Q.pop() ;                        // Delete too old value
if rp = min (N, wp + SE2) then
    return (Q.front()[1] ) ;         // Return valid value
else
    return ({}) ;                    // Return empty
```

References

1. J. Bartovský, E. Dokládalová, Petr Dokládal, and V. Georgiev. Pipeline architecture for compound morphological operators. In *ICIP10*, 2010.
2. 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*, 15(9):1156 – 1169, sept. 2005.
3. Ch. Clienti, S. Beucher, and M. Bilodeau. A system on chip dedicated to pipeline neighborhood processing for mathematical morphology. In EURASIP, editor, *EUSIPCO 2008*, Lausanne, August 2008.
4. A. Cord, D. Jeulin, and F. Bach. Segmentation of random textures by morphological and linear operators. In *8th ISMM*, pages 387–398, Oct. 2007.
5. 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.
6. K. I. Diamantaras and S. Y. Kung. A linear systolic array for real-time morphological image processing. *J. VLSI Signal Process. Syst.*, 17(1):43–55, 1997.
7. P. Dokládal and E. Dokládalová. Computationally efficient, one-pass algorithm for morphological filters. *Journal of Visual Communication and Image Representation*, 22(5):411–420, 2011.
8. E.R. Dougherty. *Mathematical morphology in image processing* . Taylor and Francis, Inc., 1992.
9. J. Gil and M. Werman. Computing 2-d min, median, and max filters. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(5):504–507, 1993.

10. J.-C. Klein and J. Serra. The texture analyser. *J. of Microscopy*, 95:349–356, 1972.
11. D. Lemire. Streaming maximum-minimum filter using no more than three comparisons per element. *CoRR*, abs/cs/0610046, 2006.
12. 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.
13. 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.*, 25(1):79–93, 2000.
14. P. Maragos. Pattern spectrum and multiscale shape representation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 11(7):701–716, 1989.
15. G. H. Mealy. A method for synthesizing sequential circuits. *Bell Systems Technical Journal*, 34:1045–1079, 1955.
16. L. Najman and H. Talbot, editors. *Mathematical Morphology: From Theory to Applications*. ISTE Ltd and John Wiley & Sons Inc, 2010.
17. R. Sabourin, G. Genest, and F. Prêteux. Off-line signature verification by local granulometric size distributions. *IEEE Trans. Pattern Anal. Mach. Intell.*, 19(9):976–988, 1997.
18. J. Serra. *Image Analysis and Mathematical Morphology*, volume 1. Academic Press, New York, 1982.
19. J. Serra. *Image Analysis and Mathematical Morphology*, volume 2. Academic Press, NY, 1988.
20. J. Serra and L. Vincent. An overview of morphological filtering. *Circuits Syst. Signal Process.*, 11(1):47–108, 1992.
21. F. Y. Shih, T. K. Chung, and C. C. Pu. Pipeline architectures for recursive morphological operations. *IEEE Trans. Image Processing*, 4(1):11 –18, jan. 1995.
22. P. Soille, E. Breen, and R. Jones. Recursive implementation of erosions and dilations along discrete lines at arbitrary angles. *IEEE Trans. Pattern Anal. Mach. Intell.*, 18(5):562–567, 1996.
23. S. Sternberg. Grayscale morphology. *Comput. Vision Graph. Image Process.*, 35(3):333–355, 1986.
24. E. R. Urbach and M. H. F. Wilkinson. Efficient 2-D grayscale morphological transformations with arbitrary flat structuring elements. *IEEE Trans. Image Processing*, 17(1):1 –8, jan. 2008.
25. M. Van Droogenbroeck and M. J. Buckley. Morphological erosions and openings: Fast algorithms based on anchors. *J. Math. Imaging Vis.*, 22(2-3):121–142, 2005.
26. M. van Herk. A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels. *Pattern Recogn. Lett.*, 13(7):517–521, 1992.
27. 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.
28. L. Vincent. Granulometries and opening trees. *Fundamenta Informaticae*, 41(1-2):57–90, January 2000.
29. Xilinx. Virtex-5 family documentation, 2009, available at http://www.xilinx.com/support/documentation/virtex-5.htm.
30. J. Xu. Decomposition of convex polygonal morphological structuring elements into neighborhood subsets. *IEEE Trans. Pattern Anal. Mach. Intell.*, 13(2):153–162, 1991.
31. X. Zhuang and R. M. Haralick. Morphological structuring element decomposition. *Computer Vision, Graphics, and Image Processing*, 35(3):370–382, 1986.