Efficient parallelization on GPU of an image smoothing method based on a variational model

Carlos A.S.J. Gulo, Henrique F. de Arruda, Alex F. de Araujo, Antonio C. Sementille, João Manuel R.S. Tavares

Received: date / Revised: date

Medical imaging is fundamental for improvements in diagnostic accuracy. However, noise frequently corrupts the images acquired, and this can lead to erroneous diagnoses. Fortunately, image pre-processing algorithms can enhance corrupted images, particularly in noise smoothing and removal. In the medical field, time is always a very critical factor, and so there is a need for implementations which are fast and, if possible, in real-time. This study presents and discusses an implementation of a highly efficient algorithm for image noise smoothing based on General Purpose Computing on Graphics Processing Units (GPGPU) techniques. The use of these techniques facilitate the quick and efficient smoothing of images corrupted by noise, even when performed on large-dimensional data sets. This is particularly relevant since GPU cards are becoming more affordable, powerful, and common in medical environments.

Keywords: GPGPU, CUDA, Image Processing, Multiplicative noise.

1. Introduction

Computational image processing is a field that has seen tremendous advances in recent years. These advances are the result of huge demands coming from areas such as medicine [1], agriculture [2], security [3], traffic and satellite data analysis [4] and industry [5]. These fields require image-processing tasks such as noise and artefact removal and smoothing [6], geometrical correction [7], contrast enhancement [8], image restoration, [9] and illumination correction [10]. Briefly, the use of image processing techniques, particularly of image pre-processing, is mainly intended to enhance the data presented in the original images so that the processed data can be analyzed more easily using higher-level techniques of computational image analysis, such as image segmentation [11] or image registration [12]. However, many of the original images that need to be enhanced have large dimensions [13], and need to be processed in real- or near real-time [14]. This is the case, for example, in the fields of robotic navigation or assisted surgery, or even when the input data are long sequences of 2D or 3D images, such as in ultrasound imaging [15]. Additionally, to obtain more robust and efficient results, the computational complexity of the more recent methods has considerably increased, leading to slower runtimes. Therefore, the use of parallel computing strategies has

attracted attention, and this has led to higher speeds, particularly in time-constrained applications for medical diagnosis [16].

Frequently, noise corrupts images, and this may be due to the image acquisition procedure involved, or to artefacts generated by data transmission or other processes [17]. The image smoothing method proposed by Jin and Yang [18] has obtained very promising results, particularly when applied to medical images. However, a long computational time when performing several iterations on the input image is required by this method, especially when applied to large-scale images; as a result, its use has become less attractive for some potential applications. Additionally, there is a frequent and increasing demand for fast responses from computational methods in high-resolution image processing, and real-time is preferable due to the severe time constraints that characterize medical imaging.

Therefore, we have developed a parallel implementation of the smoothing method proposed by Jin and Yang [18] using General Purpose Computing on Graphics Processing Units (GPGPU) [19] and Compute Unified Device Architecture (CUDA) [20] in order to speeding up its runtime. We have assessed the performance of this strategy by comparing the runtime for parallel implementation (GPU) against that of sequential implementation (CPU).

The method adopted for image smoothing selects each pixel from the input image, and thus requires a large number of calculations; this leads to the long runtime mentioned. Briefly, the method involves the use of an $(m \times n)$ matrix, which is processed for T iterations. Thus, the computational complexity of the processing of the input image is equal to $O(m \times n \times T)$, where m and n are the number of rows and columns of the input image, respectively.

In our parallel implementation, the input image data are stored in the GPU's memory, where the highest number of accesses occur, in order to eliminate as many data accesses as possible within the main memory system [21, 19]. Hence, input image processing is executed in parallel in the GPU. The experimental findings confirmed that the combination of the CUDA architecture and GPGPU techniques were very promising in terms of speeding up the runtime for image processing and computational analyses. These approaches led to high processing performance at a low cost, mainly when compared to parallel implementations in multicomputers.

As far as the authors known, this was the first time that the smoothing method adopted was parallelized using CUDA architecture and GPGPU techniques. The findings are of great interest for image processing and analysis, mainly within the medical community. In this case, medical images of ever higher resolution need to be smoothed as fast as possible in real clinical scenarios. Nowadays, computers with GPUs are commonly available in medical environments and, although these computers are not always the most up-to-date models, their computational power is still sufficient to achieve efficient fast results.

This paper is organized as follows: section 2 introduces the image smoothing method proposed by Jin and Yang [18]; section 3 describes the metrics of Structural SIMilarity (SSIM), Peak Signal-to-Noise Ratio (PSNR) and Normalized Cross-Correlation (NCC), all of which are used to assess the quality of the

smoothing results; section 4.3 presents the parallel implementation of the image smoothing method; the computational runtimes demanded by the CPU- and GPU-based implementations are discussed in section 5; and finally, section 6, presents the concluding remarks.

2. Image Smoothing Method

Images frequently have multiplicative noise, which comes from multiplying an original image *I* by a noisy image I_n [22]. This type of noise is present, for example, in microscopy, ultrasound and infrared imaging [23]. Multiplicative noise is usually more difficult to remove than additive noise [24]. Therefore, to overcome this problem, variational models for multiplicative noise removal have been integrated into smoothing methods specially developed for such images [24, 25]. In 2011, Jin and Yang [18] proposed a very promising method for removing and smoothing multiplicative noise from corrupted images using the variational model for additive noise removal proposed by Rudin et al. [26], as shown here:

$$min_u\{J(u) + \lambda \int_\Omega (f-u)^2\}, \quad (1)$$

where Ω is a closed area belonging to R^2, f is the image corrupted by additive noise, u is the image in the current smoothing iteration, $J(u)$ is a regulator term, and λ is a weight parameter. Jin and Yang designed the method specifically to remove multiplicative noise from ultrasound images, and they concluded that the function proposed by Krissian et al. [27] could be adopted to solve the variational model of Eq. 1, using:

$$E(u) = \int_\Omega \frac{(f-u)^2}{u}, \quad (2)$$

where u is the original image, $f = u + \sqrt{ug}$ is now the input image corrupted by multiplicative noise, and g is a Gaussian variable with a non-zero mean. Thus, the variational model adopted by Jin and Yang [18] is:

$$min_u \left\{J(u) + \lambda \int_\Omega \left[\frac{(f-u)^2}{u}\right]\right\}, \quad (3)$$

where $\lambda > 0$ is a weight parameter. As such, the model given by Eq. 3 deals with the problem of multiplicative noise removal by adopting:

$$\partial_t u = div\left(\frac{\nabla u}{|\nabla u|}\right) + \lambda\left(\frac{f^2}{u^2} - 1\right), \quad (4)$$

where ∇ and *div* are the gradient and divergent operators respectively. In order to discretize the continuous part of Eq. 4, Rudin et al. used the finite difference scheme [26], adopting $h=1$ for the step size and Δt for the time interval, which leads to:

$$
\begin{aligned}
A &= D_x^+(u_{i,j}) = u_{i+1,j} - u_{i,j}, \\
B &= D_x^-(u_{i,j}) = u_{i,j} - u_{i-1,j}, \\
C &= D_y^+(u_{i,j}) = u_{i,j+1} - u_{i,j}, \\
D &= D_y^-(u_{i,j}) = u_{i,j} - u_{i,j-1}, \\
|D_x(u_{i,j})| &= \sqrt{A^2 + (m[C,D])^2 + \delta}, \\
|D_y(u_{i,j})| &= \sqrt{C^2 + (m[A,B])^2 + \delta},
\end{aligned}
\tag{5}
$$

where the parameter $\delta > 0$ is a constant defined close to zero, and term *m* is defined as:

$$
m[a,b] = \frac{sign(a)+sign(b)}{2} min(|a|,|b|), \tag{6}
$$

where $min(|a|,|b|)$ is a function that returns the smallest absolute value between *a* and *b*, and $sign(\mathrm{a})$ is a function that determines the sign of *a*, returning 1 if *a* is positive, -1 if it is negative, and 0 if a is equal to 0. Assuming the iterations of the model *k = 1,2, ...* , Eq. 4 can be rewritten as:

$$
\begin{aligned}
u_{i,j}^{k+1} = \; & \delta t \left[\frac{-D_x^+(u_{i-1,j}^k) + D_x^+(u_{i,j}^k)}{-|D_x(u_{i-1,j}^k)| + |D_x(u_{i,j}^k)|} + \right. \\
& \left. \frac{-D_y^+(u_{i,j-1}^k) + D_y^+(u_{i,j}^k)}{-|D_y(u_{i,j-1}^k)| - |D_y(u_{i,j-1}^k)|} \right] , \\
& + \lambda^k \left(\frac{f^2}{(u_{i,j}^k)^2} - 1 \right) + u_{i,j}^k
\end{aligned}
\tag{7}
$$

where *f* is the input image affected by multiplicative noise. In this method, the λ parameter is automatically calculated for each new iteration as:

$$
\lambda^k = \frac{1}{\sigma^2} \left(\Sigma_{i,j} \left[\left(D_x^- \left(\frac{D_x^+(u_{i,j}^k)}{|D_x(u_{i,j}^k)|} \right) + D_y^- \left(\frac{D_y^+(u_{i,j}^k)}{|D_y(u_{i,j}^k)|} \right) \right) \frac{(u_{i,j}^k - f) u_{i,j}^k}{u_{i,j}^k + f} \right] \right), \tag{8}
$$

where σ^2 is the variance of the image at iteration *k*. As an example, Fig. 1 shows the result of the smoothing method when applied to ultrasound images.

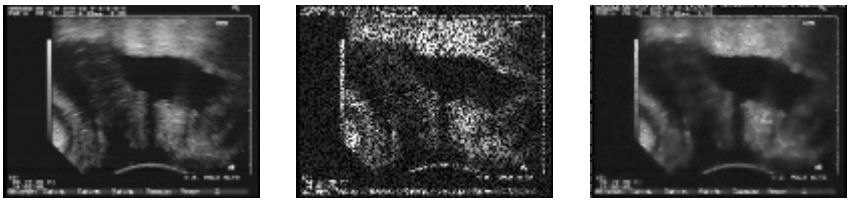

Fig. 1 - Original, noisy and smoothed (128x128 pixels) images, respectively.

3. Assessment Metrics

The comparison between two images is a natural task for the human visual system, but it is not so natural for computer systems. Therefore, various authors have proposed different solutions which assess the similarities between two images, and in particular, evaluate the performance of image pre-processing

methods [28, 29, 30, 31, 32]. Basically, there are two classes of solutions: one is based on intensity error, and the other on structural information.

3.1. Based on Intensity Error

For image smoothing, the comparative solutions or similarity indices use intensity error in order to estimate the error between the enhanced image, i.e. the smoothed image, and the original image before noise corruption. The main disadvantage of these similarity indices is the possibility of failure where there are displacements between the images under comparison. Moreover, these indices compare the intensity variation of each pixel of the input images, which can lead to similar results for images with different types of geometrical distortions [29]. Nevertheless, indices based on intensity error are frequently used to compare the performance of image enhancement [33, 34, 35] and smoothing [13, 17] methods, due to their simplicity.

In particular, the Peak Signal-to-Noise Ratio (PSNR) index has been widely used to assess the performance of image restoration and smoothing methods. This index determines the ratio between the highest possible strength of a signal, which in the case of images is the highest intensity value, and its strength as affected by noise [17, 15]. For simplicity, the PSNR is represented according to a logarithmic scale (base 10), since some signals can have very high values.

The PSNR can be calculated from the Mean Squared Error (MSE), which is computed as:

$$MSE = \frac{1}{m \times n} \sum_{i=0}^{m-1} \sum_{j=1}^{n-1} [I(i,j) - I_r(i,j)]^2, \quad (9)$$

where m and n are the dimensions of the images I and I_r to be compared, as follows:

$$PSNR = 10 log_{10} \left(\frac{MAX_I^2}{MSE} \right), \quad (10)$$

where MAX_I is the maximum intensity value that a pixel can assume, which is equal to 255 for 8-bit grayscale images. Thus, the higher the PSNR value is, the more efficient the performance of the pre-processing algorithm is. The two images are considered identical, when the MSE value is 0 (zero), and the PSNR value is undefined.

Normalized cross-correlation (NCC) is another metric based on pixel intensity. It is widely used in image registration [30, 36] to compare the degree of similarity between two input images. NCC is as follows:

$$NCC = \frac{\sum_{i=1}^{m \times n} x_i y_i}{\sqrt{\sum_{i=1}^{m \times n} x_i^2 \sum_{i=1}^{m \times n} y_i^2}}, \quad (11)$$

where x_i and y_i denote the intensity values of each pixel of the $(m \times n)$ images under comparison, leading to values in the interval $[0, 1]$, where 1 (one) indicates a best match [37].

3.2. Based on Structural Information

In this class of quality metrics, the goal is to find changes in the structural information of the images under comparison. The analysis of the structural information represented in the input images assumes that the human vision system is adapted to extract, i.e. segment, structural information from what is seen, and to search for changes in the structures detected. In other words, any possible differences, such as those due to artefacts generated by noise processes [37], are quantified.

The Structural Similarity Index (SSIM) is the main index in this category which analyzes the performance of computational image processing methods [38, 13]. Wang et al. [29] proposed this similarity index in an attempt to prevent images with very different visual qualities ending up with high similarity values, as can happen when the similarity indices are based on intensity error. The index measures the change in three components of each image under comparison: luminance, contrast and image structure. The former is defined as average pixel intensity. The contrast component is modeled using the standard deviation of the intensity, while image structure comes from the normalized image using the standard deviation of images under comparison. The SSIM is as follows:

$$SSIM(x,y) = l(x,y)^{\alpha} \cdot c(x,y)^{\beta} \cdot s(x,y)^{\gamma}, \tag{12}$$

where l refers to luminance, c to contrast and s to structure, and $\alpha > 0$, $\beta > 0$ and $\gamma > 0$ are weights. These three components are relatively independent, and therefore, modifying one of them does not affect the others. More details of the calculation of these components, as well as a detailed analysis of them, are presented in [29].

The SSIM is an index which applies to each pixel of the input image and, for convenience, the mean SSIM is usually adopted. The Mean Structure Similarity Index (MSSIM) is the average of all the SSIM values obtained. For identical images, this index is equal to 1 (one). As the images become different, the index becomes lower until it is equal to -1 when the images are exactly opposite, i.e. one is the negative of the other.

4. Parallelization of the Smoothing Method

Studies have shown that GPU-based parallel methods have focused on massively parallel programming [40], and most common image processing methods can operate with parallelization strategies based on the data decomposition technique. This section describes the steps involved in the GPU-based parallel implementation, which was developed in order to optimize the runtime performance of the adopted smoothing method.

The smoothing method adopted in this study, as described in Section 2, made use of four fundamental equations to find a solution for the multiplicative noise smoothing process given by Eq. 4. The method starts by solving the finite difference scheme adopted in Eq. 5. Then, Eq. 7 obtains the final value for each pixel according to the on-going iteration, and Eq. 8 finds the associated weight parameter. Thus, Eqs. 4,

5, 7, and 8, define a sequence of steps for the parallelization of the smoothing method. The implementation procedure was based on the NVIDIA programming best practices guide [19].

The CUDA architecture was developed with the objective of using data parallelism, by establishing a new model named Single-Instruction, Multiple-Thread (SIMT). In this model, data are represented as a stream, which is structured as an array; and when running one or more instructions using this array, the instructions are defined as a kernel [16, 19]. A kernel performs operations in parallel along the entire stream, using it as both input and output [39, 19].

In the SIMT model, the calling of multiple kernels follows a hierarchy of thread groups. This feature divides each kernel into independent blocks, and as a result, the efficient threading support in the GPUs ensures transparency, portability and scalability, besides allowing a CUDA program to be executed in any number of processor cores. Threads are used for fine-grained parallelism; groups of threads, defined as "blocks", are used for coarse-grained parallelism; groups of blocks are placed in a grid which represents a kernel call. As illustrated in Fig. 2, this hierarchy allows each thread within a block and each block in a grid to have a unique identifying index [39].

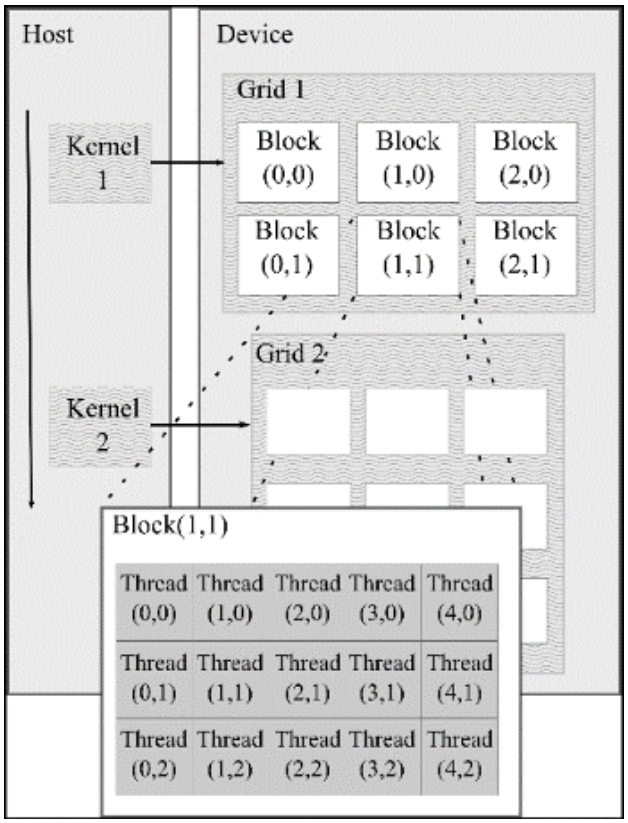

Fig. 2 - Representation of the Single-Instruction, Multiple-Thread Model (adapted from [30]).

4.1. Setting the Occupancy Level

The setting of each kernel must be adjusted to use the correct number of blocks and threads in order to optimize the occupancy of the CUDA cores (code lines in Fig. 3); i.e. if the number of blocks and threads is not sufficient, some cores will not be able to execute the code, wasting some of the processing power.

In our implementation, we used 256 threads per block in all the kernels. The number of blocks B is given by:

$$B = \frac{T_{px}+T_{Tb}-1}{T_{Tb}}, \quad (13)$$

where T_{px} is the total number of pixels of the input image, and T_{Tb} is the number of threads per block. In this case, we defined one two-dimensional variable (*numBlocks*), which has the image height (*HImage*) as the first dimension (T_{px}) and the image width as the second dimension (T_{px}). These calculations determine the settings used to perform one thread per pixel. When there are excess threads, a stopping criterion discards them.

```
dim3 threadsPerBlock(16, 16)                          1
dim3 numBlocks((HImage + 15) / threadsPerBlock.y,     2
  (WImage + 15) / threadsPerBlock.x)                  3
```

Fig. 3 - Definitions for the settings of each kernel used in the experiments.

Applications developed for massively parallel architectures achieve greater performance when the graphics card resources are used efficiently. The occupancy level of the GPU measures the proportion of active processors in the graphics card during a kernel execution. This calculation takes into account the following specification query attributes acquired from the CUDA device: the maximum number of threads per block, the number of blocks per multiprocessor, the number of registers per multiprocessor, and the shared memory per multiprocessor. Increasing the number of concurrent threads is a good strategy for the purpose of making full use of the GPU, and the limit of threads is defined by the architecture. However, a high level of GPU occupancy does not guarantee an additional performance gain [19] because there is a problem of memory latency, and a high level of occupancy may reduce the overall performance [40].

4.2. Optimizing the Memory Hierarchy in CUDA

As shown in Fig. 4, each multiprocessor can use four types of memory: a set of registers for each Stream Multiprocessor (SM), a shared memory between the SMs, a constant cache shared between the SMs, and a texture cache which optimizes the bandwidth of the texture memory. Registers have the largest bandwidth and, like other kinds of memory, threads can access them; threads can also access data in different memory spaces. Each SM used in the experiments has 256kB worth of memory registers [19].
In the case of shared memory, the bandwidth is similar to the registers, and threads can cooperate to load and compute data shared by them. Each memory module has a set of 32-bit registers, which makes the threads access consecutive positions of a data vector more efficiently. A module can receive multiple requests for the same data but this creates conflicts. However, automatic serialization satisfies all memory access requests. As this serialization can reduce bandwidth performance, a broadcast device is set up to prevent the reading of all the threads at the same memory address (NVIDIA, 2010). On the other hand, all threads can access the GPU global memory (GDRAM) simultaneously. However, there are some restrictions which improve the bandwidth. Global memory has the lowest bandwidth but has the largest

storage capacity. In order to obtain the maximum possible speed-up, a group of threads are used which has consecutive indices, and is bundled into a unit named a warp. Thus, a single SM can run multiple warps simultaneously. The size of a warp depends on the GPU specification (NVIDIA, 2010; Kirk & Hwu, 2010).

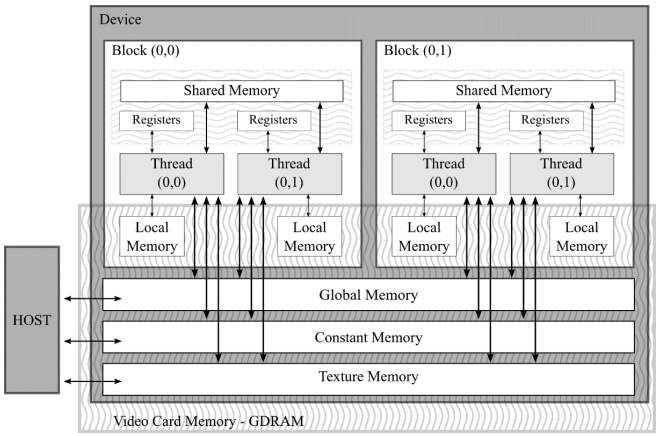

Fig. 4 - Memory spaces accessed by each thread (adapted from [41]).

All threads have read-only access to the GPU memory cache, which has 48kB for each SM; moreover, the threads of a half-warp can read only one memory address. Only instructions from the GPU can write into this kind of memory, and these processes persist throughout the execution of multiple kernel calls [19].

All threads can also access the texture memory, which is only read by kernels. This kind of memory uses a separate cache with a capacity of 32kB per SM, and provides high performance access when all threads perform operations on memory addresses close to them [40]. The types of access of on-chip memory for Compute 3.5 and later devices are indicated in Table 1.

Table 1 - Types of memory access in CUDA [42] (r - reading access, w - writing access, INT - internal memory space location, EXT - external memory space location).

Memory	Location	Access	Cached	Scope
Register	INT	r/w	no	One thread
Local	INT	r/w	yes	One thread
Shared	INT	r/w	N/A	All threads in a block
Global	EXT	r/w	yes	All threads + host
Constant	EXT	r	yes	All threads + host
Texture	EXT	r/w	yes	All threads + host

4.3. Implementation of Kernels in CUDA C

Tasks of computational image processing and analysis usually involve a large amount of data processing. Thus, the first strategy is to allocate the required space in the GDRAM, and then copy the input image as a data matrix from the host memory (RAM) to the device's memory (GDRAM); this process allows data to be managed directly in the GPU. Accesses to the coalesced memory are performed in contiguous

segments, half-warps access the segments simultaneously. Such accesses are known as coalesced memory accesses, and they enable parallel operations, thereby reducing the number of memory transactions [19, 39]. The data are then loaded into contiguous segments, and this allows a thread block to process an input image more efficiently; moreover, both the global memory and the texture memory are used.

Eqs. 5, 7 and 8 were implemented in the kernels called *kDiFinitas*, *kVariancia* and *kFinal*, respectively. The threads from the *kDiFinitas* kernel perform the computations in Eq. 5 in each image pixel independently. This kernel has several threads, each of which represent a matrix index, and process a specific image pixel. Thus, to manage access to a set of image pixels in the "for-loops", each pixel has an access condition.

First of all, in the *kDiFinitas* kernel, each pixel from the input image is associated with a thread, then the thread blocks are stored in the texture memory. After running the *kDiFinitas* kernel, the *kVariancia* kernel performs the parallelized computation of the λ parameter according to Eq. 7. In the parallel implementation, an auxiliary vector stores the values of the operations involved in each iteration, i.e. each thread calculates the resultant value of each iteration. Fig. 5 presents the pseudocode of the developped algorithm.

```
1   Input: Noisy image
2   /* Host program executed on CPU */
3   Allocate CPU and GPU memory
4   Store image to CPU memory
5   Copy image from CPU memory to GPU memory
6   Set the number of threads per blocks
7   /* kDiFinitas: Kernel program executed on each thread block */
8   Parallel each image pixel
        Compute the finite difference using Eq.(5)
9   /* kVariancia: Kernel program executed on each thread block */
10  Parallel each image pixel
        Compute weight parameter by Eq.(8)
        Call kFinal kernel
11  /* kFinal: Kernel program executed on each thread block */
12  Parallel each image pixel
        Create a new vector with zeros to store the smoothed image
        Compute the new pixel values for each new iteration using Eq.(7)
13  Copy image from GPU memory to CPU memory
14  Output: Smoothed image
```

Fig. 5 - Pseudocode of the developped parallel implementation.

The *kFinal* kernel computes the weight parameter, used previously in the *kVariancia* kernel, and then applies it to each image pixel, giving access to the texture cache and coalesced access to the global memory. Each thread attributes the resulting value to the corresponding memory, providing the data needed to calculate the final sum of each image pixel. Finally, the *SomaElem* kernel assists with the calculation of the vector values. The vector is divided into two equal parts, their values are summed, each thread sums two values, and keeps them in the lowest available vector position. This procedure continues until only one vector position remains for the storage of the resulting sum. In the case of a vector with an odd number of elements, an extra element with a zero value is added. This procedure uses the partitioning strategy of the global memory to optimize the bandwidth of the active warps during memory access; the

warps are organized into partitions. This is the slowest kernel used, and this is because the memory blocks become less contiguous while the elements are processed. Fig. 6 illustrates the implemented parallelization technique.

An image corrupted by multiplicative noise is used as input (Step 1), and after running the kernels described previously in steps 4 to 8, the result will be a new noise-smoothed image. Steps 4, 6, and 7 perform the reading of data in the texture memory. On the other hand, the results of each step of memory writing go into the global memory, where the output images are stored.

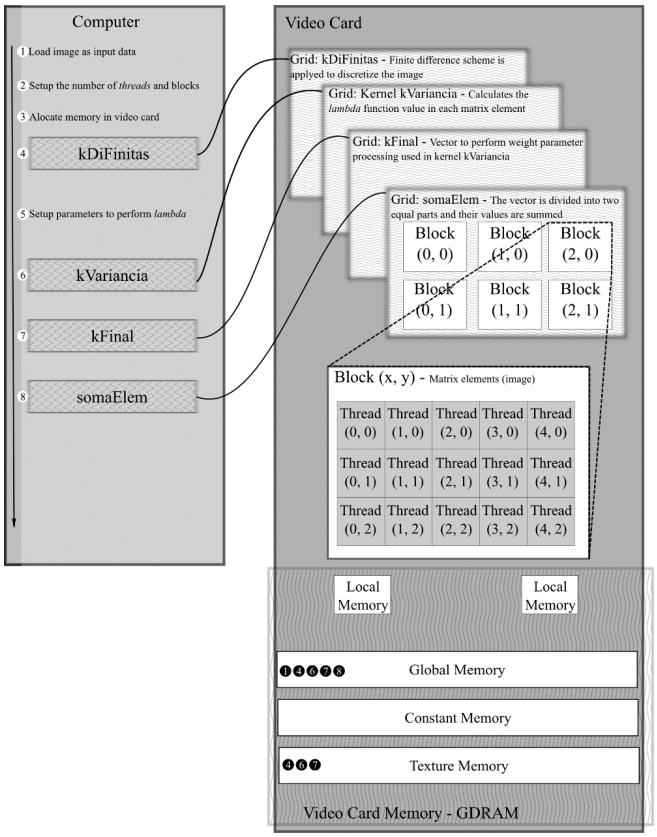

Fig. 6 - Parallel CUDA-based implementation of the adopted image smoothing method.

Eqs. 5, 7 and 8 were implemented as a nested “for-loop”. A CPU-based implementation was also developed as a comparison with the GPU-based implementation. The main memory system was accessed contiguously for all of these loops in order to optimize execution, and the GDRAM was accessed contiguously as well, creating a fair comparison [19] between the implementations.

5. Experiments and Discussion

This section describes the infrastructure used to perform the experiments and also discusses the results.

5.1. Test Infrastructure

The used test infrastructure includes a desktop computer equipped with an Intel(R) Core(TM) i7-4790 3.60 GHz processor, 16GB of RAM (DDR3 - 1600 MHz), Linux Ubuntu 14.04 operating system, CUDA[1] nvcc release 7.5 compiler driver, and GNU gcc/g++ compiler version 4.8.4. Additionally, there was a GPU NVIDIA Tesla K20c, with 2496 CUDA cores and 5GB of GDRAM.

5.2. Results and Discussion

In this section, we present results of experiments aimed at evaluating the performance of the method adopted. The runtime performance of GPU-based implementation is the focus of this study; however, the PSNR, SSIM and NCC metrics were used to confirm the smoothing method's accuracy. In the tests, 15, 25 and 50 smoothing iterations were adopted.

We used a set of six images with different resolutions (128x128, 256x256, 512x512, 1024x1024, 2048x2048 and 4096x4096 pixels) built synthetically with an image editor software and then corrupted with synthetic multiplicative noise of a variance equal to 0.3. There were 100 iterations for each test, and the mean and the standard deviation values of the time spent smoothing each input image were calculated. The total time spent (Table 2) was computed from the moment the data was loaded into the main memory system until the end of the smoothing process when the resultant image was produced. The function *cudaThreadSynchronize* was performed after each kernel call, forcing the CPU to wait for the complete kernel execution, and the *sdkResetTimer, sdkStartTimer* and *sdkStopTimer* timing functions were used to obtain the kernel execution time. The execution times of each kernel were added together to obtain the total execution time. Table 2 shows that the execution times of the CPU-based implementation were longer than those of the GPU-based implementation except in the case of the smallest test image (128x128 pixels). This distinct behavior occurred because the speed-up achieved with the data processed in the CUDA cores did not justify the computational effort involved in transferring a small amount of data to the GPU memory or the latency times necessary for the initialization of the GPU. For large images, the speed-up of the GPU was around 10, but less for smaller ones. Moreover, the GPU-based implementation achieved noise smoothing in real-time for all tested images. The parallel implementation had transparent and portable scalability in GPUs based on CUDA architecture; besides, the performance scales increased exponentially, as shown in Fig. 7. Furthermore, when we considered images with dimensions greater than 256x256 pixels, a speed-up of the GPU-based implementation was evident; for example, it was about 10.65 times faster for images with 4096x4096 pixels.

[1] CUDA compiler and development suite are available to download through the NVIDIA website https://developer.nvidia.com/cuda-downloads.

Table 2 - Comparison between the computational time (in milliseconds) required by the CPU- and GPU-based implementations to smooth the test static images with 50 iterations.

Images	Tesla	CPU
128x128	30.41 ± 0.81	14.08 ± 0.10
256x256	36.72 ± 0.20	56.37 ± 0.26
512x512	59.04 ± 0.96	225.90 ± 1.17
1024x1024	133.38 ±1.86	944.99 ± 18.34
2048x2048	423.94 ± 1.23	3761.56 ± 32.18
4096x4096	1617.16 ± 7.09	15180.35 ± 26.22

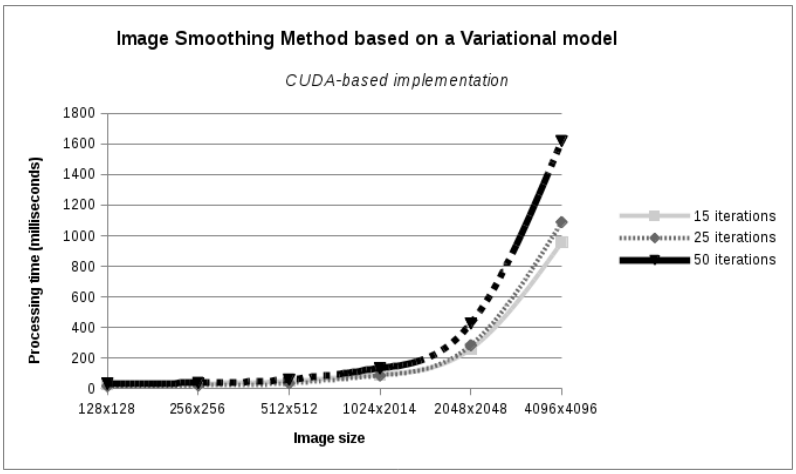

Fig. 7 - Processing time of the proposed GPU-based implementation, which scales up exponentially.

As an illustrative example, Fig. 8 shows the results of the CPU- and GPU-based implementations applied to the test images. Fig. 8 shows, from left to right, the image affected by the multiplicative noise, and the images smoothed by the CPU- and GPU-based implementations.

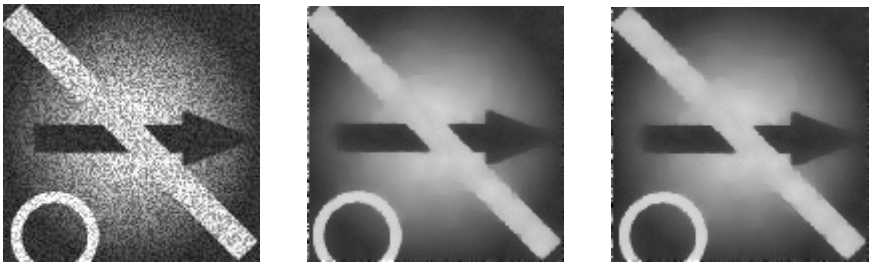

Fig. 8 - An original noisy test image with 4096x4096 pixels and the smoothed images obtained by the CPU- and GPU-based smoothing implementations, respectively.

The values listed in Table 3 were computed using NCC and SSIM metrics in order to confirm that the structural information resulting from the noise images corresponded to smoothed images, since all values were close to 1 (one). As for the smoothing method's accuracy and time performance, the optimal number of iterations for better image preservation seems to be 15.

The PSNR values were also computed for each static test image before and after being smoothed by the CPU- and GPU-based implementations (Table 4). The values demonstrate the efficiency of the smoothing method, and confirmed that the two implementations smoothed the images using the method adopted.

Table 3 - NCC and SSIM values computed for the static test images using 15, 25, and 50 iterations.

Images	NCC			SSIM		
	15	25	50	15	25	50
128x128	0.99999	0.99996	0.99962	0.99992	0.99869	0.98652
256x256	0.99973	0.99884	0.99752	0.99894	0.99612	0.98383
512x512	0.99667	0.99679	0.99554	0.99959	0.99932	0.99715
1024x1024	0.99664	0.99696	0.99587	0.99967	0.99967	0.99914
2048x2048	0.99819	0.99827	0.99798	0.99996	0.99995	0.99988
4096x4096	0.99889	0.99894	0.99864	0.99999	0.99999	0.99997

Table 4 - PSNR values computed for the static test images before (noisy) and after being smoothed by the CPU- and GPU-based implementations using 15, 25, and 50 iterations.

Images	PSNR						
	Noisy	GPU smoothed			CPU smoothed		
		15	25	50	15	25	50
128x128	+16.71226	+26.45354	+28.01845	+27.08323	+26.45101	+28.01758	+27.08876
256x256	+13.63118	+21.45840	+21.15450	+20.48712	+21.46481	+21.13883	+20.51067
512x512	+10.71005	+11.70306	+11.84286	+12.18230	+11.70372	+11.84272	+12.18366
1024x1024	+11.04700	+14.64996	+14.63267	+14.81482	+14.64584	+14.63146	+14.80796
2048x2048	+10.39326	+13.64743	+13.73511	+14.05504	+13.64635	+13.73472	+14.05501
4096x4096	+9.92586	+13.15495	+13.30716	+13.67279	+13.15451	+13.30648	+13.67202

We also tested three synthetic videos with 240 frames and different resolutions (128x128, 256x256, 512x512 pixels), and one real ultrasound video with 255 frames of 320x240 pixels. The smoothing method was applied only once for each video frame.

The average runtime for the real ultrasound video was 5.92 seconds for the CPU-based implementation, and 2.87 seconds for the parallel implementation in CUDA. Thus, the processing time of the parallel implementation was about 2.06 times faster when processing the entire ultrasound video. Fig. 9 shows an example of the smoothing of a video frame selected randomly from the tested video.

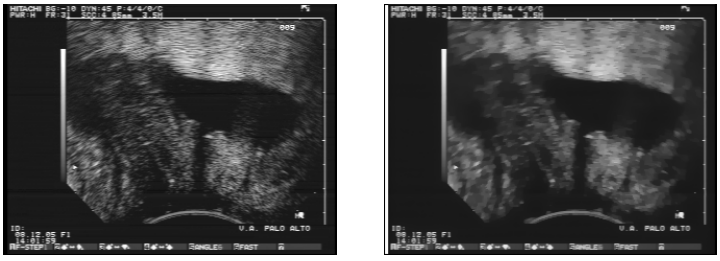

Fig. 9 - The original image and the image smoothed by the parallel implementation, respectively

Table 5 indicates the frame rates of the CPU- and the GPU-based implementations when smoothing the four test videos. In this table, the values in bold can be considered in line with real-time processing (> 20 frames per second), and therefore acceptable for routine medical image processing [43, 44]. As shown in Table 5, the experiments using the parallel GPU-based implementation revealed an even higher reduction in the runtime of the smoothing method in relation to the CPU-based implementation, confirming initial expectations.

The CUDA architecture as a computational infrastructure for image pre-processing has revealed to be a viable, capable and alternative option to deliver high-performance processing in many applications; moreover it can even provide real-time processing at an affordable cost [40]. Here, the performance gain of the parallel GPU-based implementation confirms the high processing capacity available in the CUDA architecture, with all videos used in the experiments processed in real-time. Today, the available resources in these graphic cards have increased the performance gain more efficiently, taking into consideration the number of cores and GDRAM memory as well as the SIMT parallel model associated with memory optimization techniques.

Table 5 - Frames per Second (FPS) rate obtained in the CPU- and GPU-based implementations with the smoothing method applied with 15, 25 and 50 iterations. (The values in bold can be considered in line with real-time processing.)

	Total of FPS rate obtained					
Video Resolution	**GPU**			**CPU**		
	15	25	50	15	25	50
128x128	**116.60**	**69.61**	**34.12**	**249.79**	**152.93**	**76.78**
256x256	**94.23**	**57.16**	**28.84**	**52.73**	**33.10**	16.39
320x240	**88.61**	**54.00**	**27.93**	**48.16**	**32.20**	16.81
512x512	**57.79**	**37.10**	19.59	13.75	8.44	4.21

Therefore, the benefit of using GPU-based implementations can be totally justified since the reduction in the runtime can minimize or even eliminate the time restrictions; such restrictions are common in many applications (such as in the medical field) that use image processing and analysis methods, requiring fast or real-time results for image-based diagnosis. However, optimal implementation requires maximum efforts, particularly when using the CUDA architecture.

6. Conclusions

The use of parallel computing techniques to fully explore the high-performance multiprocessor architecture is not new. However, the cost of the more traditional hardware for high-performance computing is not low; thus, more affordable alternatives such as GPU hardware should be considered.

The present work has described how to use the high-performance computing CUDA-based architecture as a computational infrastructure to accelerate an algorithm for noise image removal. The parallel GPU-based implementation developed was compared against the corresponding sequential CPU-based implementation in several experiments, and image quality metrics confirmed the similarity of the smoothing results achieved by each implementation. The parallelization of the image smoothing method based on a variational model using the CUDA architecture reduced the runtime by up to 10.65 times in comparison to the CPU-based implementation.

The novel CUDA-based implementation developped to smoothing multiplicative noise by using an effective variational method seems to be a high-performance solution for applications with images

susceptible to this type of noise, and which have high processing time constraints. Moreover, the proposed GPU-based parallelization approach has transparency, portability and scalability, thanks to the adopted SIMT model.

More and more complex methods and larger and larger data sets are used in the medical imaging domain that has high time constraints, which makes the use of the CUDA architecture extremely attractive as the the study conducted here confirms. As a future works, we intend to extend the proposed CUDA-based implementation to enable it to perform in multi-GPUs, besides combining it with multithread (OpenMP) and multicomputer (MPI) in order to achieve higher performances using heterogeneous parallel computing platforms.

7. Acknowledgements

The first author would like to thank the "Universidade do Estado de Mato Grosso" (UNEMAT), in Brazil, for the support given. The National Scientific and Technological Development Council (CNPq) partially supported this work through process 234360/2014-9 and grant 2010/15691-0.

Henrique Ferraz de Arruda thanks the Coordination for the Improvement of Higher Education Personnel (CAPES) for the financial support received.

Authors gratefully acknowledge the funding of Project NORTE-01-0145-FEDER-000022 - SciTech - Science and Technology for Competitive and Sustainable Industries, cofinanced by "Programa Operacional Regional do Norte" (NORTE2020), through "Fundo Europeu de Desenvolvimento Regional" (FEDER).

References

[1] Z. Ma, J.M.R.S. Tavares, R.N. Jorge, T. Mascarenhas, "A review of algorithms for medical image segmentation and their applications to the female pelvic cavity," *Computer Methods in Biomechanics and Biomedical Engineering,* vol. 13, nº 2, pp. 235-246, 2010.

[2] H. Erives, G.J. Fitzgerald, "Automated registration of hyperspectral images for precision agriculture," *Computers and Electronics in Agriculture,* vol. 47, nº 2, pp. 103-119, 2005.

[3] R. Arjona, I. Baturone, "A hardware solution for real-time intelligent fingerprint acquisition," *Journal of Real-Time Image Processing,* vol. 9, nº 1, pp. 95-109, 2014.

[4] L. Chen, M. Zhang, Z. Xiong, "Series-parallel pipeline architecture for high-resolution catadioptric panoramic unwrapping," *IET-Imaging Processing,* vol. 4, nº 5, pp. 403-412, 2010.

[5] I. Kunttu, L. Lepisto, "Shape-based retrieval of industrial surface defects using angular radius Fourier descriptor," *IET Image Processing,* vol. 1, nº 2, pp. 231-236, 2007.

[6] T. Mélange, M. Nachtegael, S. Schulte, E. E. Kerre, "A fuzzy filter for the removal of random impulse noise in image sequences," *Image and Vision Computing,* vol. 29, nº 6, pp. 407-419, 2011.

[7] T. Aittokallio, J. Salmi, T.A. Nyman, O.S. Nevalainen, "Geometrical distortions in two-dimensional gels: applicable correction methods," *Journal of chromatography. B, Analytical technologies in the biomedical and life sciences,* vol. 815, nº 1-2, pp. 25-37, 2005.

[8] R.K. Jha, P.K. Biswas, B.N. Chatterji, "Contrast enhancement of dark images using stochastic resonance," *IET Image Processing,* vol. 6, nº 3, pp. 230-237, 2012.

[9] Y.-D. Wu, Y. Sun, H.-Y. Zhang, S.-X. Sun, "Variational PDE based image restoration using neural network," *IET Image Processing,* vol. 1, nº 1, pp. 85-93, 2007.

[10] M. Ezoji, K. Faez, "Use of matrix polar decomposition for illumination-tolerant face recognition in discrete cosine transform domain," *IET Image Processing,* vol. 5, nº 1, pp. 25-35, 2011.

[11] Z. Ma, R.N.M. Jorge, J.M.R.S. Tavares, "A shape guided C-V model to segment the levator ani muscle in axial magnetic resonance images," *Medical Engineering & Physics,* vol. 32, nº 7, pp. 766-774, 2010.

[12] F.M.P. Oliveira, T.C. Pataky, J.M.R.S. Tavares, "Registration of pedobarographic image data in the frequency domain," *Computer Methods in Biomechanics and Biomedical Engineering,* vol. 3, nº 6, pp. 731-740, 2010.

[13] Q. Chen, Q.-sen Sun, D.-shen Xia, "Homogeneity similarity based image denoising," *Pattern Recognition,* vol. 43, nº 12, pp. 4089-4100, 2010.

[14] V.I. Ponomaryov, "Real-time 2D-3D filtering using order statistics based algorithms," *Journal of Real-Time Image Processing,* vol. 1, nº 3, pp. 173-194, 2007.

[15] F.P.X. Fontes, G.A. Barroso, P. Coupé, P. Hellier, "Real-time ultrasound image denoising," *Journal of Real-Time Image Processing,* vol. 6, nº 1, pp. 15-22, 2011.

[16] A. Merigot, A. Petrosino, "Parallel processing for image and video processing: Issues and challenges," *Parallel Computing*, 2008.

[17] E. López-Rubio, "Restoration of images corrupted by Gaussian and uniform impulsive noise," *Pattern Recognition,* vol. 43, nº 5, pp. 1835-1846, 2010.

[18] Z. Jin, X. Yang, "A variational model to remove the multiplicative noise in ultrasound images," *Journal of Mathematical Imaging and Vision,* vol. 39, nº 1, pp. 62-74, 2011.

[19] NVIDIA, "GPU Tutorial: Build environment, Debugging/Profiling, Fermi, Optimization/CUDA 3.1 and Fermi advice," 2010.

[20] N. Wilt, The CUDA Handbook: A comprehensive guide to GPU programming, Addison-Wesley, 2013.

[21] D. Castano-Diez, D. Moser, A. Schoenegger, S. Pruggnaller, A.S. Frangakis, "Performance evaluation of image processing algorithms on the GPU," *Journal of Structural Biology,* vol. 164, nº 1, pp. 153-160, 2008.

[22] G.A. Triantafyllidis, M. Varnuska, D. Sampson, D. Tzovaras, M.G. Strintzis, "An efficient algorithm for the enhancement of JPEG-coded images," *Computer & Graphics,* vol. 27, nº 4, pp. 529-534, 2003.

[23] J. Ji, "Robust approach to independent component analysis for SAR image analysis," *IET - Image Processing,* vol. 6, nº 3, pp. 284-291, 2012.

[24] G.A.J. Aubert, J.-F. Aujol, "A variational approach to removing multiplicative noise," *SIAM Journal of Applied Mathematics,* vol. 68, nº 4, pp. 925-946, 2008.

[25] Y.-M. Huang, M.K. Ng, Y.-W. Wen, "A new total varation method for multiplicative noise removal," *SIAM Journal on Imaging Sciences,* vol. 2, nº 1, pp. 20-40, 2009.

[26] L.I. Rudin, S. Osher, E. Fatemi, "Nonlinear total variation based noise removal algorithms," *Journal of Physica D,* vol. 60, nº 1-4, pp. 259-268, 1992.

[27] K. Krissian, R. Kikinis, C.-F. Westin, K. Vosburgh, "Speckle-constrained filtering of ultrasound images," *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2005.

[28] M.P. Eckert, A.P. Bradley, "Perceptual quality metrics applied to still image compression," *Signal Processing,* vol. 70, nº 3, pp. 177-200, 1998.

[29] Z. Wang, A.C. Bovik, H.R. Sheikh, E.P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing,* vol. 13, nº 4, pp. 600-612, 2004.

[30] F.P.M. Oliveira, J.M.R.S. Tavares, "Medical image registration: A Review," *Computer Methods in Biomechanics and Biomedical Engineering,* vol. 17, nº 2, pp. 73-93, 2014.

[31] S. Winkler, "Issues in vision modeling for perceptual video quality assessment," *Signal Processing,* vol. 78, nº 2, pp. 231-252, 1999.

[32] G. Ramponi, N.K. Strobel, S.K. Mitra, T.-H. Yu, "Nonlinear unsharp masking methods for image contrast enhancement," *Journal of Electronic Imaging,* vol. 5, nº 3, pp. 353-367, 1996.

[33] S. Hashemi, S. Kiani, N. Noroozi, M.E. Moghaddam, "An image contrast enhancement method based on genetic algorithm," *Pattern Recognition Letters,* vol. 31, nº 13, pp. 1816-1824, 2010.

[34] O. Ghita, P.F. Whelan, "A new GVF-based image enhancement formulation for use in the presence of mixed noise," *Pattern Recognition,* vol. 43, nº 8, pp. 2646-2658, 2010.

[35] Y. Shkvarko, A.C. Atoche, D. Torres-Roman, "Near real time enhancement of geospatial imagery via systolic implementation of neural network-adapted convex regularization techniques," *Pattern Recognition Letters,* vol. 32, nº 16, pp. 2197-2205, 2011.

[36] R.S. Alves, J.M.R.S. Tavares, "Computer Image Registration Techniques Applied to Nuclear Medicine Images," *Computational and Experimental Biomedical Sciences: Methods and Applications*, vol. 21, J.M.R.S. Tavares and R.M.N. Jorge, Eds., Springer, 2015, pp. 173-191.

[37] A. Nakhmani, A. Tannenbaum, "A new distance measure based on generalized image normalized cross-correlation for robust video tracking and image recognition," *Pattern Recognition Letters,* vol. 34(3), pp. 315-321, Feb 2013.

[38] Z. Wang, A.C. Bovik, L. Lu, "Why is image quality assessment so dificult," *ICASSP International Conference on Acoustics, Speech, and Signal Processing*, 2002.

[39] L. Zhang, W. Dong, D. Zhang, G. Shi, "Two-stage image denoising by principal component analysis with local pixel grouping," *Pattern Recognition,* vol. 43, nº 4, pp. 1531-1549, 2010.

[40] D. Kirk, W.-M. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Elsevier, 2010, p. 75.

[41] I. K. Park, N. Singhal, M.H. Lee, S. Cho, C.W. Kim, "Design and performance evaluation of image processing algorithms on GPUs," *IEEE Transactions on Parallel and Distributed Systems,* vol. 22, nº 1, pp. 91-104, 2011.

[42] W.-m.W. Hwu, GPU Computing GEMS, Emerald ed., Morgan Kaufmann and NVIDIA, 2011.

[43] R. Farber, CUDA Application Design and Development, Elsevier, 2011.

[44] N. Kehtarnavaz, M.N. Gamadia, Real-Time image and video processing: From research to reality, First ed., University of Texas at Dallas, USA: Morgan & Claypool Publishers, 2006.

[45] D. Levin, U. Aladl, G. Germano, P. Slomka, "Techniques for efficient, real-time, 3D visualization of multi-modality cardiac data using consumer graphics hardware," *Computerized Medical Imaging and Graphics ,* vol. 29, nº 6, pp. 463-475, 2005.

[46] E. Todorovich, A.L.D. Pra, L.I. Passoni, M. Vázquez, E. Cozzolino, F. Ferrara, G. Bioul, "Real-time speckle image processing," *Journal of Real-Time Image Processing,* pp. 1-11, 2013.