GPU Implementation of Linear Morphological Openings with Arbitrary Angle

Pavel Karas, Vincent Morard, Jan Bartovsky, Thierry Grandpierre, Eva Dokladalova, Petr Matula, Petr Dokládal

▶ To cite this version:

Pavel Karas, Vincent Morard, Jan Bartovsky, Thierry Grandpierre, Eva Dokladalova, et al.. GPU Implementation of Linear Morphological Openings with Arbitrary Angle. Journal of Real-Time Image Processing, Springer Verlag, 2015, 10 (1), pp.27-41. <10.1007/s11554-012-0248-7>. <hal-00680904>

HAL Id: hal-00680904

https://hal-upec-upem.archives-ouvertes.fr/hal-00680904

Submitted on 31 May 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)

Pavel Karas · Vincent Morard · Jan Bartovský · Thierry Grandpierre · Eva Dokládalová · Petr Matula · Petr Dokládal

GPU Implementation of Linear Morphological Openings with Arbitrary Angle

Received: date / Revised: date

Abstract Linear morphological openings and closings are important non-linear operators from mathematical morphology. In practical applications, many different orientations of digital line segments must typically be considered. In this paper, we (1) review efficient sequential as well as parallel algorithms for the computation of linear openings and closings, (2) compare the performance of CPU implementations of four state-of-the-art algorithms, (3) describe GPU implementations of two recent efficient algorithms allowing arbitrary orientation of the line segments, (4) propose, as the main contribution, an efficient and optimized GPU implementation of linear openings, and (5) compare the performance of all implementations on real images from various applications. From our experimental results, it turned out that the proposed GPU implementation is suitable for applications with large, industrial images, running under severe timing constraints.

Keywords morphology, opening, closing, linear, 1-D SE, parallel, efficient, algorithm, implementation, GPU

1 Introduction

Openings and closings are non-linear image operators of mathematical morphology [1]. They are at the basis of statistical measures called granulometries [2–5] and of a class of non-linear morphological filters called Alternate Sequential Filters (ASF) [6, 7].

In practical applications, the granulometries allow estimation of a priori unknown geometrical characteristics of objects in the image. For illustration, we can cite (1) medical imaging applications e.g., blood cell classification [8], (2) automated document analysis [9], or (3) industrial control [10]. The role of the ASF filters is to reduce the noise while preserving the principal features in the image. They represent the principal element of numerous applications e.g., texture analysis [11] or remote sensing [12]. Even the morphological openings themselves are useful for their filtering properties in some industrial applications such as [13].

Generally speaking, to obtain the desired result–size distribution or filtering effect–we have to use a sequence of openings and/or closings with varying parameters of the applied computing window, so-called structuring element (SE). For a given shape of the SE, these variable parameters are the progressively increasing size of SE and rotation angle. In order to ensure the exhaustivity of the result, applications often require computing of an enormous number of iterations with greater SE, often approaching hundreds of pixels. Considering continually increasing image resolution used in industrial applications, one can intuitively feel that it results in overwhelming requirements on the computing power. This is true even despite recent efficient algorithms [14, 15].

In this context, we study how to efficiently implement the above mentioned operators on graphics cards with the objective to reduce these computing requirements on the system. Initially, graphics cards were designed for graphics purposes only and were not programmable. Based on numerous parallel processors they were very powerful compared to their price. Current GPUs have passed the one Tera FLOPS barrier, and there is no need to use dedicated graphics languages any more since several frameworks have been developed for GPGPU[1] purposes: CUDA [16] by nVidia and OpenCL [17] by the Khronos Group are today

P. Karas
Centre for Biomedical Image Analysis, Faculty of Informatics, Masaryk University, Botanicka 68a, 60200 Brno, Czech Republic. E-mail: xkaras1@fi.muni.cz

V. Morard · P. Matula · P. Dokládal
Centre of Mathematical Morphology, Department Mathematics and Systems, Mines ParisTech, 35, rue St. Honoré, 77300 Fontainebleau Cedex, France. E-mail: {vincent.morard, petr.matula, petr.dokladal}@mines-paristech.fr

P. Matula
University of Heidelberg, BIOQUANT, IPMB, and German Cancer Research Center, Dept. Bioinformatics and Functional Genomics.

J. Bartovský · T. Grandpierre · E. Dokládalová
Laboratoire d'Informatique Gaspard-Monge, Equipe A3SI, Université Paris-Est, ESIEE Paris, 93162 Noisy-le-Grand Cedex, France. E-mail: {j.bartovsky, t.grandpierre, e.dokladalova}@esiee.fr

[1] General-purpose computing on graphics processing units

the most popular in the GPGPU community. Both are based on C language extensions. Notice that in this paper, we use the CUDA language for all presented implementations on GPU. Nevertheless, the principles remain the same when moving to a different language.

In our study, we focus on the morphological openings using linear SE under arbitrary angle, essential in a wide range of practical applications dealing with linear image structures such as fingerprint analysis, geosciences [18], and vessel segmentation [19].

The main novelty of this paper consists of an efficient and optimized GPU implementation of the algorithm by Bartovsky et al. [15], which turned out to be the most suitable for a parallel implementation on GPU. We also present performance comparisons and experimental results of all implementations on real data. It turned out that the proposed GPU implementation can compute linear openings for 180 directions of an image of size 640×640 pixels within 60 ms.We also show that the proposed implementation is significantly faster than the state-of-the-art implementation in the OpenCV_GPU library [20], especially for larger SEs.

In the remainder of this paper we start by the introduction of the mathematical background in Section 2. Section 3 reviews and compares the state of the art of efficient implementations for computing linear morphological openings of different orientations. Section 4 presents the first GPU implementation of two candidate algorithms by Bartovsky [15] and Morard [14]. Afterwards, the Section 5 describes the optimized implementation of Bartovsky algorithm, including the parallelism enhancement discussion. Section 6 illustrates the use of this efficient implementation in practical applications. It includes also the discussion of overall experimental results. Finally, the conclusions recall the main contributions of our work and briefly introduces its perspectives.

2 Basic Notions

Before defining morphological openings and closings, we define erosions and dilations for gray-scale images. Let a mapping f: $\mathbb{Z}^2 \to \mathbb{R}$ be a gray-scale discrete image, and $\mathscr{F}$ be the collection of all such images. The support used throughout this paper shall be a rectangular subset domain of $\mathbb{Z}^2$.

The erosion and dilation are mappings $\varepsilon_B, \delta_B : \mathscr{F} \to \mathscr{F}$, parameterized by the so-called structuring element (SE) B, $B \subset \mathbb{Z}^2$. Here, we restrict the family of SE to flat and translation-invariant. Hence, the dilation and erosion are, as usually, defined by

$$\delta_B(f) = \bigvee_{b \in B} f_b \,; \qquad \varepsilon_B(f) = \bigwedge_{b \in \widehat{B}} f_b \tag{1}$$

where the hat $\widehat{\ }$ denotes the transposition of the structuring element, i.e., $\widehat{B} = \{x \mid -x \in B\}$, and f_b denotes the translation of the function f by some vector b, and $\bigvee$ and $\bigwedge$ denote the supremum and infimum on a collection of functions.

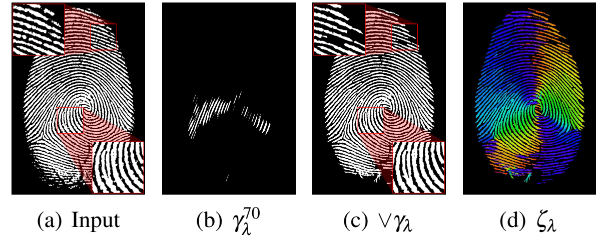


Fig. 1 Example application of linear openings. **(a)** input image, **(b)** linear opening with a digital line segment of length $\lambda = 41$ pixels and orientation 70 degrees, **(c)** enhancement of linear structures, and **(d)** color coded local orientation of linear structures calculated using definitions in Eq. (3). In **(a)** and **(c)**, two zoomed regions are shown for a better comparison.

Similarly, the morphological openings and closings are mappings γ_B, φ_B: $\mathscr{F} \to \mathscr{F}$, also parameterized by B. The transposition $\widehat{B}$ used in the definition of erosion ensures that the dilation and erosion form a so-called adjunction pair. This allows us to obtain the morphological opening and closing by concatenation of the dilation and erosion:

$$\gamma_B(f) = \delta_B[\varepsilon_B(f)] \,; \qquad \varphi_B(f) = \varepsilon_B[\delta_B(f)]. \tag{2}$$

Erosions and dilations are dual under complementation, i.e. for functions $\delta_B = -\varepsilon_B(-f)$, see e.g. [21]. Consequently, γ_B and φ_B are also dual and all algorithms developed for openings can easily be used also for closings.

In the sequel, we focus on linear morphological openings and closings obtained with SE in a form of a 1-D digital line segment of the length λ and orientation α, denoted by γ_λ^α and φ_λ^α, respectively.

Further operators can be built based on linear openings. The first operator, $\vee\gamma_\lambda(f)$, is computed by taking the supremum of the openings by digital line segments in all orientations; the second operator, $\zeta_\lambda(f)$, can extract the local orientation of linear structures:

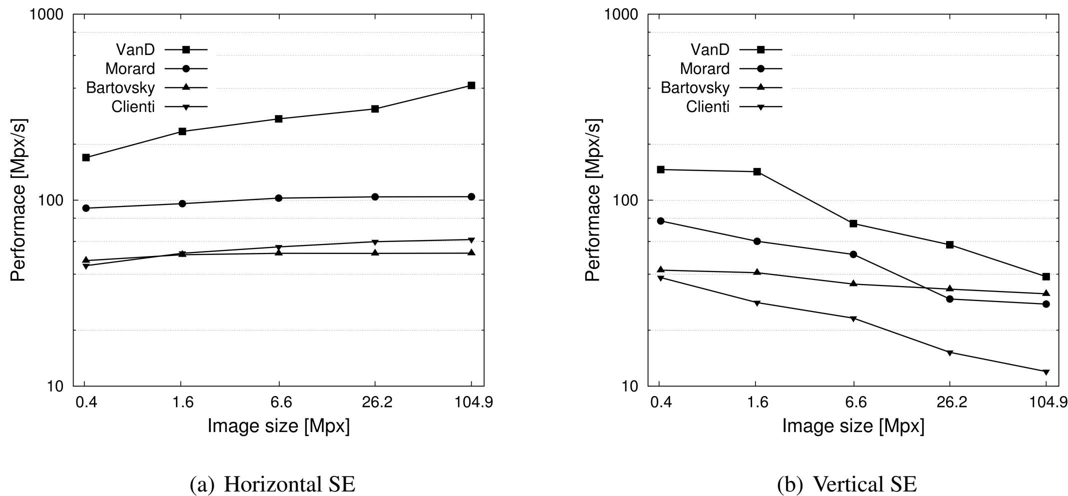
$$\vee\gamma_\lambda(f) = \bigvee_{\alpha \in [0,180]} \gamma_\lambda^\alpha(f) \,; \qquad \zeta_\lambda(f) = \arg \bigvee_{\alpha \in [0,180]} \gamma_\lambda^\alpha(f). \tag{3}$$

As an example, the effect of these operators on a real image is presented in Fig. 1.

3 Opening Algorithms

Opening algorithms can be divided into three classes: erosion and dilation chaining (a two-stage algorithm), direct computation, and algorithms based on connected component tree building.

Two-stage algorithm: computed by using Eq. (2), it is the simplest and historically earlier approach used to compute openings. Noting N the number of pixels of the image



(a) Horizontal SE

(b) Vertical SE

Fig. 2 Comparison of CPU implementations for horizontal and vertical SEs of size approx. 5% of the image width.

and L the size of the SE, the complexity of the naïve approach is $O(N \times L)$. The computational complexity was improved throughout the years to make the algorithm feasible for practical applications. In 1985, Pecht [22] defined a logarithmic decomposition of the SE. This decomposition, which removes most of the redundancy, was further extended by Coltuc [23], reducing the complexity to $O(N \times \log L)$. Later, the complexity was further reduced to linear $O(N)$, hence independent of the size of the structuring element, by Van Herk, and Gil and Werman [24, 25]. The linear algorithm will be referred to as the *HGW algorithm* hereafter. In [26], Soille et al. extended this work to arbitrary-oriented openings. In [27], Clienti et al. improved the HGW algorithm by removing the image backward scanning to reduce latency.

Direct Computation: for the family of openings with 1-D SE, new algorithms were introduced recently to directly compute openings in only one scan of the entire image, preserving the complexity of $O(N)$. In [28], Van Droogenbroeck and Buckley introduced an algorithm based on anchors, allowing very fast computation of the linear openings. The anchors are points, which are not affected by the opening. Nevertheless, the algorithm uses a histogram. The main drawback of using a histogram is the increasing memory consumption for finely quantized data. Even though the memory is no longer an issue for PC architectures, it becomes a penalizing factor for parallelized implementations on other architectures with limited memory like GPU. Secondly, search over a long histogram becomes costly and finely, for floating point accuracy, the histogram can not be used at all.

Later, Morard et al. [14] introduced a very simple algorithm based on an ordered stack of *cords* where a cord refers to a continuous set of pixels of intensity greater than or equal to a certain value I. With the inclusion relation between cords and by analyzing the length of each cord, the computation of linear openings and of linear granulometries is straightforward. Finally, Bartovsky et al. [15] also developed an algorithm to build linear openings. It sequentially scans the signal and erases every peak narrower than SE. Further explanations on these algorithms are given in section 4.

Connected component tree: such approach was described in [29]. The approach is based on building connected components, hence it can be adopted for more complex tasks such as watershed segmentation [30]. The drawback of the algorithm is that the complexity depends on the number of gray levels in the image and requires random access data, consequently it is not adapted for applications running under strict time constraints.

3.1 Parallel Implementations

There are several implementations of HGW algorithm in the literature since it can be easily parallelized. Brambor [31] described a parallel implementation of the HGW algorithm on SIMD architectures. Their implementation was tested on an Intel CPU with the SIMD-SSE2 instruction set. Clienti [27] improved the HGW algorithm and implemented[2] it on a SIMD architecture as well. Domanski et al. [33] used CUDA to implement the HGW algorithm on GPU, achieving 13–33 $\times$ speedup. There are several drawbacks of the algorithm as it computes openings and closings by dilation-erosion chaining, which requires more computations than direct approaches. It also has larger memory requirements [34].

On the contrary, there are few parallel implementations of the component tree algorithms, among which we can cite Wilkinson [35], Menotti-Gomes [36] on multicore, and also Matas [30] on ccNUMA 4-core. They are effectively

[2] available in Fulguro image processing library [32]

so complex that it is difficult to exhibit some parallelism in these complex algorithms.

Finally, in order to improve the computing efficiency by parallel implementation, direct computation algorithms seem to be the best candidates compared to HGW and component-tree algorithms. This is why we focus on this class of algorithms in the remainder of this paper.

3.2 Selection of a Direct Linear Opening Algorithm

We need to select the best sequential algorithm candidate for a parallel implementation leading to the most efficient execution on a GPU platform. As explained above, such an algorithm has to allow arbitrary angles computing using direct linear opening for lower computation and complexity requirements. Hence, there are three algorithms available to benchmark and compare: (1) Van Droogenbroeck [28] referred to as *VanD*, (2) Morard et al. [14] referred to as *Morard* and (3) Bartovsky et al. [15] referred to as *Bartovsky*. To this list, we will add Clienti algorithm [27] although it is not a direct computation based algorithm. Effectively, this is one of the fastest HGW implementation that can consequently give a useful comparative point. It will be referred to as *Clienti*.

For a fair comparison, all these algorithms were implemented using the same image-processing library with the same interface and with all the optimization flags turned on. We used only one core of an Intel Core i7-870 2.93 GHz CPU for this benchmark. These algorithms have been applied on the texture images shown in Fig. 12(a), (b).

Execution performances of the four algorithms for both horizontal and vertical openings are presented in Fig. 2. The performance P is computed as $P = N/t$, where N is the image size and t is the computation time. This measure is consequently independent of the image size. Note, however, that the performance actually does depend on the image size (see e.g. Fig. 2(b)). Each marked value in the plots represents the performance computed from the mean computation of 100 openings. For each image the length of a structuring element was chosen to be equal to 5 % of the image width/height because, in most applications, the length of SE is considerably smaller than the image size.

From the benchmarks we see that for vertical SEs and for large images, the performances significantly decreases for all algorithms, except for the Bartovsky. This is explained by the fact that this algorithm accesses the data sequentially even for vertical structuring elements. It is clear that VanD algorithm is the fastest, followed by Morard, Bartovsky, and Clienti, for both vertical and horizontal orientation. While VanD achieves the best performance, it is nevertheless unsuitable for parallelization on a GPU because of its high memory requirements especially for higher bit depths and arbitrary oriented SEs. In contrast, Morard and Bartovsky algorithms are able to compute openings and closings efficiently regardless the orientation of the SE or the data type precision with minimum memory requirements. Therefore, we selected these two algorithms for parallelization and implementation onto the targeted GPU architecture.

Algorithm 1: Morard algorithm: $G \leftarrow \text{Open1D}(F, \lambda, \text{S})$

```
Input: F – input 1-D signal, λ – size of SE, S – pointer to
       LIFO stack
Result: G – output 1-D signal
Data: S – a stack of triplets (value, position, flag)
initialize S;
for rp ← 0 to |F| − 1 do
    if S.empty() or F[rp] > S.top(value) then
        S.push({F[rp], rp, false});
    else
        while F[rp] < S.top(value) do
            fz ← S.pop();
            if fz(passed) or rp − fz(position) ≥ λ then
                WriteFlatZones(F, G, rp, S, fz);
            process S;
process all zones remaining in S;
```

4 Basic Implementation on GPU

4.1 Morard and Bartovsky Algorithms

In this section, we briefly introduce the Morard and Bartovsky algorithms for computation of morphological openings and closings with a linear SE. The detailed description can be found in [14,15]. Both algorithms are able to compute the operation in O(N) time with respect to the image size and O(1) with respect to the size of SE. The Bartovsky algorithm was originally designed for streaming architectures such as FPGA and hence performs scanning of the input data in sequential order. Nevertheless, it can be simply modified to perform scanning along lines according to the orientation of SE, much like the Morard algorithm. During the scan, intensity and position of each pixel is stored in an auxiliary data structure if necessary. Whereas the Morard algorithm uses the LIFO stack, the Bartovsky algorithm uses the FIFO queue. For arbitrary directions, both algorithms use the Bresenham's lines, as described in [26].

In the Morard algorithm, the image line is scanned for pixels where the intensity changes. Whenever the current pixel intensity is higher than the preceding, the current pixel is pushed to the stack. In the opposite case, the stack is being emptied while necessary. The algorithm needs to store an extra bit for a boolean flag indicating the status of a pixel. The size of the stack is limited only by the size and the bit depth of the image. After processing the stack, the output is written. The outputs are irregular. A pseudo-code is shown in Alg. 1.

In the Bartovsky algorithm, the image line is scanned for so-called peaks. According to a peak configuration, either a pixel is pushed to the queue or the queue is being emptied. The size of the queue is limited by the size of SE. After

Algorithm 2: Bartovsky algorithm: $G \leftarrow \text{Open1D}(F, \lambda, Q)$

Input: F – input 1-D signal, λ – size of SE, Q – pointer to FIFO queue
Result: G – output 1-D signal
Data: Q – a double-end queue of pairs (*value*, *position*)

```
initialize Q;
for rp ← 0 to |F| − 1 do
    while F[rp] ≤ Q.back(value) do
        process Q;
    Q.push({F[rp], rp});
    if rp = Q.front(position) + λ then
        Q.pop();
    if rp ≥ λ then
        G[rp − λ] ← Q.front(value);
```

processing the queue, the output is written. The writes are regular and follow the reads with the well-defined latency corresponding to the size of SE. A pseudo-code is shown in Alg. 2.

4.2 GPU Parallel Architecture

Before we describe basic GPU implementations of the algorithms, we briefly review some properties of the GPU architecture, as shown in Fig. 3.

From the hardware point of view, a GPU consists of hundreds of so-called *streaming multiprocessors (SM)*. Each includes a number of *shader processor (SP)* cores, a number of *registers*, a small *shared memory* providing communication between SPs, and fast *ALU units* for hardware acceleration of transcendental functions. One *global memory* is shared by all SMs and provides a capacity in order of GB and the memory bandwidth in order of 100 GB/s. There are also two additional read-only cached memory spaces accessible by all threads: the *constant* and *texture* memory spaces. They can help programmers to improve the performance of their implementations [37,38]. In some recent GPU architectures, such as FERMI by nVidia [39], the global memory is cached as well.

From the programmer's point of view, every program consists of two parts, a *host code* for CPU, and a *kernel code* for GPU. Before executing a kernel, the host program allocates a memory on GPU and transfers data if necessary. Then a kernel is configured and executed. The configuration defines the number of threads allocated for kernel execution. Generally, the number of threads should be much higher than number of processing units of GPU. This approach allows (1) the proper scaling on various hardware configurations, and (2) hiding the memory latencies. The threads form groups called *blocks* (as in CUDA [37]) or *work-groups* (as in OpenCL [40]), following the hierarchy of SMs and SPs. Synchronization of threads and data sharing is possible within the group only. The threads are executed concurrently in *warps*, usually of 32 threads each.

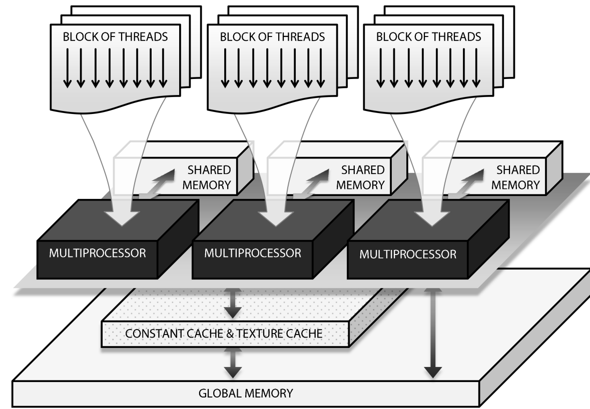


Fig. 3 Thread mapping and memory hierarchy in the GPU architecture.

4.3 GPU Implementation of Morard and Bartovsky Algorithms

Regarding the design of both algorithms, the input image is scanned in the sequential manner. However, all lines of the image can be scanned concurrently, as shown in Fig. 4. Thus, the parallelism can be introduced by binding individual threads to individual lines of the image. Each image line is processed independently using its own algorithm-dependent stack or queue, respectively. This requires the GPU memory to be large enough to contain all auxiliary data structures. Here, the Bartovsky algorithm is favorable, since the sizes of the queues used are limited to the size of SE whereas the stacks used by Morard algorithm are generally limited by the size of the image.

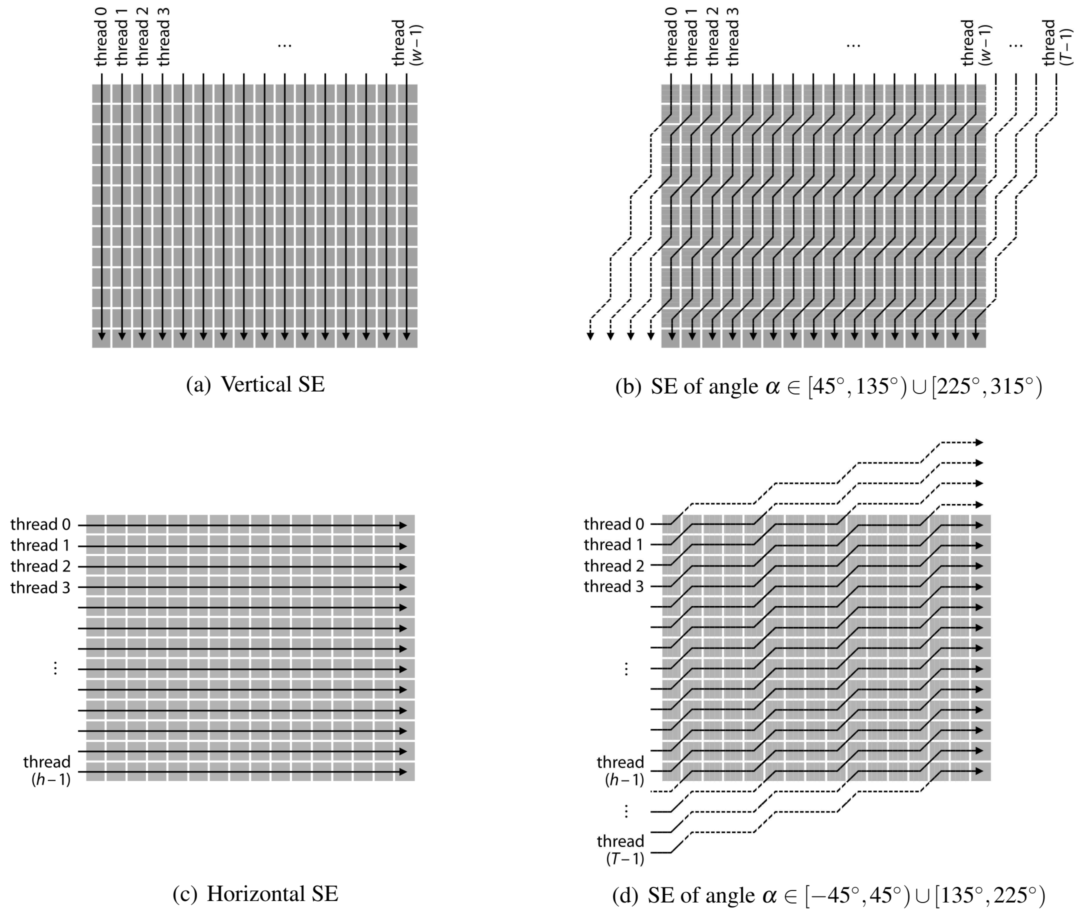
The mapping of threads for all SE orientations is described in Fig. 4. For an arbitrary angle α, the overall number of threads can be simply computed as follows:

$$T = w + \lceil h|\cot\alpha|\rceil,\ \alpha \in [45°, 135°) \cup [225°, 315°), \quad (4a)$$

$$T = h + \lceil w|\tan\alpha|\rceil,\ \alpha \in [-45°, 45°) \cup [135°, 225°), \quad (4b)$$

where w and h are the width and the height of the input image, respectively. Thus, generally $T > w$ or $T > h$, respectively. Some of threads spend a part of the computation time outside the image domain where they do not compute anything. Thanks to thread locality, this brings little overhead only because in the GPU thread scheduler, thread warps that fall outside the image domain are quickly replaced by those that fall inside.

It should be noted that the processing of both the stack and the queue is data-dependent, thus data accesses are irregular. Therefore, during this part of the computation, threads are divergent. This limits the overall performance of the parallel implementation. In the Bartovsky algorithm, the data accesses to both input and output images are regular.



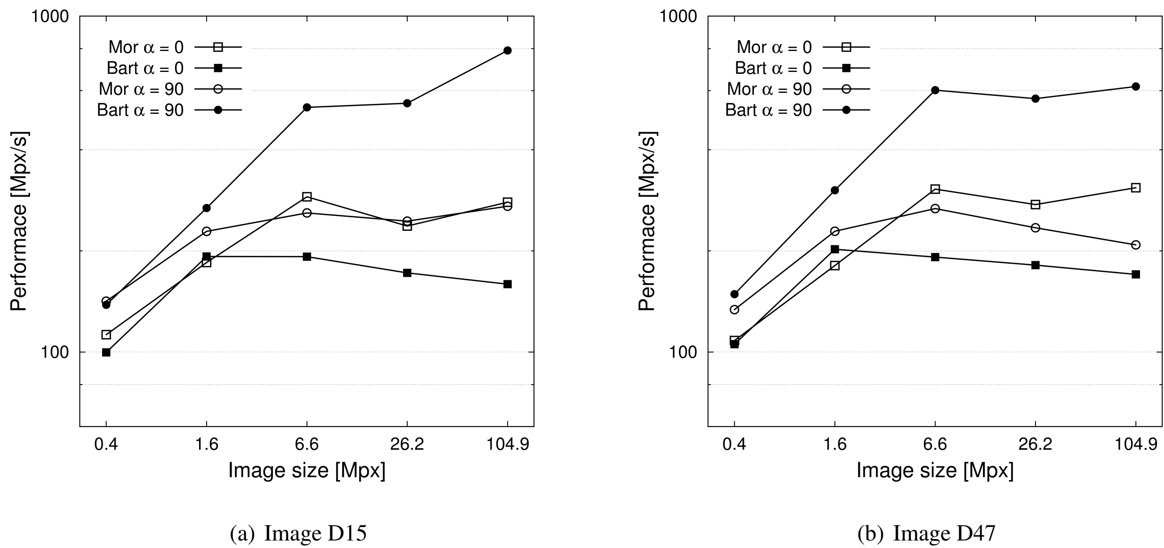
(a) Vertical SE

(b) SE of angle $\alpha \in [45°, 135°) \cup [225°, 315°)$

(c) Horizontal SE

(d) SE of angle $\alpha \in [-45°, 45°) \cup [135°, 225°)$

Fig. 4 The mapping of threads to the 2-D image grid in the GPU implementation of the algorithm for computing 1-D morphological openings/closings. Each thread, denoted by its ID, is mapped to an individual image column or row, respectively.



(a) Image D15

(b) Image D47

Fig. 5 Comparison of basic GPU implementations of Morard and Bartovsky algorithms. Opening was computed with SE of both horizontal and vertical direction and size approx. 5 % of image width.

4.4 Experimental Results

The basic GPU implementations, described in the previous section, were compared on nVidia Tesla C2050 GPU with 14 MPs at 1.15 GHz and 3 GB RAM. Again, all experiments were made with the texture images shown in Fig. 12(a), (b). Results are shown in Fig. 5. The presented performance does not include the times needed for the data transfer between CPU and GPU.

Contrarily to the performance on CPU, the Bartovsky algorithm achieves better performance than the Morard algorithm. However, its performance is highly sensitive to the SE orientation, as shown in Fig. 5. For a horizontal SE, the performance is significantly lower due to extensive L1 cache misses, as detected by Nsight Visual Profiler [41]. As the L1 cache is stored in memory banks and memory accesses are synchronized, most of cache queries are directed to the same memory bank leaving other memory banks unused. This is not the case of the Morard algorithm since it generally keeps the memory accesses irregular.

5 Efficient Implementation of the Bartovsky Algorithm

The choice of the algorithm to optimize is guided by the analysis of the Bartovsky and Morard algorithms in sections 4.3 and 4.4. Despite higher performances of the Morard algorithm on CPU, the Bartovsky algorithm has several advantages:

(1) Both the data accesses to the input and output image are regular. This helps to make the thread execution synchronous and reduces the thread divergence.

(2) The maximum length of the FIFO queue is limited by the length of SE. This strongly limits the memory requirements. Recall that we are to bind one thread per image line (Fig. 4). For large images, with potentionally many threads, it is important to have a small memory footprint of each thread.

In the basic implementation on GPU (contrarily to CPU), the Bartovsky algorithm is sensitive to the SE orientation. Hence, we shall introduce several optimization steps not only to increase the overall performance but, particularly, to keep the performance stable for all orientations. These steps are described in the following sections.

To prove the choice of the Bartovsky algorithm, we applied the optimization steps also on the Morard algorithm and performed tests to compare both optimized GPU implementations. The results are shown in Section 6.

5.1 Parallelism Enhancement

In the basic implementation, the parallelism was introduced by mapping GPU threads to individual image rows or columns, creating the grid of h or w threads, respectively (where h and w are height and width of a 2-D input image, respectively). However, if an input image is not large enough, the GPU's MPs are not fully occupied. Therefore, we introduce more parallelism by splitting the image into two or more parts. In the following text, we refer to them as *zones*. As each pixel can be affected by $(2\lambda-2)$ neighboring pixels (where λ is the length of SE), the zones need to overlap by $2(\lambda-1)$ pixels. An example of the image split for a vertical SE is shown in Fig. 6.

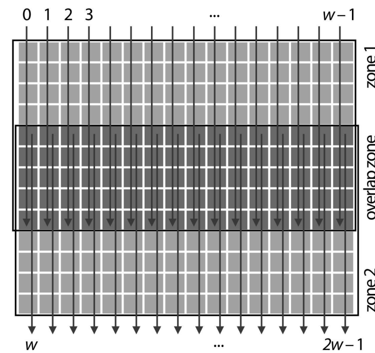


Fig. 6 Image split applied for opening/closing with vertical SE of size 2. Image is split into 2 zones, introducing twice more threads. The threads are denoted by vertical arrows, analogously to Fig. 4.

It is evident that the theoretical speed-up s that can be achieved depends on the SE length λ, the image size and the number of zones Z. For vertical SE we get the following:

$$s = Z\left[1 - \frac{(Z-1)(2\lambda-1)}{h}\right], \qquad (5)$$

where h is the image height. For small λ we get $s \approx Z$, while for $\lambda \approx h/(2Z)$ we get $s \approx 1$. It should be noted that for large input images it is not necessary to introduce more parallelism by splitting the image, it can even decrease the performance. The optimal choice of the parameter Z is discussed in section 5.4.

5.2 Optimization of Data Accesses

To reduce the memory latency, the usage of the global memory should be avoided where possible. Alternative memory spaces can be used for both the input image and the FIFO queues. The input image can beneficially be stored in the read-only cached texture memory. This is true also for the recent FERMI architecture, which introduced L1 cache [39, 42], because the texture memory helps significantly improve the performance for horizontal SEs where L1 cache fails due to bank conflicts. The FIFOs can be stored in the shared memory. As the amount of the available shared memory is limited, the maximum length l_{max} of the FIFO is limited to $l_{max} = S/(S_b \times d)$, where S is the amount of the shared memory available, S_b is the number of threads per block (workgroup) sharing the memory, and d is the size of the data type. Hence, the most recent values of the FIFO are stored in the shared memory in a circular buffer so the position of the first element varies during the computation. The rest of the FIFO

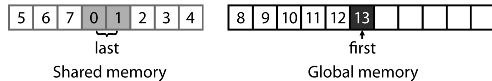


Fig. 7 FIFO storage in the shared and the global memory. The numbers indicate the order of pushing the element into the FIFO. The marked elements (the first one and the last two) are also cached in registers.

is stored in the global memory. The FIFO storage is shown in Fig. 7.

The constraint on the FIFO length mentioned above can be considered as a hard limit. However, using as much shared memory as possible can lead to a performance decrease because in that case, the shared memory is allocated for one thread block per MP only. Since the MPs are capable of execution of more thread blocks, the optimal FIFO length needs to be chosen carefully, as discussed further.

To conclude our choice of memory spaces, using L1 and L2 caches for input data turned out to be inefficient for some SE orientations, as explained in Section 4.4. The texture memory is read-only, cached, and can be allocated up to the size of the device memory. Therefore, it is suitable for the input image. The size of the texture cache is relatively small (8 kB per multiprocessor) but sufficient, thanks to the spatial locality. The shared memory is rewritable, relatively small (up to 48 kB per MP), and fast. In terms of latency, it is comparable with registers, provided that block conflicts are avoided—which is guaranteed in our implementation. Thus, it is suitable for FIFO caching.

5.3 Consideration of Arbitrary Orientations

As noted in section 4.4, the performance of Bartovsky algorithm on GPU is sensitive to SE orientation. Thus, a careful attention should be paid to this issue when computing openings in arbitrary directions. By using the texture memory, the input image is cached and the latency of memory reads is reduced. The final step is to optimize the memory writes to the output image. Experiments proved that openings with SEs of orientation $\alpha \in (-45°, 45°)$ are computed faster when writing the output image transposed. The output image is subsequently corrected using the modified *transpose* kernel from [43].

5.4 Configuration of Performance Parameters

In the optimized GPU implementation, we have introduced several parameters that influence the performance: the block size (work-group size) S_b, the number of zones Z, and the optimal length l of the FIFO buffer allocated in the shared memory. They all depend on these properties: the image dimensions (w or h), the SE length λ, the number of the MPs N_MP available on the used device, the shared memory size per MP S, the warp size W and the number of blocks (work-groups) N_b that can be executed on a single MP. For optimal configuration, the following set of rules should be satisfied:

1. S_b should be multiple of W,
2. $Z \geq (S_\text{b} \times N_\text{b} \times N_\text{MP})/T$ where T is defined in Eq. (4),
3. $Z \leq h/(2\lambda)$ or $Z \leq w/(2\lambda)$ following Eq. (5),
4. $l \leq S/(S_\text{b} \times d \times N_\text{b})$,
5. l should be a power of two allowing the bitwise operator "&" to be used instead of the costly modulo operator for addressing the FIFO queue items.

It should be noted that some of the rules cannot be satisfied in some cases. In particular, rules (2) and (3) can be conflicting for small input images. Performances for some parameter configurations are presented in the following section. CUDA programmers are advised to use a tool called "CUDA Occupancy Calculator" which can help to compute the optimal kernel configuration [43].

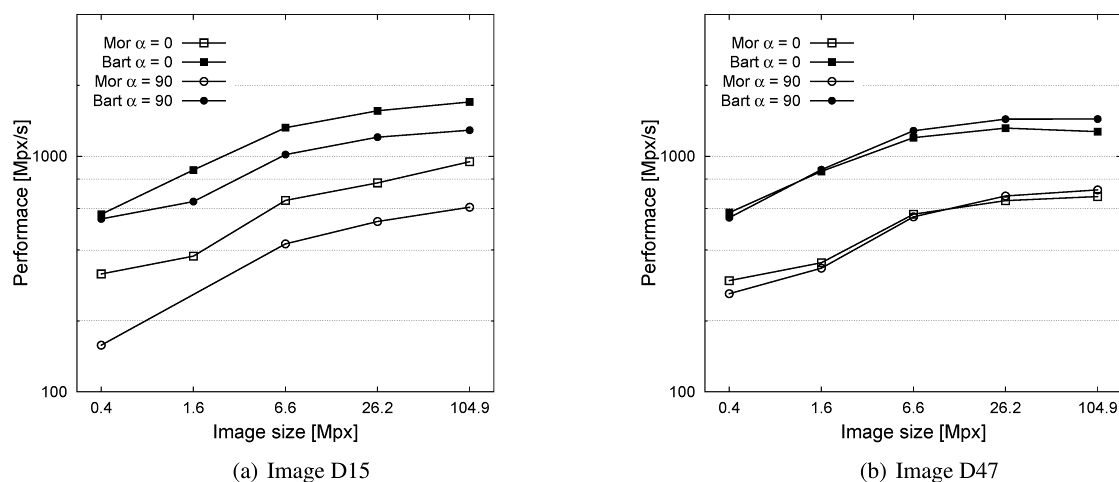
6 Experimental Results

We made the performance analysis of the optimized GPU implementations, based on images with clear linear structures (see Fig. 12), and the results of our comparisons are shown in Fig. 8, 9, and 10.

In the first experiment (Fig. 8), the comparison of optimized GPU implementations of Bartovsky and Morard algorithms is shown, in analogy to Fig. 5. We assumed that the former is more suitable for the GPU architecture than the latter. This assumption was confirmed by the experiments. Thus, in the following experiments, we used the more successful implementation.

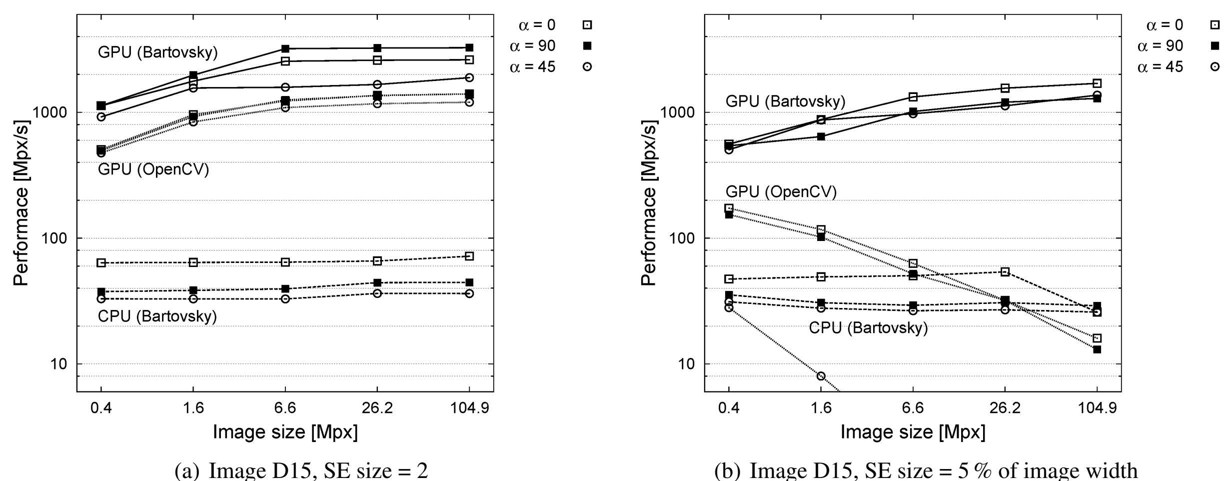
In the second experiment (Fig. 9 and 10), we compared the performance of our GPU implementation, referred to as "GPU (Bartovsky)", to the corresponding CPU implementation, referred to as "CPU (Bartovsky)", and also to the state-of-the-art implementation in the OpenCV library with the GPU support (so-called OpenCV_GPU) [20], referred to as "GPU (OpenCV)". It turns out that our GPU implementation is approximately 10–50 $\times$ faster than the CPU implementation, depending on the input data size and the length of the SE. Despite the fact that the Bartovsky algorithm itself is sequential so the parallelism introduced in its GPU implementation is limited, our implementation is faster than the OpenCV_GPU in every case. Whereas for small SEs the difference between the two GPU implementations is negligible, for larger horizontal and vertical SEs the speedup is up to 50 $\times$. For diagonal (and arbitrarily oriented) SEs, the performance of the OpenCV_GPU implementation falls down very quickly. This is because this implementation uses the NVIDIA Parallel Primitives (NPP) library [44] which supports only simple SE shapes, therefore the line SE has to be represented by a corresponding 2-D rectangle, i.e. a matrix with elements correctly set to 0 or 1.

The most important performance limit of our implementation is the number of threads that can be executed. If the



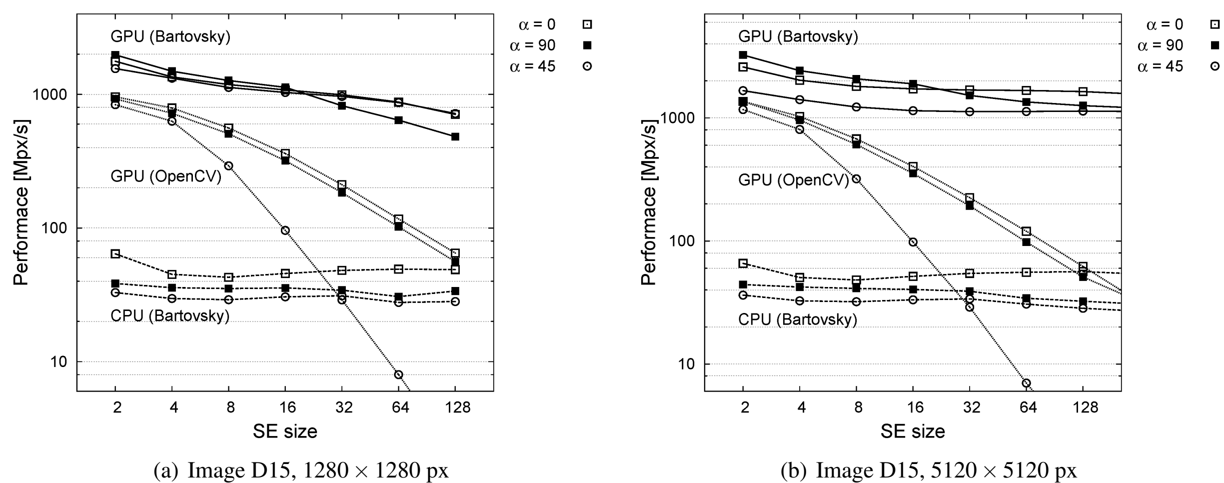
(a) Image D15 (b) Image D47

Fig. 8 Comparison of optimized GPU implementations of Morard and Bartovsky algorithms. For comparison with the basic GPU implementation, refer to Fig. 5.



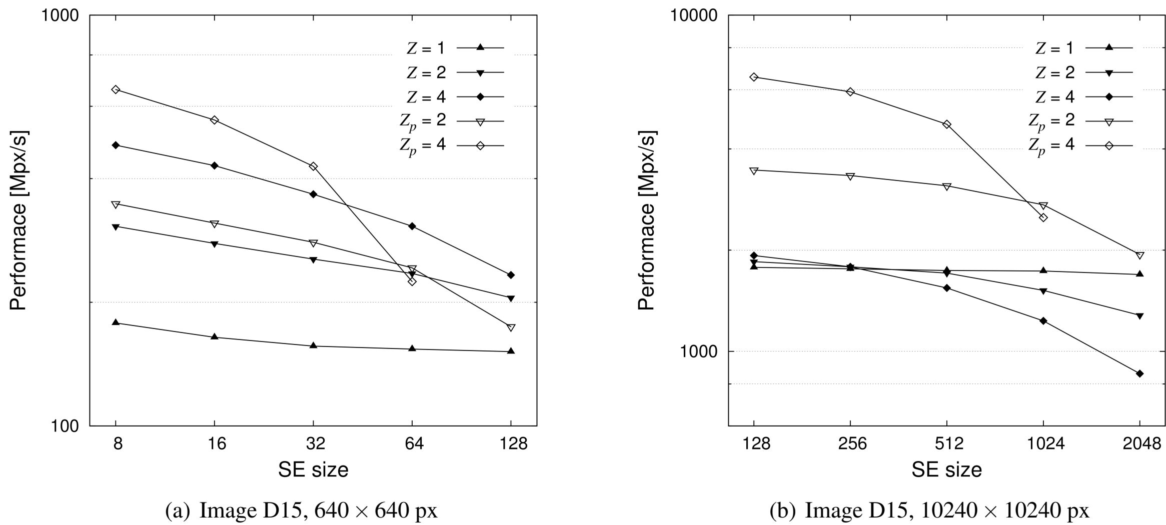
(a) Image D15, SE size = 2 (b) Image D15, SE size = 5 % of image width

Fig. 9 Comparison of CPU and GPU implementations for various image sizes, various SE orientations, and fixed SE sizes.



(a) Image D15, 1280×1280 px (b) Image D15, 5120×5120 px

Fig. 10 Comparison of CPU and GPU implementations for fixed image sizes, various SE orientations, and various SE sizes.



(a) Image D15, 640 × 640 px

(b) Image D15, 10240 × 10240 px

Fig. 11 The contribution of the image split into performance for various image sizes: $Z = 1$ means no split, $Z = 2$, $Z = 4$ means split into 2 and 4 zones, respectively, as described in 5.1; $Z_p = 2$, $Z_p = 4$ means theoretical speedup, as predicted in Eq. (5).

Table 1 Optimal kernel configuration for nVidia graphics cards and for a vertical SE. Choice of the optimal block size S_b, the Z parameter, and the size of the shared memory l depend on many parameters, as described in section 5.4. The Z parameter should be decreased if the SE size is too large. Compute capability refers to a core architecture of nVidia graphics cards [45].

Compute capability 1.x				Compute capability 2.x			
S_b	Image width w	Z	l	S_b	Image width w	Z	l
32	$< 64N_{MP}$	4	16	32	$< 64N_{MP}$	4	32
64	$< 128N_{MP}$	4	8	96	$< 192N_{MP}$	4	16
64	$< 256N_{MP}$	2	8	96	$< 384N_{MP}$	2	16
64	$\geq 256N_{MP}$	1	8	96	$\geq 384N_{MP}$	1	16

input image is too small, there is not enough threads and the GPU's is underused. This can be avoided by splitting the image in zones (refer to Fig. 6). Adjacent zones need to overlap to avoid border effects. For large SE sizes, the overlap becomes large, with the consequence that the performances decrease, see Fig. 10(a). For larger image sizes, the decrease is proportionally lesser, see Fig. 10(b).

Thanks to the optimizations described in section 5.3 we achieved stable performance for all SE orientations. The performance was tested for all $\alpha \in [0°, 180°)$, although for the sake of simplicity, the graphs show only three orientations. To conclude, our GPU implementation can be used for an arbitrary SE length and direction, achieving the performance more than 1000 Mpx/s. This allows computing one opening on a 40 Mpx image with any 1-D SE in any orientation.

The contribution of splitting the image into zones is shown in Fig. 11 and compared with a theoretical speedup, as computed by Eq. (5). For smaller images, the optimization increases the performance as expected. Note that for large SE sizes, the performance does not decrease as quickly as theoretically predicted. It is because the further parallelism introduced by the split not only occupies more MPs but it also helps to hide memory latencies. For larger images, the number of threads is large enough to occupy MPs, hence the image split does not introduce a further speedup.

The optimal choice of parameters for nVidia graphics cards and for a vertical SE is shown in Table 1. For AMD Radeon graphics cards, the optimum values may differ. For other orientations, the optimal parameters are analogous.

The performance values do not include the data transfers between CPU and GPU. According to our measurements, the time needed to transfer data is comparable with the time needed for the computation of a single opening, hence the overall speedup is half. Here, the GPU implementation is favorable for images larger than 6 Mpx. In the computation of multiple openings, the data transfer overhead is negligible. The performance values for our GPU implementation were obtained on top-class Tesla C2050 GPU (current price 1500 EUR), but we did several test also on 10 × cheaper GeForce GTX 470 GPU, and the results were comparable.

6.1 Practical Applications

In practice, linear openings and closings can be used for the detection of either local or global orientations of linear structures. Hence, we tested and compared two CPU implementations (Bartovsky, Morard) and our GPU implementation of linear openings and closings based on images from three different application domains, namely fingerprint analysis, texture characterization, and document analysis. In all cases, we computed a set of linear openings allowing massive parallelism on GPU simply by computing linear openings in all

Table 2 Comparison of CPU and GPU implementations for computation of an angular spectrum. The speedup is computed by comparing the GPU implementation with the best CPU (Morard) implementation.

Image	Image size	No. of angles	Time [ms] CPU (Bart)	CPU (Mor)	GPU (Bart)	Speedup GPU/CPU
Fingerprint	784×1133	180	3768.7	3129.2	115.9	27.0
D15	640×640	180	2425.8	2014.0	58.0	34.7
D47	640×640	180	2397.7	2014.0	55.1	36.6
Music1	1000×1411	81	5830.0	4840.7	129.3	37.4
Music2	1000×1301	81	7297.1	4958.8	136.5	36.3

directions concurrently avoiding the problem with insufficient MPs occupancy. We will show that this leads to a significant speed-up even for small input images. The benchmark results for all applications are shown in Table 2.

6.1.1 Local Orientation of Linear Structures in Fingerprint Images

The first application was the computation of local orientation of linear structures in a fingerprint image (Fig. 1). The operator $\zeta_\lambda(f)$ was computed according to Eq. (3). The GPU implementation achieves a significant speedup (approximately $27\times$) even for small input images (0.9 megapixels).

6.1.2 Angular Spectrum of Texture Images

The second application was the computation of the angular spectrum of texture images. The spectrum was computed in order to find the most important orientation(s) of linear structures in the image. Two test images along with their spectra are shown in Fig. 12. A spectrum $\sigma_\lambda(\alpha)$ of an image f is computed as follows:

$$\sigma_\lambda(\alpha) = \sum_{x \in \Omega(f)} \left[\gamma_\lambda^\alpha(f)\right](x). \qquad (6)$$

Again, the GPU implementation achieves a significant speedup (approximately $35\times$) for input images of 0.4 megapixels.

6.1.3 Rotation Detection of Music Sheet Scans

In the third practical application, we detected the rotation of music sheet scans. The music sheets were scanned and stored in an electronic archive. In the process of scanning, a random rotation could occur due to imperfect insertion of the paper to the scanner. The rotation was detected by computing linear closings with large SEs ($\lambda = 250$) of 81 different orientations within the angular range from -10° to 10° with the step of 0.25°. The angular spectrum was computed according to Eq. (6). Two test images along with their spectra are shown in Fig. 13. In this case, the achieved speed-up was approximately $37\times$.

7 Perspectives - Extention to 2-D

The 2-D opening is not separable into two orthogonal 1-D openings as is the dilation. Hence, one cannot directly combine two orthogonal 1-D openings to obtain a 2-D opening.

It is known, that efficient GPU accelerations can only be obtained with simple, regular threads, using as low memory as possible. Hence, the separability principle is useful. Following this idea, one can form 2-D openings by concatenating 2-D erosion and 2-D dilation which are separable. A 1-D dilation algorithm with alike properties as the Bartovsky algorithm has been published in [34].

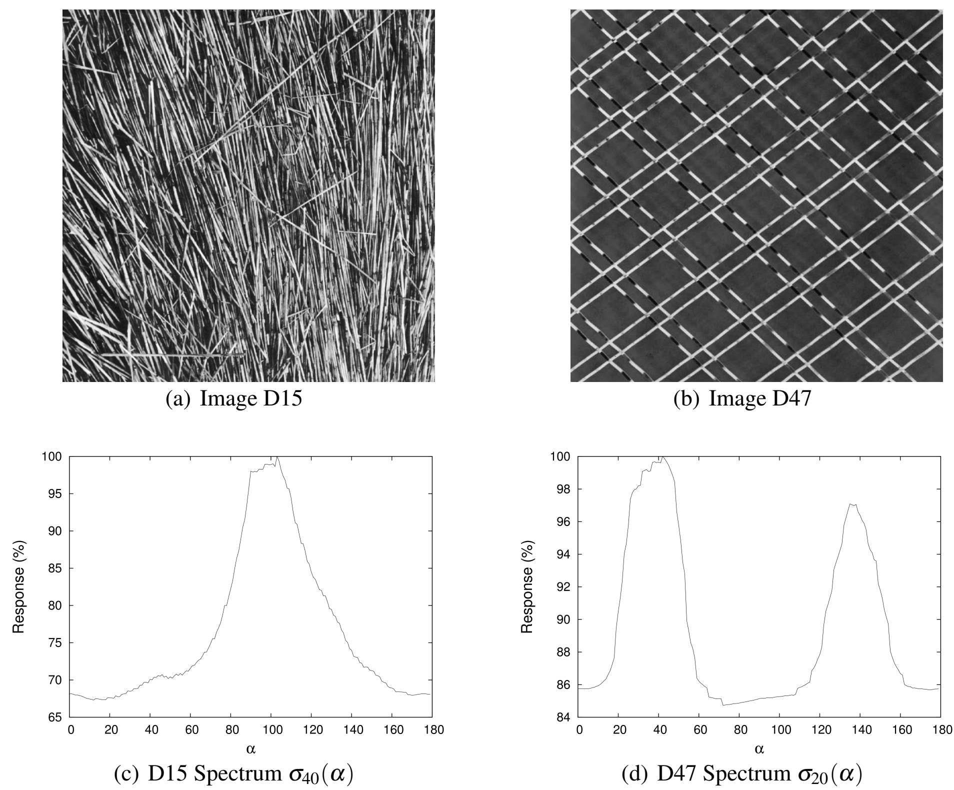
Assume a 2-D rectangular B in Eq. (1). It can be decomposed into two (horizontal and vertical) 1-D dilations and two 1-D erosions. Similarly, for hexagons we need to compute three erosions and three dilations, and for octagons four erosions and four dilations.

All these orthogonal operators need to be computed sequentially. One cannot expect to obtain the same performances in 2-D as with 1-D openings, since the execution times of the orthogonal operators are added together.

8 Conclusions

We have reviewed and compared the most efficient linear morphological opening/closing algorithms. At present, the fastest approaches (Van Droogenbroeck and Buckley, Morard et al., and Bartovsky et al.) compute the opening within a single image scan. The algorithm of Van Droogenbroeck and Buckley is the fastest one on CPU, however, it is efficient for 8-bit gray-scale images and for vertical and horizontal linear openings only. Morard and Bartovsky algorithms are applicable to any data accuracy (including floating point).

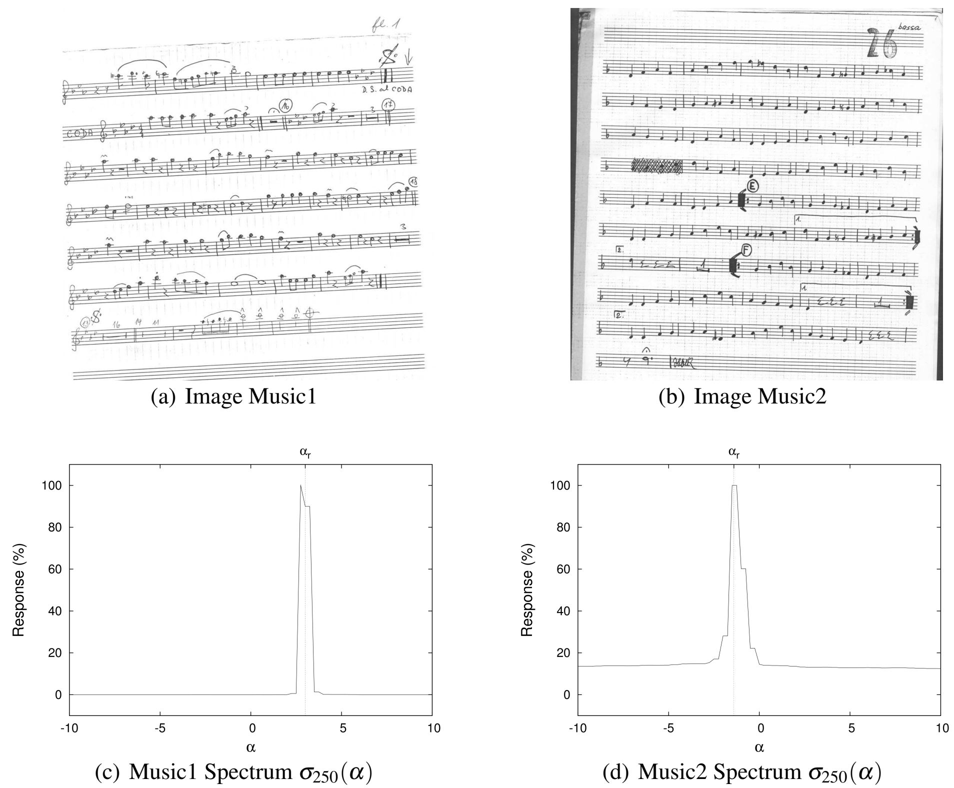
As described in the paper, both Morard and Bartovsky algorithms themselves are sequential. Hence, the only possibility of introducing more parallelism is on the thread execution level. We explain the choice of the algorithm (Bartovsky) to implement with regard to the GPU architecture (little memory, synchronous execution of threads). We have used various optimization techniques to speed up the code. Mapping various types of data (input, output and FIFOs) to various memory spaces is a crucial aspect. The choice of



(a) Image D15 (b) Image D47

(c) D15 Spectrum $\sigma_{40}(\alpha)$ (d) D47 Spectrum $\sigma_{20}(\alpha)$

Fig. 12 Texture images [46] and their angular spectra.



(a) Image Music1 (b) Image Music2

(c) Music1 Spectrum $\sigma_{250}(\alpha)$ (d) Music2 Spectrum $\sigma_{250}(\alpha)$

Fig. 13 Music sheet scans (courtesy of Josef Pilný, Big Band Lanškroun) and their angular spectra. The real (manually measured) rotation angle is denoted by α_r.

memory spaces is described in detail in Section 5.2. We also observed performance limits on small images, hence we try to introduce more spatial parallelism by splitting small images. Finally, we give rules of optimal configuration regarding input image size and the features of the available GPU.

We provide comparisons with the fastest implementations on CPU and on GPU. The current state-of-the-art standard, OpenCV_GPU, is suitable for GPU architecture but has time complexity dependent on the SE size. The proposed implementation obtained stable performance over all orientations and sizes of the structuring element. For a single opening of a large image, we have measured up to 50 $\times$ speedup compared with CPU. For small images, the gain is significant (up to 37 $\times$ speedup) if one computes a set of openings in multiple directions. For example, within 60 ms, the GPU implementation is capable of computing a single opening at arbitrary angle of a 60 Mpx image, or a set of openings in 180 directions of a 640$\times$640 px image.

To conclude, this solution is suitable for applications with large, industrial images, running under severe timing constraints, such as production control in e.g. metallurgy or textile industry. A typical such application requiring using high-resolution images, and running under severe time constraints is the surface control. Thin (often μm) cracks in large surfaces require using high resolution images, and the timing is given by the industrial cycle.

Source codes of the CPU and GPU implementations of the Bartovsky algorithm are publicly accessible under GNU GPL license [47].

Acknowledgements

This work has been done during ERASMUS stage of Pavel Karas at ESIEE Paris. The presented work is the result of collaboration between ESIEE Paris (A3SI team), Mines-ParisTech (CMM) and Masaryk University in Brno (CBIA).

This stay period of Pavel Karas at ESIEE Paris has been also supported by the Ministry of Education, Youth and Sport of the Czech Republic (Projects No. MSM-0021622419 and No. LC535). This research was also supported by the Grant Agency of the Czech Republic (Grant No. P302/12/G157).

References

1. J. Serra. Morphological filtering: An overview. *Signal Processing*, 38:3–11, 1994.
2. L. Vincent. Fast opening functions and morphological granulometries. In *SPIE*, volume 2300, pages 253–267, 1994.
3. S. Batman, E. R. Dougherty, and F. Sand. Heterogeneous morphological granulometries. *Pattern Recognition*, 33(6):1047 – 1057, 2000.
4. L. Vincent. Granulometries and opening trees. *Fundam. Inf.*, 41:57–90, January 2000.
5. E. R. Urbach, J. B. T. M. Roerdink, and M. H. F. Wilkinson. Connected rotation-invariant size-shape granulometries. *Pattern Recognition, International Conference on*, 1:688–691, 2004.
6. J. Serra and L. Vincent. An overview of morphological filtering. *Circuits, Systems and Signal Processing*, 11(1):47–108, 1992.
7. H. Heijmans. A new class of alternating sequential filters. In *I of Proceedings of 1995 IEEE Workshop on Nonlinear Signal and Image Processing*, pages 3–0, 1995.
8. N. Theera-Umpon and P. D. Gader. Counting white blood cells using morphological granulometries. *J. Electronic Imaging*, pages 170–177, 2000.
9. A. Bagdanov and M. Worring. Granulometric analysis of document images. In *Proceedings of the International Conference on Pattern Recognition*, volume I, pages 478 – 481, 2003.
10. S. Outal, D. Jeulin, and J. Schleifer. A new method for estimating the 3d size-distribution curve of fragmented rocks out of 2d images. *Image Analysis and Stereology*, 27(2), 2011.
11. D. Talukdar and R. Acharya. Estimation of fractal dimension using alternating sequential filters. In *Proceedings of the 1995 International Conference on Image Processing (Vol. 1)*, ICIP '95, Washington, DC, USA, 1995. IEEE Computer Society.
12. S. O. Sigurjonsson, J. A. Benediktsson, and J. R. Sveinsson. Street tracking based on sar data from urban areas. In *International Geoscience and Remote Sensing Symposium*, pages 1273–1276, 2005.
13. M. Kowalczyk, P. Koza, P. Kupidura, and J. Marciniak. Application of mathematical morphology operations for simplification and improvement of correlation of images in close-range photogrammetry. In *ISPRS08*, page B5: 153 ff, 2008.
14. V. Morard, P. Dokládal, and E. Decencière. Linear openings in arbitrary orientation in O(1) per pixel. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 1457–1460, May 2011.
15. J. Bartovský, P. Dokládal, E. Dokládalová, and M. Bilodeau. Fast streaming algorithm for 1-d morphological opening and closing on 2-d support. In P. Soille, M. Pesaresi, , and G.K. Ouzounis, editors, *ISMM 2011*, volume 6671 of *LNCS*, pages 296–305. Springer, July 2011.
16. nVidia Corporation. NVIDIA GPU Computing Developer Home Page. `http://developer.nvidia.com/category/zone/cuda-zone`, Jun 2011.
17. Khronos Group. OpenCL. `http://www.khronos.org/opencl/`, 2011.
18. 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*, 33:151–158, 2007.
19. F. Zana and J.-C. Klein. Segmentation of vessel-like patterns using mathematical morphology and curvature evaluation. *IEEE Transactions on Image Processing*, 10:1010–1019, 2001.
20. Willow Garage. OpenCV_GPU. `http://opencv.willowgarage.com/wiki/OpenCV_GPU`, Oct 2011.
21. P. Soille. *Morphological Image Analysis: Principles and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
22. J. Pecht. Speeding-up successive minkowski operations with bit-plane computers. *Pattern Recognition Letters*, 3(2):113–117, 1985.
23. D. Coltuc and I. Pitas. On fast running max-min filtering. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 44(8):660–663, 1997.
24. M. van Herk. A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels. *Pattern Recognition Letters*, 13(7):517–521, 1992.
25. J. Gil and M. Werman. Computing 2-d min, median, and max filters. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 15(5):504–507, 1993.
26. P. Soille, E.J. Breen, and R. Jones. Recursive implementation of erosions and dilations along discrete lines at arbitrary angles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(5):562–567, 1996.
27. C. Clienti, M. Bilodeau, and S. Beucher. An efficient hardware architecture without line memories for morphological image processing. In *Proceedings of the 10th International Conference on Advanced Concepts for Intelligent Vision Systems*, ACIVS '08, pages 147–156, Berlin, Heidelberg, 2008. Springer-Verlag.

28. M. Van Droogenbroeck and M.J. Buckley. Morphological erosions and openings: fast algorithms based on anchors. *Journal of Mathematical Imaging and Vision*, 22(2):121–142, 2005.
29. L. Garrido, P. Salembier, and D. Garcia. Extensive operators in partition lattices for image sequence analysis. In *Signal Processing*, pages 157–180, 1998.
30. 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 *Advanced Concepts for Intelligent Vision Systems*, pages 230–241. Springer, 2008.
31. J. Brambor. *Algorithmes de la Morphologie Mathématique pour les architectures orietées flux*. PhD thesis, École des Mines de Paris, 2006.
32. Ch. Clienti. Fulguro image processing library. `http://sourceforge.net/projects/fulguro/`, 2011.
33. L. Domanski, P. Vallotton, and D. Wang. Parallel van Herk/Gil-Werman image morphology on GPUs using CUDA. GTC 2009 Conference posters, 2009.
34. P. Dokládal and E. Dokládalová. Computationally efficient, one-pass algorithm for morphological filters. *Journal of Visual Communication and Image Representation*, 22:411–420, 2011.
35. M. H. F. Wilkinson, H. Gao, W. H. Hesselink, J.-E. Jonker, and A. Meijster. Concurrent computation of attribute filters on shared memory parallel machines. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30:1800–1813, 2008.
36. D. Menotti-Gomes, L. Najman, and A. de Albuquerque Araújo. 1D Component tree in linear time and space and its application to gray-level image multithresholding. In *International Symposium on Mathematical Morphology'07*, volume 1, pages 437–448. INPE, 2007.
37. nVidia Corporation. *CUDA Toolkit Reference Manual*. 2701 San Tomas Expressway, Santa Clara, CA95050, August 2010.
38. J. Kong, M. Dimitrov, Y. Yang, J. Liyanage, L. Cao, J. Staples, M. Mantor, and H. Zhou. Accelerating MATLAB Image Processing Toolbox functions on GPUs. In *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 75–85, New York, NY, USA, 2010. ACM.
39. nVidia Corporation. FERMI Tuning Guide. `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/Fermi_Tuning_Guide.pdf`, Aug 2010.
40. AMD Corporation. OpenCL Course: Introduction to OpenCL Programming. `http://developer.amd.com/zones/OpenCLZone/courses/pages/Introduction-OpenCL-Programming-May-2010.aspx`, May 2010.
41. nVidia Corporation. NVIDIA Parallel Nsight. `http://developer.nvidia.com/nvidia-parallel-nsight`, 2011.
42. J. Nickolls and W.J. Dally. The GPU Computing Era. *IEEE Micro*, 30:56–69, March 2010.
43. nVidia Corporation. CUDA SDK Code Samples. `http://developer.nvidia.com/cuda-toolkit-32-downloads`, Aug 2010.
44. nVidia Corporation. NVIDIA Performance Primitives (NPP) Library User Guide. `http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/NPP_Library.pdf`, Feb 2011.
45. nVidia Corporation. CUDA C Programming Guide. `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf`, Sep 2010.
46. T. Randen. Brodatz Textures. `http://www.ux.uis.no/~tranden/brodatz.html`.
47. Pavel Karas and Jan Bartovsky. CUDA-based Linear Openings. `http://sourceforge.net/p/linearopenings/`, Jan 2012.