Efficient graphical-processor-unit parallelization algorithm for computing Eigen values

Sofien Ben Sayadia, Yaroub Elloumi, Mohamed Akil, Mohamed Hedi Bedoui

Efficient Graphical-Processor-Unit Parallelization Algorithm for Computing the Eigen Values

Sofien Ben Sayadia,[a,b,c] Yaroub Elloumi,[a,b,c] Mohamed Akil, [a] Mohamed Hedi Bedoui [b,c]

[a]Gaspard Monge Computer Science Laboratory, Univ Gustave Eiffel, CNRS, ESIEE Paris, F-77454 Marne-la-Vallée, France.

[b]Medical Technology and Image Processing Laboratory, Faculty of medicine, University of Monastir, Tunisia.

[c]ISITCom Hammam-Sousse, University of Sousse, Tunisia.

Abstract. Several leading-edge applications such as pathology detection, biometric identification and face recognition are mainly based on blob and line detection. To address this problem, the Eigen value computing has been commonly employed due to its accuracy and robustness. However, the Eigen value computing requires a raised computational processing, an intensive memory data access and a data overlapping which involve higher execution times.

To overcome these limitations, we propose in this paper a new parallel strategy to implement the Eigen value computing using a GPU. Our contributions are: (1) to optimize instruction scheduling in order to reduce the computation time, (2) to efficiently partition processing into blocks in order to increase the occupancy of streaming multiprocessors, (3) to provide efficient input data splitting on shared memory to take benefit from its lower access time, (4) and to propose new data management of shared memory so as to avoid access memory conflict and reduce memory bank accesses.

Experimental results show that our proposed GPU parallel strategy for Eigen value computing achieves speedups of 27 compared to a multithreaded implementation, of 16 compared to a predefined function in the OpenCV library, and of 8 compared to a predefined function in the Cublas library, which are performed into a quad core multi-CPU platform. Next, our parallel strategy is evaluated through an Eigen value based method for retinal thick vessel segmentation which is essential for detecting ocular pathologies. The Eigen value computing is executed in 0.017 seconds, when using STARE database images. Accordingly, we have achieved real-time thick retinal vessel segmentation where average execution time is about 0.039 seconds.

Keywords: Eigen values; Hessian filter; Parallel algorithms; Graphics Processing unit (GPU); CUDA; Real-time GPU implementation.

Yaroub Elloumi, Email: yaroub.elloumi@esiee.fr

1 Introduction

Line and blob object detection is a fundamental step in common computer recognition domains, such as detecting and modeling friction ridges in fingerprint images for biometric identification [1] and identifying the facial features based on their border edges for face recognition [2]. In medical image diagnosis, several pathology detection methods are based on blood vessel extraction and enhancement, which correspond to linear shapes with flexible widths and orientations [3].

This work was supported by the PHC-UTIQUE 19G1408 Research program.

Moreover, several lesions and anatomical structures are automatically detected based on their curved edge shape like neovascularization and cervical vertebrae [4].

For this purpose, some works have aimed to line and blob object detection, where we particularly distinguish an approach called "Eigen value computing" which has provided higher performance for shape detection. The Eigen values correspond to two features, called respectively $\lambda 1$ and $\lambda 2$, provided for each pixel of the input image, where Frangi et al. [5] defined the relation with corresponding shapes, whether linear or blob. When $\lambda 1 \approx 0$, a $\lambda 2 \gg 0$ (resp. $\lambda 2 \ll 0$) indicates that a pixel belongs to a dark (resp. bright) linear structure in a bright (resp. dark) background. Furthermore, if $\lambda 1 \ll 0$ & $\lambda 2 \ll 0$ (resp. $\lambda 1 \gg 0$ & $\lambda 2 \gg 0$), the pixel belongs to a bright (resp. dark) blob structure in a dark (resp. bright) background, as indicated in Fig. 1.

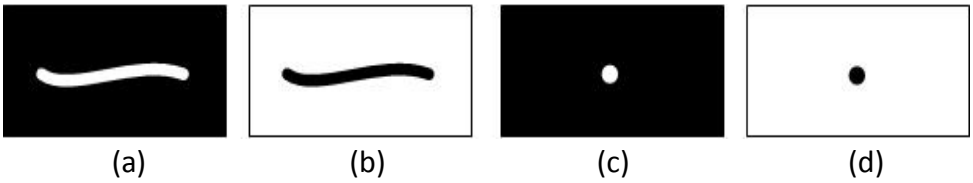


Fig. 1 Relation between Eigen values and linear or blob shapes: (a) Bright linear structure in dark background ($\lambda 1 \approx 0$ & $\lambda 2 \ll 0$), (b) Dark linear structure in bright background ($\lambda 1 \approx 0$ & $\lambda 2 \gg 0$), (c) Bright blob structure in dark background ($\lambda 1 \ll 0$ & $\lambda 2 \ll 0$), (d) Dark blob structure in bright background ($\lambda 1 \gg 0$ & $\lambda 2 \gg 0$)

The Eigen value computing is widely used in several image analysis methods such as the ones dedicated for junction detection [6], circle and spherical shape detection [7] and geometric active contour model for object outlining [8]. Moreover, Eigen value computing allows performing higher accuracies when employed in several applications such as the arterial improvement in magnetic resonance angiography [9], the membrane segmentation in tomography images [10], the shape detection in computed tomography scan imaging [11], the vascular structure segmentation and enhancement [2,9,12–14] and the pathological lesion detection [15,16].

In fact, the computing algorithm of Eigen values requires two main steps, which are respectively modeling shapes in horizontal, vertical and diagonal directions, and providing the matrices λ1 and λ2 through Hessian processing. Both steps are sequentially executed for each pixel and need computational intensive processing. Moreover, those steps simultaneously require several access requests to the same data and lead to overlapped data reading, which involve data access conflict, hence serialized accesses. Consequently, both criteria result in a higher execution time for computing the Eigen values and so for Eigen-values-based methods, as the ones suggested in [3,9,17–19]. Otherwise, the current trend is raising the image input size, such as wide field satellite imagery or ultra-high field images, thus a similar increase in execution time. In contrast, several methods require real-time responses like clutter filtering [20,21] and full waveform inversion [22]. Consequently, the higher execution time of Eigen value processing represents a limitation factor to employ the Eigen-values-based methods.

This article proposes a new parallel strategy to implement the Eigen value computing on Graphics Processing unit (GPU) architecture in order to reduce the execution time. Within this objective, three axes are explored: (i) the scheduling of Eigen value instructions is optimized; (ii) Then, the processing is efficiently partitioned on elementary blocks in order to increase the parallelism level; (iii) In addition, data are saved and managed in shared memory to take advantage of a lower access time. The article is organized as follows. In section 2, we describe Eigen values computing and its execution time behavior. In section 3, we describe our proposed parallel strategy for accelerating the Eigen value computing. The evaluation of our contributions using different types of GPU architecture is done in section 4, followed by the discussion and conclusion in section 5.

2 Eigen value computing

2.1 Processing principles

The Eigen value computing requires two main steps. The first one entitled "shape direction modelling" aims to model shapes in horizontal, vertical and diagonal directions. For this purpose, two approaches were propounded which consisted in applying either the Gaussian filter followed by a second order derivative [3], or applying the Sobel filter followed by a Laplacian filter [23], as represented respectively by top and bottom channels of the first step in Fig. 2. For the first approach, the Gaussian kernel is applied to the input image as indicated in Eq. 1, in order to reduce noise and involve the blurring aspect with respect to the Gaussian deviation.

$$I_{gauss}(i,j) = I(i,j) \otimes G(i,j,\sigma) \quad (1)$$

Where I and I_{gauss} are respectively the input and the convolved images, $\otimes$ represents the convolution operation, $G(i,j,\sigma)$ is the 2D Gaussian kernel and σ is the Gaussian deviation [5].

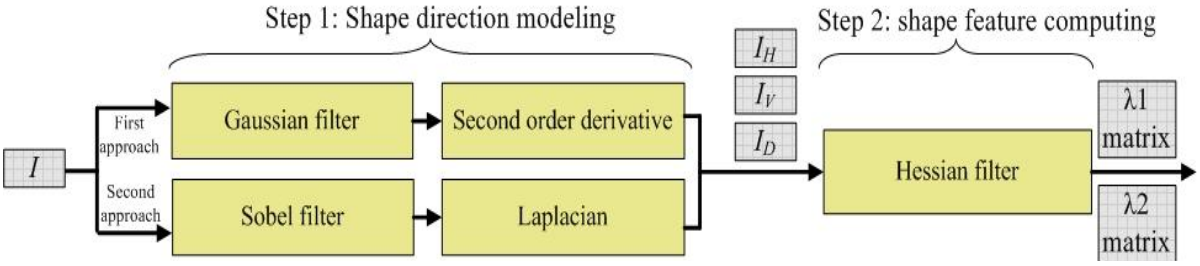


Fig. 2 Eigen value computing flowchart

Thereafter, the second derivative order is applied to the I_{gauss} image on three directions in order to obtain an effective edge representation and highlight regions having rapid intensity change [23]. Several second order derivative methods were proposed to apply the Gradient filter to the I_{gauss} image, as indicated in Eq. 2, Eq. 3 and Eq. 4 [24].

$$I_H(n1,n2) \approx I_{gauss}(n1,n2) \,.\, G_{ver}(n1,n2) \,.\, G_{ver}(n1,n2) \quad (2)$$

$$I_V(n1,n2) \approx I_{gauss}(n1,n2) \,.\, G_{ver}(n1,n2) \,.\, G_{hor}(n1,n2) \quad (3)$$

$$I_D(n1,n2) \approx I_{gauss}(n1,n2) \,.\, G_{hor}(n1,n2) \,.\, G_{hor}(n1,n2) \quad (4)$$

where G_{ver} and G_{hor} are the gradient filter matrices, $(n1,n2)$ is the pixel index in the image and I_H, I_V and I_D images are the results of the second order derivative respectively in horizontal, vertical and diagonal directions, which are illustrated as gray squares after the first step in Fig. 2.

The second step entitled "shape feature computing" aims to compute linear and blob shape features through Hessian processing. For this objective, a Hessian matrix H is provided for each pixel as indicated in Eq. 5 [3–5,19,22].

$$H(i,j) = \begin{bmatrix} I_H(i,j) & I_D(i,j) \\ I_D(i,j) & I_V(i,j) \end{bmatrix} \quad (5)$$

where $i \in [0, N-1], j \in [0, M-1]$, I_H, I_V and I_D are the images provided as described in Eq. 2, Eq. 3 and Eq. 4.

The Hessian matrix of Eigen values is obtained by solving the characteristic equation $|H - \lambda * A| = 0$ with respect to λ, where A is the identity matrix $(2x2)$. The Hessian matrix $(2x2)$ is symmetric and therefore has two float Eigen values, $\lambda 1$ and $\lambda 2$, where $|\lambda 1| < |\lambda 2|$. Indeed, $\lambda 1(i,j)$ and $\lambda 2(i,j)$ are computed in terms of pixels in I_H, I_V and I_D images having the same indices (i,j), which are illustrated respectively in Eq. 6 and Eq. 7.

$$\lambda 1(i,j) = \frac{1}{2}\left(I_V(i,j) + I_H(i,j) - \sqrt{2 * I_H(i,j) \times I_V(i,j) + I_H(i,j)^2 + I_V(i,j)^2 + 4 \times I_D(i,j)^2}\right) \quad (6)$$

$$\lambda 2(i,j) = \frac{1}{2}\left(I_V(i,j) + I_H(i,j) + \sqrt{2 * I_H(i,j) \times I_V(i,j) + I_H(i,j)^2 + I_V(i,j)^2 + 4 \times I_D(i,j)^2}\right) \quad (7)$$

Both Eigen values are computed with respect to each pixel in the input image I, as reported by the two nested loops of Algorithm 1, where $(N \times M)$ is the input image size.

Algorithm 1: Hessian processing

```
Begin
For i:1 To N do
   For j: 1 To  M do
      Compute λ1(i, j) // as indicated in (6)
      Compute λ2(i, j) // as indicated in (7)
    End for
End for
End
```

2.2 Computational time evaluation of Eigen value processing

Eigen-values-based methods are always characterized by a high processing workload, caused by the processing required for each pixel and the number of pixels [4]. In fact, the Gaussian filter, the second order derivative and the Hessian processing require a computation time of $O(M \times N)$, where $(M \times N)$ is the image size. Hence, any Eigen-values-based method requires at least a quadratic computational complexity of $O(M \times N)$. Moreover, some work has indicated that Hessian processing requires significant size of data memory in order to store either input, output images or intermediate images. In addition, Hessian computing needs several data requests and overlapped data reading like convolution computing in the modeling shape direction step, which implies data access latency. Elsewhere, the size growth of input image leads to a similar growth on computational time and on data accesses. Indeed, the image size has risen much faster [15,25–27] than the computer processing power, the memory capacities and the bandwidth for data scheduling [17]. Those criteria lead to a higher execution time for Eigen value computing, which is not suitable for several timing-constrained practical fields, and hence limiting the employment of Eigen-value-based methods.

Thereupon, the predefined function for Eigen processing included in the Open Computer Vision (OpenCV) library [28] is time consuming, which requires 4 seconds for a single image of the High-Resolution Fundus (HRF) database, as proved later in section 4.2.3. Likewise, several

Eigen-value-based methods suffer from higher execution times [17,18,23]. As an example, the cerebral micro bleeds are detected in approximately 17 minutes, when executed on 2.4 GHz Central-Processing-unit (CPU) architecture [9]. In addition, the retinal vessels are segmented for an image with a 2048×1536 size in 31.31 seconds when implementing on 4 GHz Intel i7 CPU [19]. The retinal thick vessels [3] are also extracted in more than 20 seconds when running on 2 GHz Intel i7 CPU using images with a size of 1500×1152.

To tackle this problem, some works have indicated that Eigen value computing is adequate for Single-Instruction-Multiple-Data (SIMD) scheduling and put forward parallel strategies to enhance computational behaviors. In the case of the Harris-Hessian (H-H) algorithm [29], the determinant of the Hessian matrix is applied to avoid false detection. Then, the algorithm was implemented on GPU architecture using the CUDA, where image was equally partitioned into blocks and block threads. In addition, the memory hierarchy is explored to enhance the communication time.

The GPU realization including copying data from the source image into texture and constant memories, achieved a speedup between 10-20x when implemented on the NVIDIA GeForce 9800GTX architecture. Moreover, the Eigen-value-based method propounded in [22] was implemented on a single GPU architecture using the CUDA where data was saved in shared memory to avoid the limitation of the global memory. The implementation achieved a 12-time speedup on the Tesla C2075 GPU card compared with the 8 core CPU implementation optimized by OpenMP. Otherwise, the work described in [4] provided a clutter filtering framework based on Eigen value computing. The framework is modeled on matrix algebra concepts. Then, the instructions are rescheduled and implemented on GPU architecture.

Those works have confirmed that a GPU parallel implementation enhances considerably the execution times of Eigen-value-based methods. Moreover, managing data into memory hierarchy is valuable to reduce the computational performance. However, such works proceed always to alter Eigen value processing [17,22,30]. Then, the parallel implementations are suggested for the whole methods, where the parallel scheduling depends on upstream and downstream Eigen-value processing [4,29,30]. Moreover, the data partitioning is closely related to a specific data size. Indeed, none of these related studies have addressed the problem of overlapping data access. Besides, each proposed parallel strategy is entirely evaluated, where Eigen value computing cannot be evaluated separately. Consequently, no parallel strategy can be a widespread for any Eigen-value-based method.

3 Our proposed parallel strategy of computing Eigen values on GPU Architecture

In this section, we propose a parallel strategy for computing the Eigen values on GPU architecture. As indicated in subsection 2.1, the processing is sequentially composed of Gaussian filter, second order derivative and Hessian processing. The experimentation, detailed in section 4, involves that Hessian processing requires more than 80% of the whole execution time. Furthermore, the Gaussian filter has already been implemented on GPU [10,14,31], and so has the second order derivative [11,32]. Therefore, we focus on proposing a parallel strategy to efficiently implement Hessian processing on GPU architecture. For this purpose, we describe the parallelization principles of our strategy which optimizes and efficiently partitions processing in order to increase the parallelism level, and manage data and memory access in shared memory to decrease the access time.

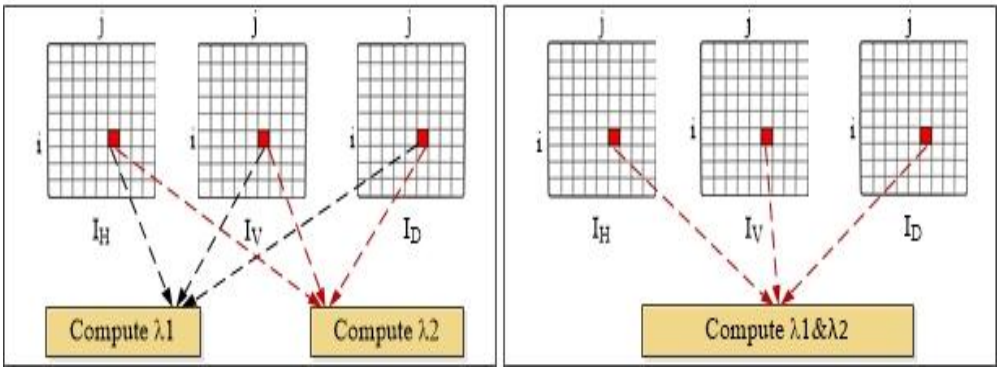
3.1 Optimization of $\boldsymbol{\lambda 1}$ and $\boldsymbol{\lambda 2}$ Eigen value computing

Both $\lambda 1(i,j)$ and $\lambda 2(i,j)$ require reading pixels having the same indexes (i,j) from I_H, I_V and I_D images, where pixels are respectively modeled by red cells in Fig. 3(a). Moreover, the same data are requested several times to compute both Eigen values, as indicated respectively in Eq. 6 and Eq. 7. These data accesses involve an excess of memory usage, which prevents parallel data reading. As a result, scheduling $\lambda 1(i,j)$ and $\lambda 2(i,j)$ in two separate threads leads to a higher latency due to the rise on communication time.

Therefore, we proceed to compute both $\lambda 1(i,j)$ and $\lambda 2(i,j)$ in the same thread in order to reduce reading the $I_H(i,j)$, $I_V(i,j)$ and $I_D(i,j)$ pixels, as depicted in Fig. 3 (b). Therefore, simultaneous data access is avoided which reduces the communication time. In addition, we distinguish that both $\lambda 1$ and $\lambda 2$ Eigen values require a common sub-processing that can be performed in an intermediate parameter $p(i,j)$, as shown in Eq. 8, which presents the important processing and data access required for each Eigen value.

$$p(i,j) = \sqrt{2 \times I_H(i,j) \times I_V(i,j) + I_H(i,j)^2 + I_V(i,j)^2 + 4 \times I_D(i,j)^2} \tag{8}$$

Since both Eigen values are processed in the same thread, we compute the $p(i,j)$ parameter once, and used afterwards to provide $\lambda 1(i,j)$ and $\lambda 2(i,j)$, as given inside the nested loops of Algorithm 2. This optimization leads reduces the computation time from $(26 \times M \times N)$ to $(16 \times M \times N)$, where $(M \times N)$ is the input image size.



(a) (b)

Fig. 3 Computing λ1 and λ2: (a) in different threads; (b) in the same thread

Algorithm 2 : Optimized Hessian processing

1: **Begin**
2: **For** i:1 **To** N **do**
3: **For** j: 1 **To** M **do**
4: Compute $p(i,j)$ // as indicated in (16)
5: $\lambda 1(i,j) \leftarrow \frac{1}{2} \times (I_V(i,j) + I_H(i,j) - p(i,j))$
6: $\lambda 2(i,j) \leftarrow \frac{1}{2} \times (I_V(i,j) + I_H(i,j) + p(i,j))$
7: **End for**
8: **End for**
9: **End**

3.2 Proposed scheduling of Hessian processing on GPU architecture

We notice that each recent GPU architecture has a memory hierarchy where an efficient memory management is a key issue to adequately using the computational power of the GPU. The global memory is the main one in GPUs, which can be accessed from all cores. Depending on the GPU model, it can be expressed in terms of Giga Byte (GB), which allows saving large data. However, it has the slowest accessing speed. As cited in [7,33], the access of data saved in global memory requires a considerable communication time, when comparing to others GPU memories. As highlighted in[3,27], shared memory is about 100x faster than global memory. Hence, saving input images in shared memory considerably reduces the communication time, and so for the whole computational performance of parallel Hessian processing. However, their storage

capacities are limited which is always expressed in terms of kilobyte (KB). Moreover, a shared memory corresponds to a single block, where stored data is available only through block threads.

Thus, the challenge is to split data to be saved in the shared memory, where data processing will be performed in the shared memory-related block, in order to benefit from its higher speed access. In this objective, we simultaneously perform coarse grained and fine grained levels of parallelism. In the coarse-grained level, we partition images between blocks to parallelize their related processing. In the fine-grained level, the Eigen values are computed in parallel between threads of the same block.

The coarse-grained parallelism consists in partitioning the data into minimal number of blocks while the processed data size does not exceed the storage capacity of shared memory. For this purpose, we aim to maximizing the sub-image size while being able to be saved in shared memory. For that, the I_H, I_V and I_D images are divided into sub-images, as illustrated in the left side of Fig. 4. The extracted sub-images, called respectively SI_H, SI_V and SI_D must have the similar size ($Size_{SI} \times Size_{SI}$). In addition, their pixels have the same indexes, with respect to the Eigen values processing indicated in Eq. 6 and Eq. 7. As a result, the $Size_{SI}$ is computed as indicated in Eq. 9.

$$Size_{SI} = \sqrt{Cap_{Shared} / (3 \times Size_{pixel})} \quad (9)$$

Where Cap_{Shared} is the storage capacity of shared memory in terms of bytes, the 3 value corresponds to the number of sub-images and $Size_{pixel}$ is the required bytes to save one sub-image pixel. Each block computes Hessian processing through SI_H, SI_V and SI_D to provide the two sub-images $Sub_{\lambda 1}$ and $Sub_{\lambda 2}$, where the block number (NB_{block}) is defined as indicated in Eq. 10.

$$NB_{block} = \frac{H_I \times W_I}{Size_{SI} \times Size_{SI}} \quad (10)$$

Where $(H_I \times W_I)$ is the size of the whole input image and $Size_{SI}$ is computed as defined in Eq. 9.

In the CUDA software environment, the GPU device executes a grid of thread blocks. Each block has a predefined number of threads that execute the same code. All threads in the same block are organized into a thread warp that has relatively a consistent program behavior.

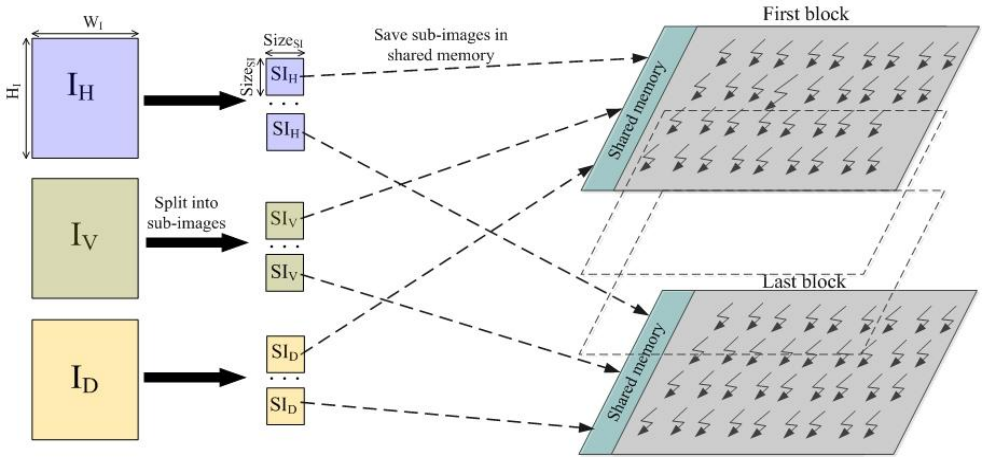


Fig. 4 Coarse-grain parallelism levels in the Hessian GPU implementation

Subsequently, we proceed to scheduling in parallel the Hessian processing allocated for a single block. In fact, the threads in a same block are split into warps which are performed sequentially, while threads of the same warp are executed in parallel. Thus, we choose to define a single warp in each block. Hence, the processing allocated to a block is split into warp threads in order to raise the parallelism level and to synchronize thread processing. Hence, all data of input and output sub-images, which are respectively (SI_H, SI_V and SI_D) and ($Sub_{\lambda 1}$ and $Sub_{\lambda 2}$), are split into threads of a single warp. As modeled by crossed rectangles in Fig. 5, the first partitions in SI_H, SI_V and SI_D, are read by the first thread in the warp. Similarly, the provided Eigen values are saved in related partitions in $Sub_{\lambda 1}$ and $Sub_{\lambda 2}$ images, as modeled by dotted rectangles in

Fig. 5. Therefore, the pixels of the same sub-image are partitioned equally on $pixels_{thread}$, as shown in Eq. 11, where each partition is allocated to a single thread.

$$pixels_{thread} = \frac{Size_{SI} \times Size_{SI}}{maxTh} \quad (11)$$

Where $maxTh$ is the maximal number of thread on a warp and $Size_{SI}$ is the sub-image size defined in Eq. 9.

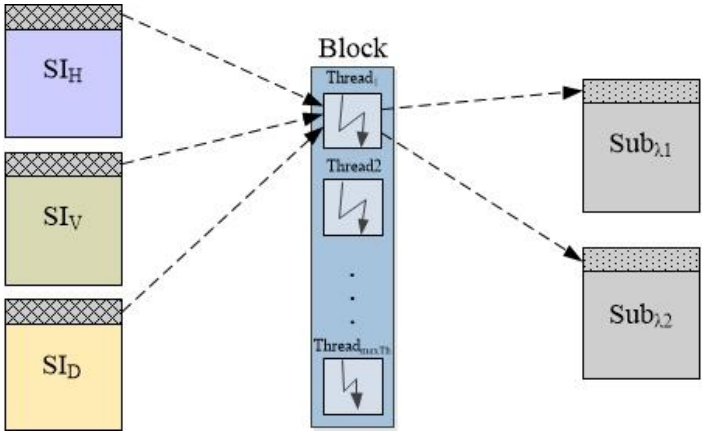


Fig. 5 parallelization of Hessian processing in the same block

Algorithm 3 : Parallel Hessian processing
Inputs: I_H, I_V and I_D matrices **Outputs :** $\lambda 1$ and $\lambda 2$ matrices
1: **Begin** *2:* **For all** k **from** 0 **To** $NB_{block} - 1$ **do in parallel** *3:* **Forall** j **from** 0 **To** $maxTh - 1$ **do in parallel** *4:* **For** i **from**$(\boldsymbol{k} * \times Size_{SI}) + (j * pixels_{thread})$**To**$(\boldsymbol{k} \times Size_{SI}) + \big((j+1) \times pixels_{thread}\big) - 1$ **do** *5:* Compute $p(i,j)$ // as indicated in (16) *6:* $\lambda 1(i,j) \leftarrow \frac{1}{2} \times \ (I_V(i,j) + I_H(i,j) - p(i,j))$ *7:* $\lambda 2(i,j) \leftarrow \frac{1}{2} * \times \ (I_V(i,j) + I_H(i,j) + p(i,j))$ *8:* **End for** *9:* **End for all** *10:* **End for all** *11:* **End**

Consequently, Hessian processing is partitioned on NB_{block} blocks to be executed in parallel, as indicated by the iterative structure in the second instruction of Algorithm 3. The $Size_{SI}$ pixels allocated to each block are split into $maxTh$ threads to be run in parallel, as

modeled in the third instruction of Algorithm 3. Each thread will perform $pixels_{thread}$ pixels sequentially, while the corresponding iterative structure is provided in the fourth instruction.

3.3 Shared memory management

We describe in this section our proposed method to manage data in order to optimize the access time to the shared memory. Data are transferred from global to shared memory of each block. The shared memory is divided into memory modules having an equal data size, called banks. Each single data request consists of an access of its corresponding memory bank.

However, several distinct requests of the same data lead to similar accesses of its memory bank, where accesses are performed sequentially. For the same block, saving SI_H, SI_V and SI_D sub-images on shared memory consists in ordering their pixels successively, as illustrated in Fig. 6 (a). However, each thread processing requires access to a similar pixel set from the three sub-images. As indicated in the processing partitioning, threads in the same block require access to a neighbor pixel which may be stored in the same memory banks, as depicted in Fig. 6 (a), where the first three threads have input data stored in the same bank. Therefore, the access requests from several threads to the same memory bank lead to a bank conflict, and so to serialize accesses.

For this purpose, we reorganize the data of sub-images in shared memory: The pixels of SI_H, SI_V and SI_D images are redistributed such that data needed for every thread are saved with consecutive addresses in the shared memory. Therefore, data required to compute a couple of Eigen values will be saved in the same memory bank, as represented in Fig. 6 (b). As a result, the number of memory bank accesses will be reduced and access conflicts will be avoided, which results in decreasing the communication time and improving the execution time.

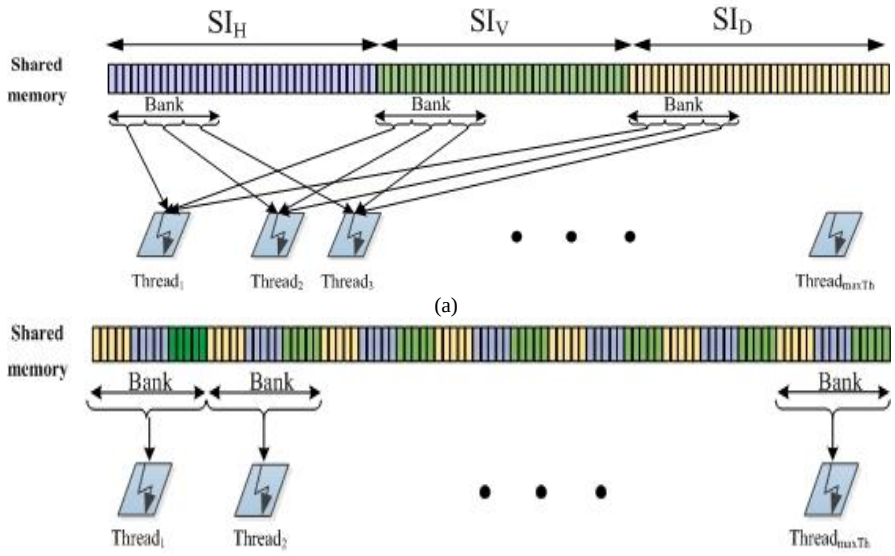


Fig. 6 Coalesced memory accesses illustrating thread reading (a) Respective data elements allocated in separate memory banks and with different addresses; (b) Respective data elements allocated in same memory banks and with consecutive addresses

4 Experimental results and evaluation

4.1 Experiment principles of our implementation method: a case study of thick vessel extraction from image fundus

Our experimentation targets three objectives that are described respectively in the following sub-sections. The first one aims to evaluate the impact of the parallelism strategy on the Hessian processing execution time separately. The second objective consists in studying the execution time improvement of Eigen value computing, as proceeded in [34]. Thereafter, the third one consists at evaluating the impact of our proposed parallelism through a practical field application.

In fact, several ocular pathologies cause gradually the loss of visual field until involving vision lost. Based on the world report on vision [35] edited by the World Health Organization (WHO), a higher number of affected persons are estimated which are expressed in terms of hundreds of millions cases for each pathology. In addition, patient number will increase in

coming years due to aging, unbalance nutrition, etc. Therefore, the early diagnosis is highly recommended in order to cure the ocular diseases or interrupt their evolution. To address this problem, several research activities aimed to propose automated methods for ocular pathology detection from fundus images. Those methods are timing constrained to reduce the ophthalmologist workload and raise the number of diagnosed images. For this purpose, the parallelism evaluations of both Hessian processing and Eigen-values computing are performed using fundus images. The last experimentation consists at evaluation the parallelism strategy impact on commonly main processing of automated methods for ocular pathology detection.

Our parallel strategy is provided independently of image size. Therefore, we choose to carry out the evaluation using varied size to deduce the correlation between parallelism and image size. In this context, we randomly select 10 fundus images from three public data sets having different image sizes which are respectively the STructured Analysis of the Retina (STARE), the Methods to Evaluate Segmentation and Indexing Techniques in the field of Retinal Ophthalmology (MESSIDOR) and the HRF databases indicated in Table 1.

Table 1. Fundus image sizes in terms of fundus image database

	STARE	MESSIDOR	HRF
Image size	700 x 605	2240 x 1488	3540 x 2336

Moreover, the optimized CUDA implementation is proposed to ensure execution time improvement whatever the GPU-architecture is. For this purpose, the execution times of both Hessian and Eigen processing implemented on multi-CPU architecture are compared to execution times for the same processing when implemented in different multi-GPU architectures. the initial code is implemented using C++ in the CPU platform Intel core i7-4790 having a processor frequency 3.67 GHZ (up to 4 GHZ), with 8 MB cache memory and 8 Go RAM and

compiled on a 32-bit mode. Similarly, the parallel codes is implemented in three GPU architectures where architectural parameters are given in Table 2.

Table 2. architectural parameters in terms of GPU-architecture

	GTX 950	GTX 960	GTX 980
SM	6	8	8
Core/SM	128	128	256
Architecture	Maxwell	Maxwell	Maxwell
Number of cores	768	1024	2048
Base clock (MHz)	1127 – 1178	1024 – 1188	1190
Shared memory size	48Kbytes	48Kbytes	96Kbytes
Compilation mode	32-bits	32-bits	32-bits

4.2 Evaluation of the proposed parallel strategy

4.2.1 Execution time evaluation of parallel Hessian processing

Our parallel strategy consists in efficiently managing data into shared memory in order to reduce the communication time. Therefore, we implement Hessian processing respectively in the CPU architecture, in GPU architecture using only global memory and in GPU architecture using shared memory indicated in our parallel strategy described in section 3. Such implemention is performed and evaluated using the three fundus image sets mentioned in Table 1, on both kinds of architectures described in Table 2. The Fig. 7 (a) (respectivelly Fig. 7 (b) and Fig. 7 (c)) illustates the execution time values when implemention is performed on GTX-950 architecture using STARE images (respectively MESSIDOR and HRF images). We deduce that our parallelism strategy offers a higher execution time improvement as regards to the CPU implementation. The achieved speedups, with repsect to CPU implementation and global memory GPU implementation, are modelled in Fig. 7 (d) (respectivelly Fig. 7 (e) and Fig. 7 (f)).

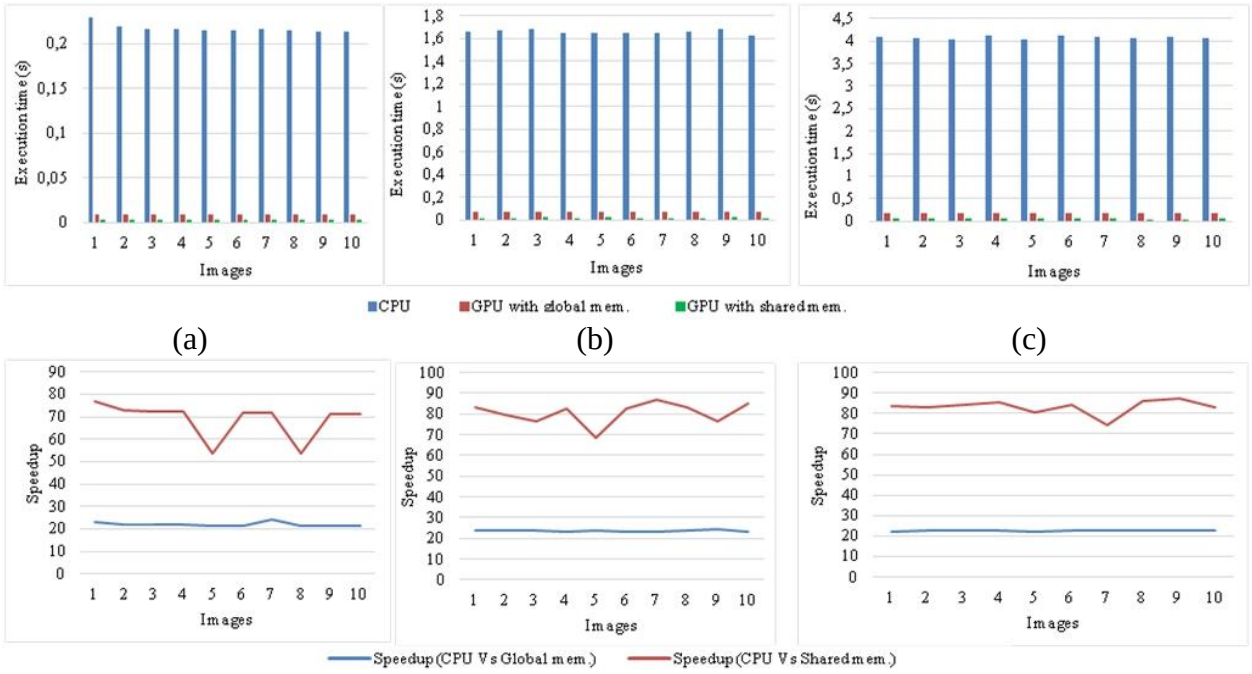


Fig. 7 Hessian filter implementation in GTX-950 architecture: (a) Execution time in terms of STARE images, (b) Execution time of MESSIDOR images, (c) Execution time of HRF images, (d) Speedup in terms of STARE images, (e) Speedup in terms of MESSIDOR images, (g) Speedup in terms of HRF images

The same experimentation process is performed using the GTX-960 architecture and GTX-980 architecture, where the execution times and speedup are depicted respectively in Fig. 8 and Fig. 9.

The speedup averages of shared memory implementation are provided in Table 3. The experimental results show that our parallel strategy ensures enhancing the Hessian processing execution times as regards the CPU architecture and the GPU architecture using global memory. Else, we deduce that the speedup increases in terms of image size.

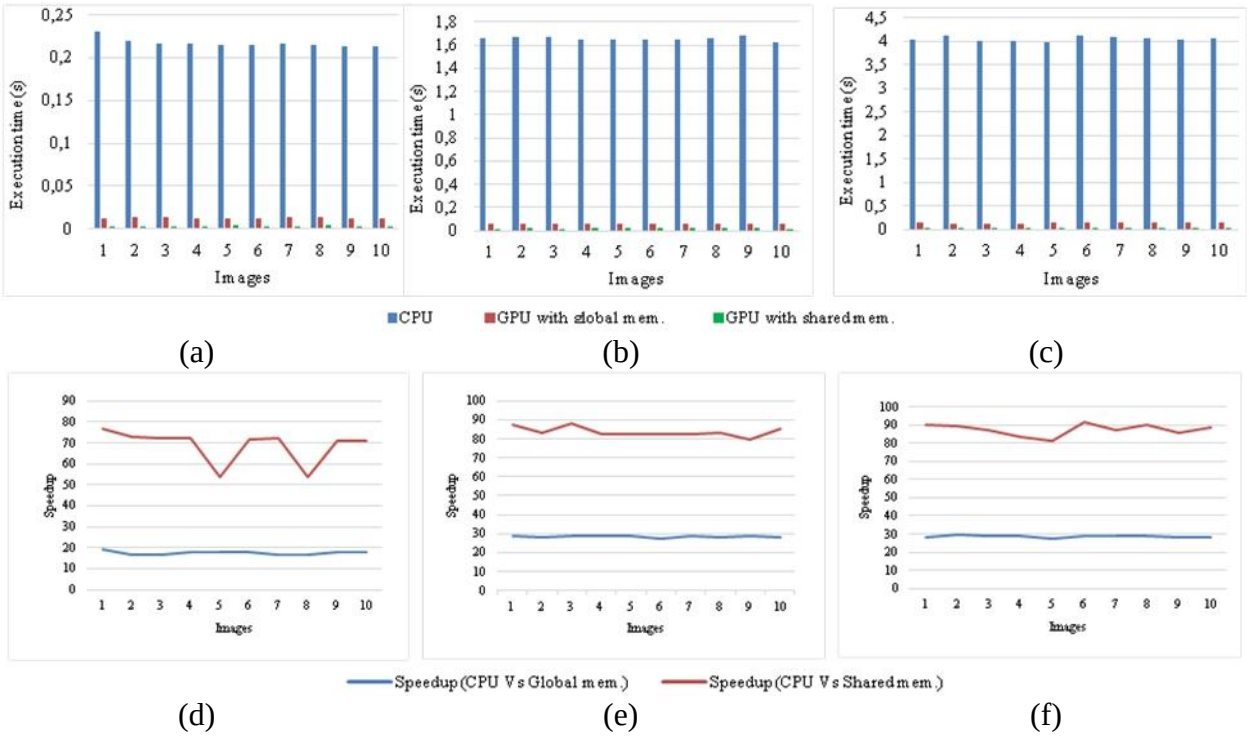


Fig. 8 Hessian filter implementation in GTX-960 architecture: (a) Execution time in terms of STARE images, (b) Execution time of MESSIDOR images, (c) Execution time of HRF images, (d) Speedup in terms of STARE images, (e) Speedup in terms of MESSIDOR images, (g) Speedup in terms of HRF images

Table 3. Speedup of parallel Hessian filter implementation in terms of image sizes and GPU architectures.

	GTX 950		GTX 960		GTX 980	
	Global memory Speedup	Shared memory Speedup	Global memory Speedup	Shared memory Speedup	Global memory Speedup	Shared memory Speedup
STARE	22.84	67.81	17.5	67,81	25.39	79.8
MESSIDOR	23.52	80	28.55	87.15	43.6	129
HRF	23.93	83.3	28.59	87.34	44.37	153.7

Moreover, the speedup rises as a function of the parallelism level offered by the GPU architecture where GTX-980 having 2048 GPU cores allows achieving a higher speedup than GTX-950 and GTX-960 having 768 GPU cores and 1024 GPU cores. Accordingly, we deduce that with respect to the Permanent improvement of GPU architecture (Number of cores, streaming multiprocessor), a similar improvement in speed can be achieved. Furthermore, we deduce that saving and managing data in shared memory allows significantly reducing the execution time in comparison to saving data in global memory where the speedups exceed 3

whatever the image size and the GPU architecture are.

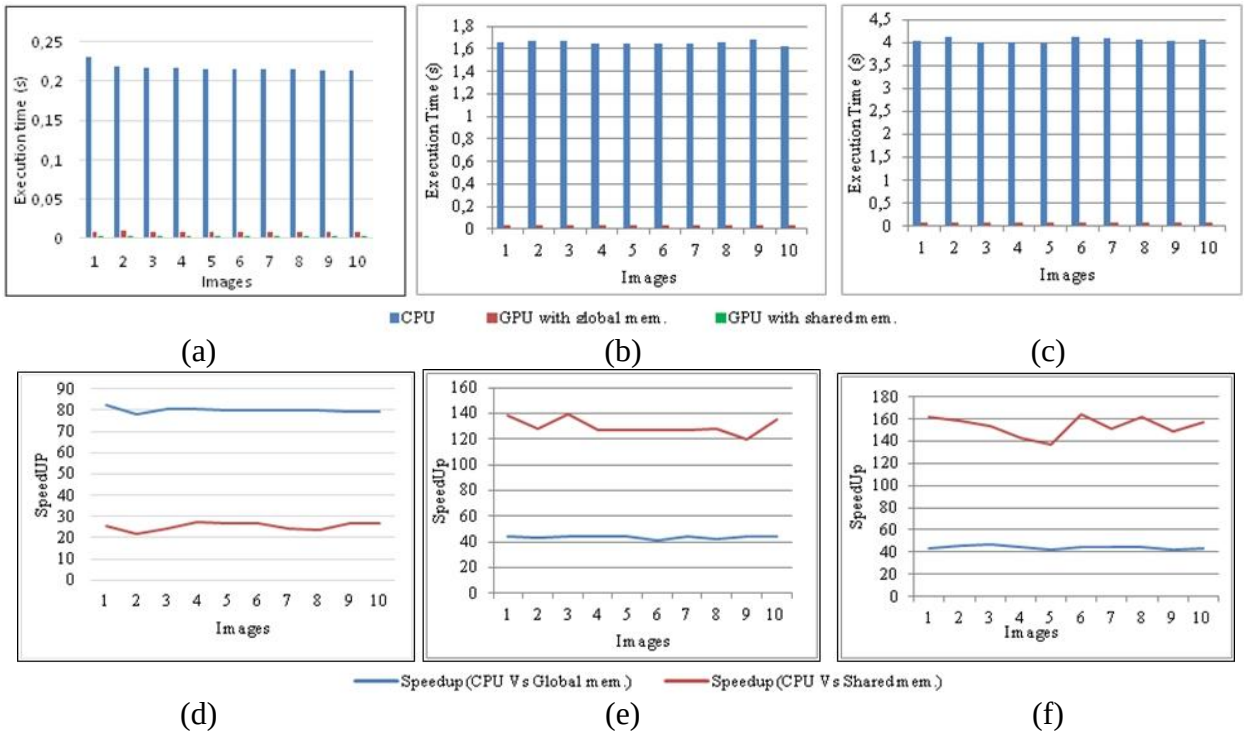


Fig. 9 Hessian filter implementation in GTX-980 architecture: (a) Execution time in terms of STARE images, (b) Execution time of MESSIDOR images, (c) Execution time of HRF images, (d) Speedup in terms of STARE images, (e) Speedup in terms of MESSIDOR images, (g) Speedup in terms of HRF images

4.2.2 Data transfer time evaluation of the parallel Hessian processing

To exploit the GPU architecture, input data must be transferred from CPU to Global memory and output data must similarly brought back to the CPU. The data transfer requires an elapsed time upstream and downstream the calculation execution. Moreover, a large trasferred data leads to a higher transfer time which may reduce the time saved through the parallelism strategy. For this purpose, we proceed to evaluate our parallism strategy with different data size and with multi-GPU architectures having different data throughputs. We randomly selected 10 fundus images from the three fundus image sets mentioned in Table 1, on the three of architectures described in Table 2. Then, the computation time and the data transfer time are compared to each fundus image, as shown in Fig. 10.

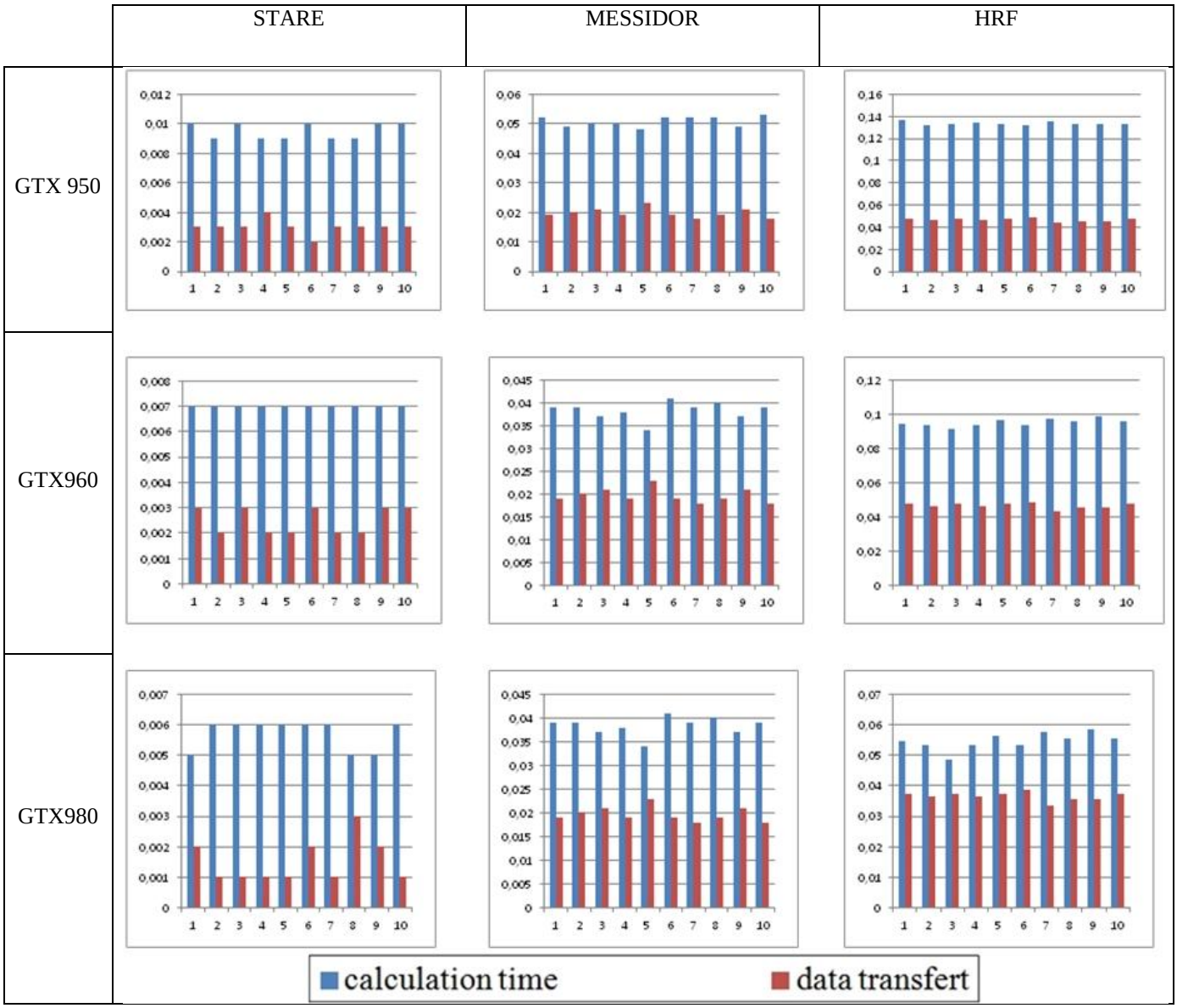


Fig. 10 Hessian filter implementation in GTX-950, GTX-960 and GTX-980 architecture using STARE, MESSIDOR and HRF fundus image

Even with the same architecture, we deduce that data transfert time increases slightly, and remains always negligible against the execution time of the multi-CPU implementation. In addition, the data tranfert requires the same time proportion with respect to the calculation time which similarly depends of the image size. We deduce that, even with the rise of data transfert time with respect to the image size, the speedup remains more important.

4.3 Execution time evaluation of parallel Eigen value computing

In this section, we evaluate the impact of our parallelism strategy on the execution time of the Eigen value computing. Firstly, we evaluate the optimized CUDA implementation against the multi-CPUs implementation. Thereafter, The parallel strategy is evaluated with respect to the predefined function in OpenCV library for Eigen value computing[36] and to the predefined parallel function predefined in Cublas library[47].

4.3.1 Evaluation of the optimized CUDA implementation

Eigen value computing consists in successively applying the Gaussian filter and second order derivative. In fact, the OpenCV library proposes predefined functions that correspond to the parallel implementation of both Gaussian filter and the second order derivative on either the CPU or GPU architecture. Therefore, the Eigen values computing are implemented using predefined OpenCV functions followed by the implementation of our parallel Hessian processing described in section 3. Two types of implementation are performed respectively in CPU and GPU architectures.

Both codes are carried out with the three image sizes and with both GPU architectures, where execution times and speedups are illustrated in Fig. 11. For example, Fig. 11 models the execution times and the speedups when implementing the Eigen value processing on CPU and GPU with shared memory using STARE images on GTX-950 architecture. The gap between sequential execution time and the parallel execution time rises in relation with the image size, which leads to achieve an average speed up, summarized in Table 4, that exceeds 27.1 for HRF image resolution.

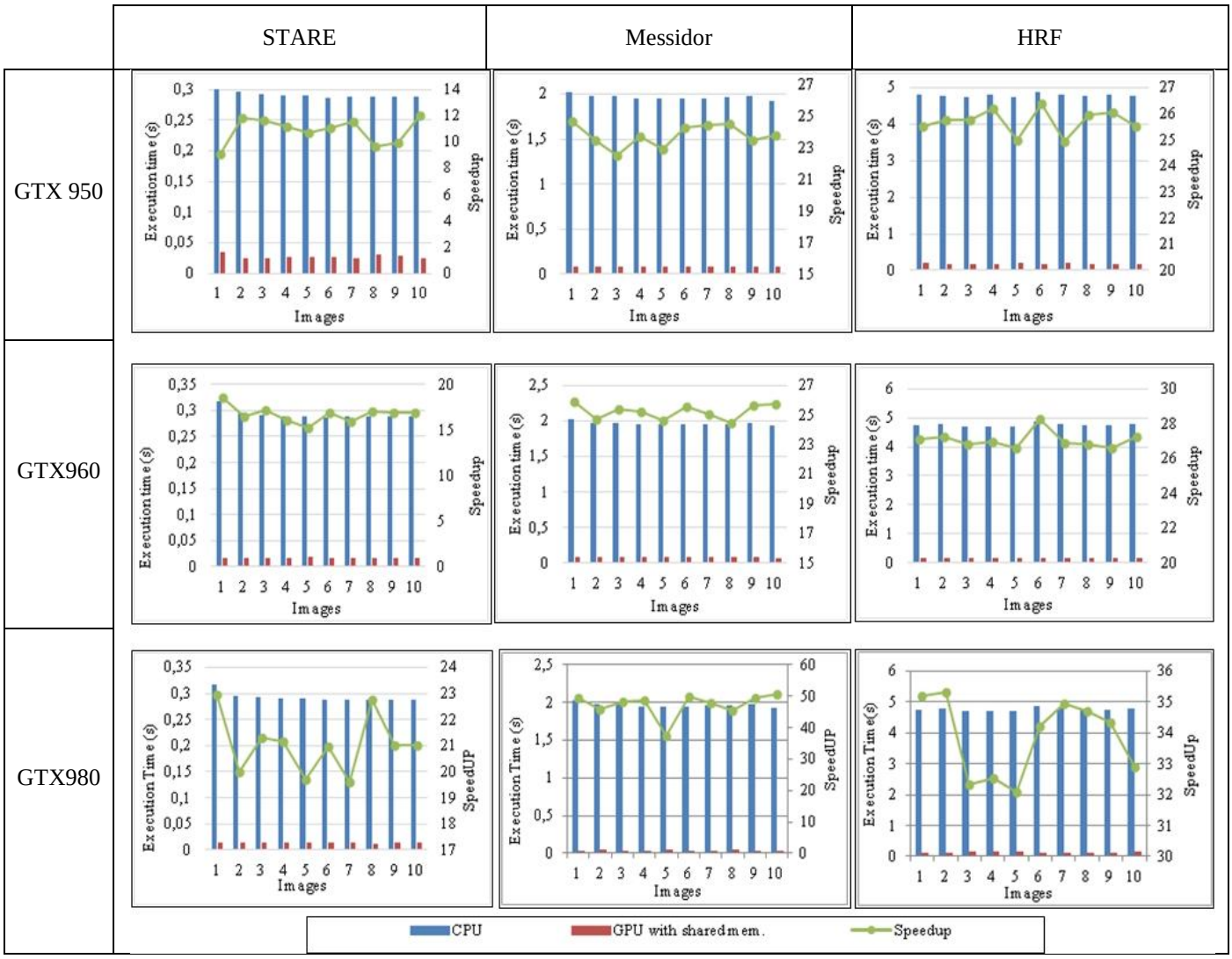


Fig. 11 Execution times and speedup of Eigen values computing in GTX-950, GTX-960 and GTX-980 architecture using STARE, MESSIDOR and HRF fundus image.

For the STARE database images, a real-time Eigen value computing is achieved where average execution times of 0.0272 and 0.017 seconds are registered for GTX-950 and GTX-960, which correspond to 36 and 58 frames per second (fps), respectively. Moreover, we deduce that speedup increases depending on the different number of blocks and threads offered by the GPU architecture.

Table 4. Speedup of Eigen value computing in terms of image sizes and GPU architectures

	GTX 950	GTX 960	GTX 980
STARE	10.73	17.17	21.05
MESSIDOR	24.53	25.16	27.64
HRF	25.7	27.1	33.85

4.3.2 Execution time evaluation against the OpenCV predefined function

In this section, we evaluate the computational performance of our optimized CUDA implementation with respect to the Eigen function defined in the Opencv library [28]. For this purpose, we randomly selected 10 fundus images from the HRF database. Then, optimized CUDA implementation and the OpenCV predefined function are performed to each fundus image, on the GTX-960 architecture, where the execution times are shown in Fig. 12. The Eigen values are provided in an average of 4.15 seconds using the OpenCV predefined function, while they require an average of 0.17 second through the optimized CUDA implementation. Hence, an average speed up, that is defined as the ratio of sum of speed up computed for each image to the number of image, about 16x is achieved. Furthermore, the execution time improvement remains similar for all fundus images.

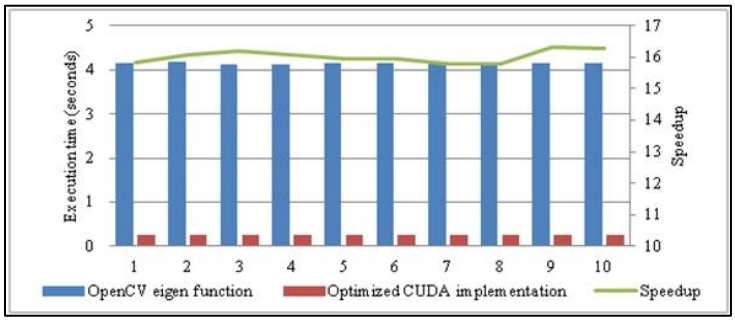


Fig. 12 Execution times and speedup of Eigen values computing: optimized CUDA implementation Vs OpenCV predefined function

4.3.3 Execution time evaluation against the cublas library predefined function

In this section, we evaluate the computational performance of our optimized CUDA

implementation with respect to the Eigen function entitled “cusolverDnDsyevd” defined in cublas library [47]. For this purpose, we randomly selected 10 fundus images from the HRF database. Then, optimized CUDA implementation and the Cublas predefined function are performed to each fundus image, on the GTX-960 architecture, where the execution times are shown in Fig. 13.

In fact, this function computes all Eigen values and eigenvectors. In addition, this function proceeds to extract the Eigen values whose are large than threshold values and them retained as key points features. Therefore, the Eigen values are provided in an average of 1.4 seconds using the “cusolverDnDsyevd” function, while they require an average of 0.17 second through the optimized CUDA implementation. Hence, our optimized CUDA implementation achieves an average speed up about 8x, as modelled for each speedup of fundus images, illustrated in Fig. 13.

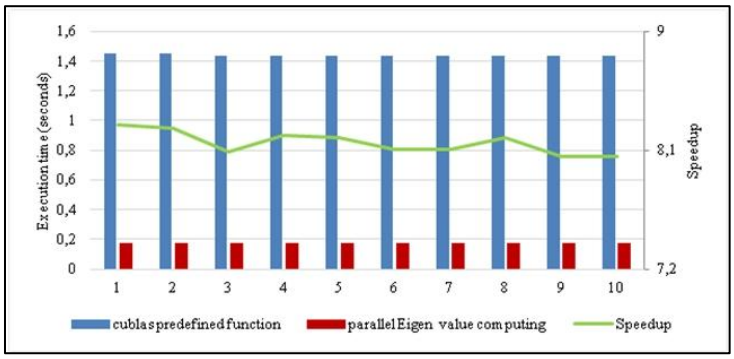
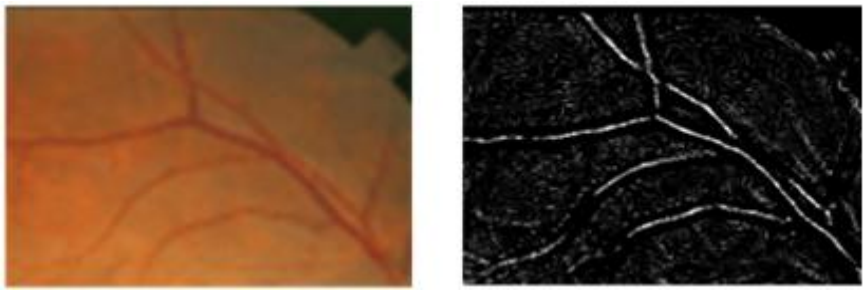


Fig. 13 Execution times and speedup of Eigen values computing: optimized CUDA implementation Vs Cublas predefined function

4.4 Execution time evaluation of thick vessel extraction approach using parallel Eigen value computing

Several methods for ocular pathology detection are based on thick vessel extraction. As an example, the micro-aneurysm and hemorrhages lesions of diabetic retinopathy appear near from thick vessels [37]. In addition, the damage of optic nerve in glaucoma disease is deduced based on

thick vessel distribution into optic disk [38–40]. Moreover, several methods proceed to segment separately thick and thin vessel tree and thereafter merge their results in order to enhance accuracy when segmenting the whole tree of retinal vessel [41,42]. Furthermore, the segmentation of thick vessel tree is considered as main step on automated methods for retinal component detection, where thick vessel convergence to the optic disk [43] and leak of thick vessel on macula region [44] are widely used to locate them. Particularly, the thick vessel tree is segmented in [3] based on Eigen Values, where the result was depicted in Fig. 14. Then, the extracted vessels are explored in order to reflect vessel density and hence locating the optic disk. For this purpose, we choose to evaluate the contribution of the Eigen value parallel implementation when segmenting the thick vessel tree, as proposed in [3].



(a) (b)

Fig. 14 (a) Retinal sub-image; (b) Eigen value matrix of retinal sub-image.

In this context, the whole thick vessel extraction is implemented twice, using the optimized CUDA implementation and the OpenCV predefined function, respectively. The upstream and downstream processing implementation remains unchanged, to ensure a trustworthy evaluation. In addition, the implementation is performed on varying image sizes and GPU architectures, where the execution times and the speedups are presented in Fig. 15, and speedup averages are given in Table 5.

Table 5. Speedup of thick vessel extraction in terms of image sizes and GPU architectures

	GTX 950	GTX 960
STARE	7.3	7.25
MESSIDOR	14.9	16
HRF	16.28	17.5

Even Eigen value computing presents partial processing, the higher speedup achieved by our suggested parallel strategy leads to a significant speedup of the whole thick vessel extraction processing, which exceeds 17x for HRF images. The parallel implementation on GTX-960 the shared memory management leads a real-time thick vessel detection which an average execution time of 0.039 seconds using STARE database images, which corresponds to 25 fps. In addition, the speedup rising of Eigen value computing in terms of image size and parallelism level results a similar speedup enhancement of the thick vessel extraction.

Conclusion

Several state-of-the-art applications are based on Eigen value computing which is distinguished as higher performance approach for line and blob detection. However, it requires computational intensive processing that involves important data access requests and higher overlapped data reading. Hence, Eigen value computing is constrained by a higher execution time.

In this context, a parallel strategy has been put forward and successfully implemented for Eigen value computing. Our contributions consist in optimizing the schedule of Eigen value processing, efficiently partitioning the processing to increase the parallelism level, and managing data in shared memory to decrease the access time. The Parallelism is evaluated through two multi-GPU architectures which provide significant speedups. As consequence, a real-time Eigen value computing is achieved when using STARE database images, where speedups of 27 and 16 are

registered compared to a multi-threaded implementation, and to the Eigen value predefined function in the OpenCV library, both performed into a quad core multi-CPU platform.

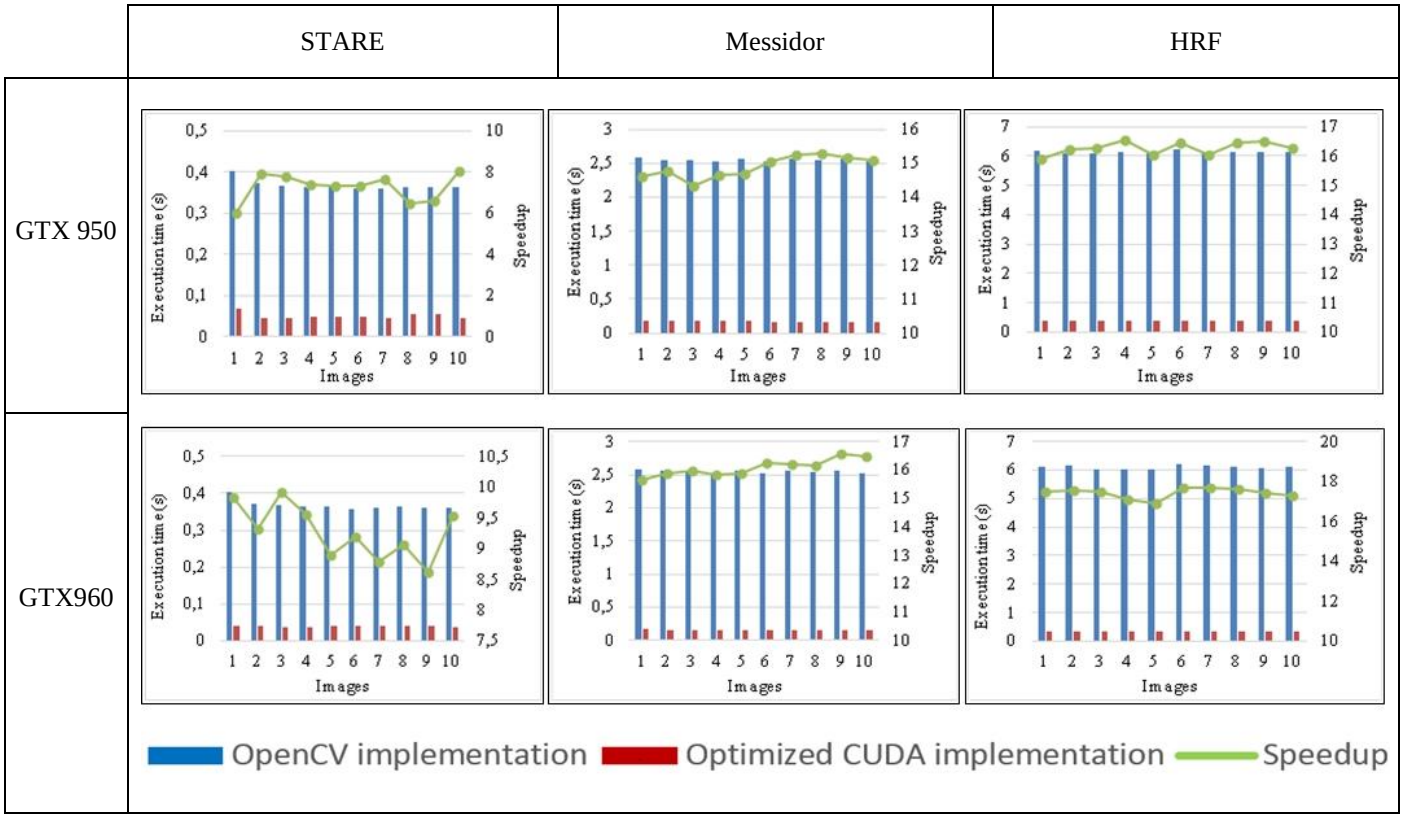


Fig. 15 Execution times and speedup of thick vessel extraction in GTX-950 and GTX-960 architecture using STARE, MESSIDOR and HRF fundus image.

In addition, the experimentation demonstrates that speedups grow in terms of image size, due to the parallelism contribution with respect to host/device data transferring. Similarly, the speedup also rises in terms of parallelism levels offered by GPU architectures, which is permanently rising. Consequently, real time Eigen-value-based methods can be achieved, when using GPU architecture offering higher parallelism level than the used ones. Thereafter, the parallel Eigen value computing has been employed with the thick vessel extraction method where a speedup of 17.5 is reached. Hence, a real-time implementation of thick vessel

segmentation has been achieved through an average execution time of 0.039 seconds when using STARE database images which corresponds to 25 fps.

On one side, our proposed parallel strategy can be directly employed to take advantage of its higher computational performance, respectively for computer vision approaches, like the geometric active contour model [23] and the detection of linear junctions in 2D images [6]. Similarly, several medical applications can take benefit from our parallelism strategy such as Cerebral Microbleeds detection [9] and clusters of micro-calcification detection [44]. Furthermore, the parallel implementation of the thick vessel extraction can be directly employed either for OD or macula detection. In addition, it can be associated to a thin vessel segmentation method in order to provide the whole retinal vessel tree in a lower time. In addition, our parallel strategy can be extended for embedded and mobile devices which are increasingly used on several domains such as medical and health care processing (Von Lühmann et al. 2017; Elloumi, Akil, and Kehtarnavaz 2018) and security systems.

Conflict of interest

The authors declare that they have no conflicts of interest.

References

1. Y. Tang et al., "Contactless Fingerprint Image Enhancement Algorithm Based on Hessian Matrix and STFT," in 2017 2nd International Conference on Multimedia and Image Processing (ICMIP), pp. 156–160 (2017) [doi:10.1109/ICMIP.2017.65].
2. Jian Li et al., "Accelerating Active Shape Model using GPU for facial extraction in video," in 2009 IEEE International Conference on Intelligent Computing and Intelligent Systems **4**, pp. 522–526 (2009) [doi:10.1109/ICICISYS.2009.5357636].
3. I. Soares, M. Castelo-Branco, and A. M. G. Pinheiro, "Optic Disc Localization in Retinal Images Based on Cumulative Sum Fields," IEEE Journal of Biomedical and Health Informatics **20**(2), 574–585 (2016) [doi:10.1109/JBHI.2015.2392712].
4. A. J. Y. Chee, B. Y. S. Yiu, and A. C. H. Yu, "A GPU-Parallelized Eigen-Based Clutter Filter Framework for Ultrasound Color Flow Imaging," IEEE Trans Ultrason Ferroelectr Freq Control **64**(1), 150–163 (2017) [doi:10.1109/TUFFC.2016.2606598].
5. A. F. Frangi et al., "Multiscale vessel enhancement filtering," in Medical Image Computing and Computer-Assisted Intervention — MICCAI'98, W. M. Wells, A. Colchester, and S. Delp, Eds., pp. 130–137, Springer Berlin Heidelberg (1998).

6. R. Su, C. Sun, and T. D. Pham, “Junction detection for linear structures based on Hessian, correlation and shape information,” Pattern Recognition **45**(10), 3695–3706 (2012) [doi:10.1016/j.patcog.2012.04.013].
7. J. Wu et al., “GPU-parallel interpolation using the edge-direction based normal vector method for terrain triangular mesh,” J Real-Time Image Proc **14**(4), 813–822 (2018) [doi:10.1007/s11554-016-0575-1].
8. K. Hanbay et al., “Continuous rotation invariant features for gradient-based texture classification,” Computer Vision and Image Understanding **132**, 87–101 (2015) [doi:10.1016/j.cviu.2014.10.004].
9. T. L. A. van den Heuvel et al., “Automated detection of cerebral microbleeds in patients with Traumatic Brain Injury,” Neuroimage Clin **12**, 241–251 (2016) [doi:10.1016/j.nicl.2016.07.002].
10. K. Preethi and K. S. Vishvaksenan, “Gaussian Filtering Implementation and Performance Analysis on GPU,” in 2018 International Conference on Inventive Research in Computing Applications (ICIRCA), pp. 936–939 (2018) [doi:10.1109/ICIRCA.2018.8597299].
11. H. B. Fredj et al., “Parallel implementation of Sobel filter using CUDA,” in 2017 International Conference on Control, Automation and Diagnosis (ICCAD), pp. 209–212 (2017) [doi:10.1109/CADIAG.2017.8075658].
12. S. A. Mahmoudi et al., “GPU-based segmentation of cervical vertebra in X-Ray images,” in 2010 IEEE International Conference On Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), pp. 1–8 (2010) [doi:10.1109/CLUSTERWKSP.2010.5613102].
13. A. Martinez-Sanchez, I. Garcia, and J.-J. Fernandez, “A differential structure approach to membrane segmentation in electron tomography,” Journal of Structural Biology **175**(3), 372–383 (2011) [doi:10.1016/j.jsb.2011.05.010].
14. T. Yano and Y. Kuroki, “Fast implementation of Gaussian filter by parallel processing of binominal filter,” in 2016 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS), pp. 1–5, IEEE, Phuket (2016) [doi:10.1109/ISPACS.2016.7824738].
15. D. Toslak et al., “Wide-field smartphone fundus video camera based on miniaturized indirect ophthalmoscopy,” Retina **38**(2), 438–441 (2018) [doi:10.1097/IAE.0000000000001888].
16. M. E. Martinez-Perez et al., “Segmentation of blood vessels from red-free and fluorescein retinal images,” Med Image Anal **11**(1), 47–61 (2007) [doi:10.1016/j.media.2006.11.004].
17. R. Wiemker et al., “A Radial Structure Tensor and Its Use for Shape-Encoding Medical Visualization of Tubular and Nodular Structures,” IEEE Transactions on Visualization and Computer Graphics **19**(3), 353–366 (2013) [doi:10.1109/TVCG.2012.136].
18. R. Su et al., “A new method for linear feature and junction enhancement in 2D images based on morphological operation, oriented anisotropic Gaussian function and Hessian information,” Pattern Recognition **47**(10), 3193–3208 (2014) [doi:10.1016/j.patcog.2014.04.024].
19. C. Zhu et al., “Retinal vessel segmentation in colour fundus images using Extreme Learning Machine,” Computerized Medical Imaging and Graphics **55**, 68–77 (2017) [doi:10.1016/j.compmedimag.2016.05.004].
20. A. C. H. Yu and R. S. C. Cobbold, “Single-ensemble-based eigen-processing methods for color flow imaging--Part I. The Hankel-SVD filter,” IEEE Trans Ultrason Ferroelectr Freq Control **55**(3), 559–572 (2008) [doi:10.1109/TUFFC.2008.682].
21. D. E. Kruse and K. W. Ferrara, “A new high resolution color flow system using an eigendecomposition-based adaptive filter for clutter rejection,” IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control **49**(12), 1739–1754 (2002) [doi:10.1109/TUFFC.2002.1159852].
22. J. Jiang and P. Zhu, “Acceleration for 2D time-domain elastic full waveform inversion using a single GPU card,” Journal of Applied Geophysics **152**, 173–187 (2018) [doi:10.1016/j.jappgeo.2018.02.015].
23. K. Hanbay and M. F. Talu, “A novel active contour model for medical images via the Hessian matrix and eigenvalues,” Computers & Mathematics with Applications **75**(9), 3081–3104 (2018) [doi:10.1016/j.camwa.2018.01.033].
24. H. Liao et al., “High performance kernel smoothing library for biomedical imaging,” 2015.
25. T. N. Kim et al., “A Smartphone-Based Tool for Rapid, Portable, and Automated Wide-Field Retinal Imaging,” Transl Vis Sci Technol **7**(5), 21 (2018) [doi:10.1167/tvst.7.5.21].
26. D. Stucht et al., “Highest Resolution In Vivo Human Brain MRI Using Prospective Motion Correction,” PLoS One **10**(7) (2015) [doi:10.1371/journal.pone.0133921].
27. A. von Lühmann et al., “M3BA: A Mobile, Modular, Multimodal Biosignal Acquisition Architecture for Miniaturized EEG-NIRS-Based Hybrid BCI and Monitoring,” IEEE Transactions on Biomedical Engineering **64**(6), 1199–1210 (2017) [doi:10.1109/TBME.2016.2594127].
28. “OpenCV: Eigen support,” <https://docs.opencv.org/master/d0/daf/group__core__eigen.html> (accessed 6 May 2020).
29. H. Xie et al., “GPU-based fast scale invariant interest point detector,” in 2010 IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 2494–2497 (2010) [doi:10.1109/ICASSP.2010.5494898].
30. A. Elliethy and G. Sharma, “Accelerated parametric chamfer alignment using a parallel, pipelined GPU realization,” J Real-Time Image Proc **16**(5), 1661–1680 (2019) [doi:10.1007/s11554-017-0668-5].
31. W.-M. Pang, K.-S. Choi, and J. Qin, “Fast Gabor texture feature extraction with separable filters using GPU,” J Real-Time Image Proc **12**(1), 5–13 (2016) [doi:10.1007/s11554-013-0373-y].
32. M. Almazrooie et al., “Parallel Laplacian filter using CUDA on GP-GPU,” in Proceedings of the 6th International Conference on Information Technology and Multimedia, pp. 60–65 (2014) [doi:10.1109/ICIMU.2014.7066604].

33. M. Soua, R. Kachouri, and M. Akil, "GPU parallel implementation of the new hybrid binarization based on Kmeans method (HBK)," J Real-Time Image Proc **14**(2), 363–377 (2018) [doi:10.1007/s11554-014-0458-2].
34. S. Uzun and D. Akgün, "Accelerated method for the optimization of quadratic image filter," Journal of Electronic Imaging **28**(03), 1 (2019) [doi:10.1117/1.JEI.28.3.033036].
35. "World report on vision," <https://www.who.int/publications-detail/world-report-on-vision> (accessed 6 May 2020).
36. Y. Zhou, F. He, and Y. Qiu, "Accelerating image convolution filtering algorithms on integrated CPU–GPU architectures," Journal of Electronic Imaging **27**(03), 1 (2018) [doi:10.1117/1.JEI.27.3.033002].
37. N. Salamat, M. M. S. Missen, and A. Rashid, "Diabetic retinopathy techniques in retinal images: A review," Artificial Intelligence in Medicine **97**, 168–188 (2019) [doi:10.1016/j.artmed.2018.10.009].
38. J. A. de la Fuente-Arriaga, E. M. Felipe-Riverón, and E. Garduño-Calderón, "Application of vascular bundle displacement in the optic disc for glaucoma detection using fundus images," Computers in Biology and Medicine **47**, 27–35 (2014) [doi:10.1016/j.compbiomed.2014.01.005].
39. N. M. Radcliffe et al., "Retinal Blood Vessel Positional Shifts and Glaucoma Progression," Ophthalmology **121**(4), 842–848 (2014) [doi:10.1016/j.ophtha.2013.11.002].
40. A. Issac, M. Partha Sarathi, and M. K. Dutta, "An adaptive threshold based image processing technique for improved glaucoma detection and classification," Comput Methods Programs Biomed **122**(2), 229–244 (2015) [doi:10.1016/j.cmpb.2015.08.002].
41. S. Roychowdhury, D. D. Koozekanani, and K. K. Parhi, "Blood Vessel Segmentation of Fundus Images by Major Vessel Extraction and Subimage Classification," IEEE Journal of Biomedical and Health Informatics **19**(3), 1118–1128 (2015) [doi:10.1109/JBHI.2014.2335617].
42. C. Tian et al., "Multi-path convolutional neural network in fundus segmentation of blood vessels," Biocybernetics and Biomedical Engineering **40**(2), 583–595 (2020) [doi:10.1016/j.bbe.2020.01.011].
43. S. B. Sayadia et al., "Computational efficiency of optic disk detection on fundus image: a survey," in Real-Time Image and Video Processing 2018 **10670**, p. 106700G, International Society for Optics and Photonics (2018) [doi:10.1117/12.2304941].
44. S. Samanta, S. K. Saha, and B. Chanda, "A Simple and Fast Algorithm to Detect the Fovea Region in Fundus Retinal Image," in 2011 Second International Conference on Emerging Applications of Information Technology, pp. 206–209 (2011) [doi:10.1109/EAIT.2011.22].
45. M. Muthuvel, B. Thangaraju, and G. Chinnasamy, "Microcalcification cluster detection using multiscale products based Hessian matrix via the Tsallis thresholding scheme," Pattern Recognition Letters **94**, 127–133 (2017) [doi:10.1016/j.patrec.2017.05.002].
46. Y. Elloumi, M. Akil, and N. Kehtarnavaz, "A mobile computer aided system for optic nerve head detection," Computer Methods and Programs in Biomedicine **162**, 139–148 (2018) [doi:10.1016/j.cmpb.2018.05.004].
47. https://docs.nvidia.com/cuda/cublas/index.html.