


A fast Hough Transform algorithm for straight lines detection in an image using GPU parallel computing with CUDA-C

R. Yam-Uicab¹ · J. L. Lopez-Martinez¹  ·
J. A. Trejo-Sanchez^{2,3} · H. Hidalgo-Silva⁴ ·
S. Gonzalez-Segura¹

Published online: 20 April 2017

© Springer Science+Business Media New York 2017

Abstract The Hough Transform (HT) is a digital image processing method for the detection of shapes which has multiple uses today. A disadvantage of this method is its sequential computational complexity, particularly when a single processor is used. An optimized algorithm of HT for straight lines detection in an image is presented in this article. Optimization is realized by using a decomposition of the input image recently proposed via central processing unit (CPU), and the technique known as segment decomposition. Optimized algorithms improve execution times significantly. In this paper, the optimization is implemented in parallel using graphics processing unit (GPU) programming, allowing a reduction of total run time and achieving a performance more than 20 times better than the sequential method and up to 10 times better than the implementation recently proposed. Additionally, we introduce the concept of Performance Ratio, to emphasize the outperforming of the GPU over the CPUs.

✉ J. L. Lopez-Martinez
jose.lopez@correo.uady.mx

R. Yam-Uicab
reyesyamm@gmail.com

J. A. Trejo-Sanchez
joel.trejo@cimat.mx

H. Hidalgo-Silva
hugo@cicese.mx

S. Gonzalez-Segura
sergio.gonzalez@correo.uady.mx

¹ Facultad de Matematicas, Universidad Autonoma de Yucatan, Mérida, Mexico

² Conacyt-Centro de Investigacion en Matematicas, Mérida, Mexico

³ Basic Sciences Department, Universidad del Caribe, Cancún, Mexico

⁴ Centro de Investigacion Cientifica y Educacion Superior de Ensenada, Ensenada, Mexico

Keywords Hough Transform · Line detection · GPU programming · Parallel computing · Pyramidal decomposition · Performance Ratio

1 Introduction

Digital image processing and computational vision allow the computer to interact with the real world, achieving the completion of more precise and generally faster work compared with that done by humans. Shape detection (i.e., lines, ellipses, curves) has several uses in real applications. One of the most commonly used methods for the detection of lines and curves in an image is the Hough Transform (HT) [1–8]. Its ease of implementation and sturdiness against image noise make it a good option when compared with other methods with the same purpose [9]. However, one of the disadvantages of the method is the computational complexity required for processing the image and storage of the data. Parallel computing is an alternative that has been used in several projects in an attempt to diminish complexity of the HT, achieving results in a shorter timescale. For example, in [3] the parallelization of the HT is presented for the detection of ellipses using a GPU achieving a reduction in the order of complexity of the initial algorithm from $O(N^5)$ to $O(N^2 + NE)$ where E is the number of edge points of the pre-processed image (to obtain the edges). Another example is presented in [2], where a method called *Additive Hough Transform (AHT)* is used to detect straight lines, through the division of an image with $m \times m$ pixels into k^2 blocks each one with $(\frac{m}{k})^2$ pixels. Each block is processed in parallel through addition properties applicable to the calculation of HT in each pixel. Another implementation is presented in [10] for a real-time line detection system. In this case, before carrying out the HT process the image is pre-processed with an edge detection filter and a *Kalman* filter is applied later to reduce the possible calculation regions. The parallel process is done with GPU with CUDA (Compute Unified Device Architecture) architecture for the operations of each pixel needed by the HT. In [2, 11–13], they implement the HT using specialized hardware.

An optimized implementation of the Hough Transform is presented in this work, using decomposition techniques. Two decomposition methods of an input image are compared, the first known as decomposition by segments (segmentation-based method) and the second as decomposition by *intercalation* or *decimation technique* (intercalation-based method). The decomposition techniques are implemented using the CUDA parallel computing platform to program in a NVIDIA GPU.

We divide the paper as follows. In Sect. 2, we describe the Hough Transform. Section 3 presents the parallel computing platform and CUDA programming model. Next, Sect. 4 describes the proposed parallel algorithm. Finally, Sect. 5 presents the experimental results, and the conclusions are shown in Sect. 6.

2 The Hough Transform

In digital image processing and computer vision, it is a common task the detection of forms like lines in certain images. One commonly used method for this purpose is the Hough Transform (HT) proposed by Paul V. C. Hough in 1962. The HT represents a

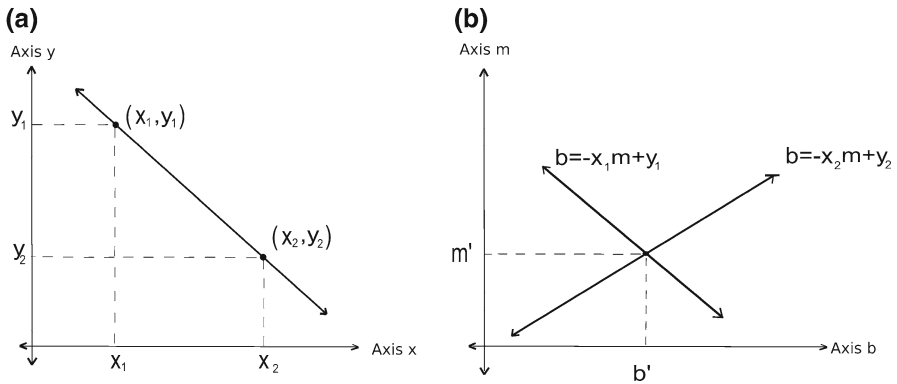


Fig. 1 **a** Both points (x_1, y_1) and (x_2, y_2) are located along the *same line* (collinear points), in the xy plane. Both points satisfy the equation of the straight line $y_i = m'x_i + b'$ for the corresponding constants m' and b' , respectively. **b** Parameter space based on m and b , where the equation $y_1 = mx_1 + b$ is associated with the point (x_1, y_1) , and the equation $y_2 = mx_2 + b$ is associated with the point (x_2, y_2) . Both lines y_1 and y_2 intersect in the point (m', b') . A set of points are collinear in xy plane if their associated lines in mb plane intersect in the same point

line with its equation $y_1 = mx_1 + b$, where (x_1, y_1) represents a point in the xy plane satisfying such equation. The constants m and b denote the slope and the intercept, respectively. These constants identify uniquely the line among any other line that goes through the point (x_1, y_1) . The HT method consists in represent the equation of the line as $b = -x_1m + y_1$, such that the point (x_1, y_1) is fixed. In this new representation, m and b are the coordinates in the mb plane, and x and y are the constants. Two points (x_1, y_1) and (x_2, y_2) in the xy plane are collinear if their associated lines in the mb plane intersect in a single point (m', b') (see Fig. 1).

A drawback of this representation is the difficulty of detecting vertical lines, owing to the fact that m can become infinite.

To solve this problem, Duda and Hart [1] proposed to use the representation of straight lines based on polar coordinates (Eq. 1) instead of the usual Cartesian coordinates. This representation uses the parameters ρ (distance of the line from its origin) and θ (the angle of the vector for the abscissae), which represent the new parameter space.

$$\rho = x \cos \theta + y \sin \theta. \tag{1}$$

When using this parameter space (ρ, θ) , sine waves instead of straight lines will be associated with each point in the parameter space, as observed in Fig. 2b.

The main advantage of the HT is that it is easy to implement as a computer program. The space of parameters $\rho\theta$ is represented by a matrix M of accumulators. The dimensions of matrix M are given by the range of the parameters θ and ρ , where $-90 < \theta < 90$, $-D < \rho < D$, where D is the maximum distance between opposite corners in a digital image [7]. The position $M(i, j)$ in the matrix of accumulators M represents the coordinates (i, j) and denotes the associated value (ρ_i, θ_j) of the space of parameters $\rho\theta$. Non-background pixels of a digital image represent points in the xy

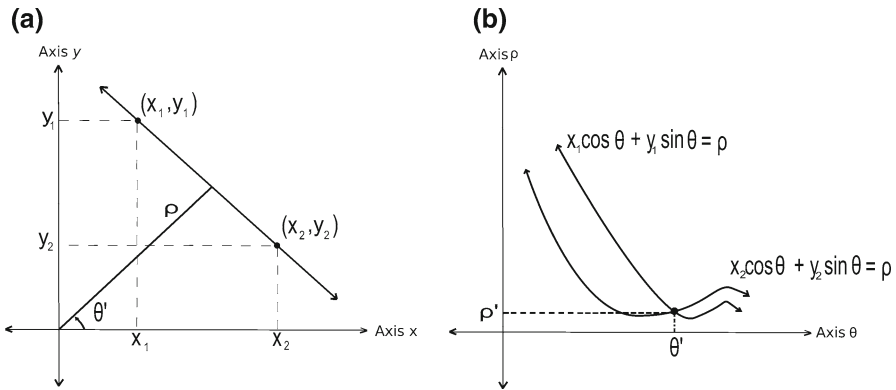


Fig. 2 **a** Parametrization of lines on xy plane, where points (x_1, y_1) and (x_2, y_2) are collinear and satisfy the equation $\rho' = x \cos \theta' + y \sin \theta'$ to the corresponding parameters ρ' and θ' . **b** Sine waves on $\rho\theta$ plane, where points (x_1, y_1) and (x_2, y_2) have parametric equations $\rho = x_1 \cos \theta + y_1 \sin \theta$ and $\rho = x_2 \cos \theta + y_2 \sin \theta$, respectively, associated with it. Both sinusoidal curves intersect in point (ρ', θ') . The point (ρ', θ') corresponds to the line that goes through points (x_1, y_1) and (x_2, y_2) in the xy plane

plane. The basic sequential algorithm [7] is represented next. This algorithm receives as input an Image I of $N \times N$ size.

1. Obtain I_b , result of binarizing I .
2. Quantize parameter space (ρ, θ) into accumulator cells $M[\rho, \theta]$, $\rho \in [\rho_{\min}, \rho_{\max}]$; $\theta \in [\theta_{\min}, \theta_{\max}]$.
3. Initialize all cells to 0.
4. For each foreground point (x_k, y_k) in the thresholded edge image I_b :
 - For each point θ_j equal all possible θ -values
 - Solve for ρ using $\rho = x_k \cos \theta_j + y_k \sin \theta_j$
 - Round ρ to the closest cell value, ρ_q
 - Increment $M(\rho, q)$ if θ_p results in ρ_q
5. Find line candidates where $M(i, j)$ is above a suitable threshold value.
6. Return lines $\rho_i = x \cos \theta_j + y \sin \theta_j$.

3 Parallel computing platform and CUDA programming model

With the increasing prevalence and easy access to GPUs, many developers, researchers and scientists have made wide-ranging use of computing. The GPU manufacturer NVIDIA provides a general purpose language enabling users to program using their graphics cards.

There are issues that have to be considered when programming on a GPU. Those issues occur due to the different architectures of a GPU. A clear example is when managing memory space, in this regard CUDA comes with functions that automate operations, a similar process to that seen in C language. In versions prior to 6 of CUDA, an explicit copy had to be made from the CPU's memory to the GPU's memory and vice-versa. This had among other repercussions, the impossibility of writing directly to the GPU's memory from a function as host, as well as implying more lines of code for the programmer.

<p>(a)</p> <pre>int main(){ int L=100; int *A, *dev_A; A=(int *) malloc (L*sizeof(int)); for(int i=0;i<L;i++) A[i]=i+1; cudaMalloc((void **)&dev_A,L*sizeof(int)); cudaMemcpy(dev_A,A,L*sizeof(int),cudaMemcpyHostToDevice); ... return 0; }</pre>	<p>(b)</p> <pre>int main(){ int L=100; int *A; cudaMallocManaged(&A,L*sizeof(int)); for(int i=0;i<L;i++) A[i]=i+1; ... return 0; }</pre>
--	--

Fig. 3 **a** CUDA syntax for memory management in versions earlier than v6. **b** CUDA syntax for memory management using Unified Memory. In both codes, a single array with a size of 100 is generated for use on the GPU

Since version 6, CUDA introduced a new component called Unified Memory, which allows memory space to be managed in a new way [14]. With this new component, memory usage is simplified, and an example of this can be seen in a comparison between code examples in Fig. 3. Currently CUDA is on version 8.

For the HT implementation in CUDA-C, we use a Unified Memory for memory for memory space management, as both load (reading) and store (writing) of data between system memory (CPU) and device memory (GPU).

The *streams* are virtual job queues used in CUDA for asynchronous operations on GPUs [15]. However, when asynchronous commands are run in CUDA without specifying a stream, a predetermined stream is used, which generates an implicit synchronization [16]. This happens for commands like calls to *kernels*, or memory copies between two addresses to the same device memory, among others.

4 The proposed parallel algorithm

Now, we present our optimization method for the HT for straight lines recognition in digital images using CUDA-C of Nvidia for parallel programming by using GPU. This parallel optimization is implemented in both decomposition techniques (segmentation and intercalation method). In [17], the previous work uses only four CPU cores for both the split of the image, and for the voting phase. The main contribution is the implementation of a parallel algorithm to both, the decomposition (Sects. 4.2 and 4.3), and the voting phase. In the parallel implementation of the voting phase, we use four GPU *kernels*. Each *kernel* generates a set of threads to perform the voting phase. The set of threads is generated in execution time using the variable MPQ (Maximum Point Quantity). In Sect. 5, we emphasize the performance of our current work, comparing it with the previous work.

4.1 Segmentation-based decomposition

In this method, the input image is divided into four quadrants (Fig. 4), similar to the Cartesian plane, generating four subimages. Then, each subimage (in is associated

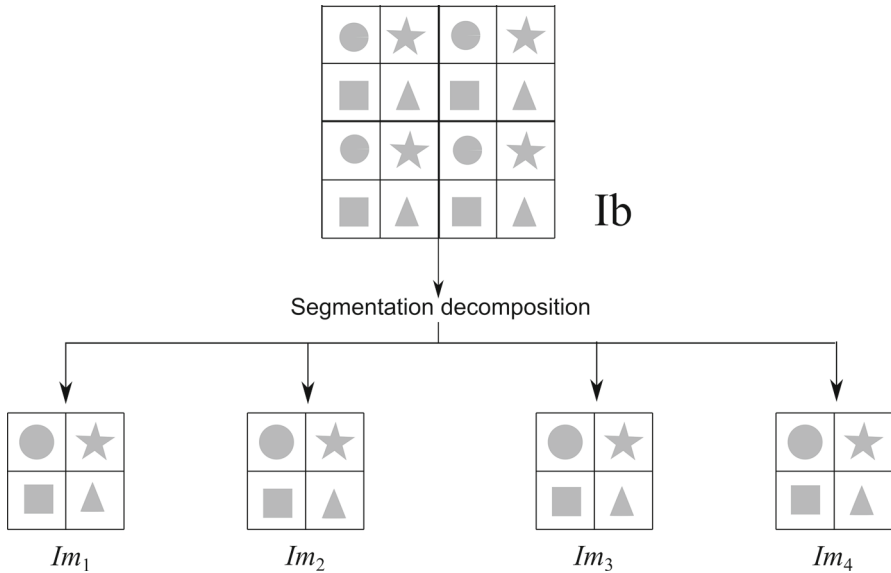


Fig. 4 Decomposition by segments of an image Ib into 4 subimages *Im*₁, *Im*₂, *Im*₃ and *Im*₄. The *geometric figures* represent the position of pixels contained in the original image and its reallocation in the new subimages

quadrant) is processed by the straight line recognition method (HT). The model of segmentation-based decomposition is given by

$$\begin{aligned}
 Im_1 &= \left\{ Ib(i, j), \quad i = 1, \dots, \left\lceil \frac{N}{2} \right\rceil; j = 1, \dots, \left\lceil \frac{N}{2} \right\rceil \right\}, \\
 Im_2 &= \left\{ Ib(i, j), \quad i = 1, \dots, \left\lceil \frac{N}{2} \right\rceil; j = \left\lceil \frac{N}{2} \right\rceil + 1, \dots, N \right\}, \\
 Im_3 &= \left\{ Ib(i, j), \quad i = \left\lceil \frac{N}{2} \right\rceil + 1, \dots, N; j = 1, \dots, \left\lceil \frac{N}{2} \right\rceil \right\}, \\
 Im_4 &= \left\{ Ib(i, j), \quad i = \left\lceil \frac{N}{2} \right\rceil + 1, \dots, N; j = \left\lceil \frac{N}{2} \right\rceil + 1, \dots, N \right\}.
 \end{aligned} \tag{2}$$

Where $N \times N$ is the size of input image Ib. For example, if this method receives as input an image Ib of 256×256 pixels, it split the image as follows. The first subimage consists of the subarray of pixels $Ib[1 \dots 128][1 \dots 128]$. The second subimage consists of the subarray of pixels $Ib[1 \dots 128][129 \dots 256]$. The third subimage consists of the subarray of pixels $Ib[129 \dots 256][1 \dots 128]$. Finally, the fourth subimage consists of the subarray of pixels $Ib[129 \dots 256][129 \dots 256]$.The decomposition is based on divide and conquer in the first level, where each subimage (in its associated quadrant) is processed by the straight line recognition method (HT). In Sect. 4.3, we give a more detailed explanation of the implementation.

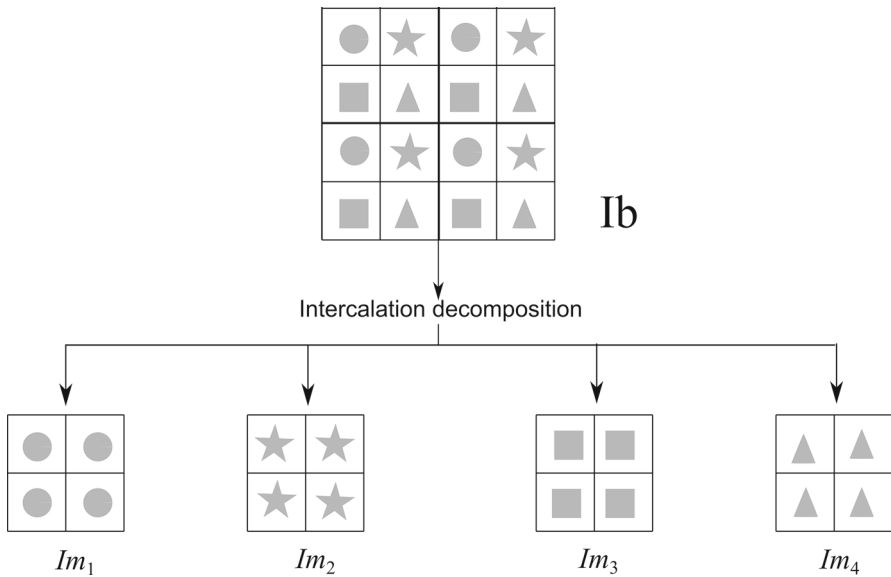


Fig. 5 Decomposition by intercalation of an image I_b into four subimages Im_1, Im_2, Im_3 and Im_4 . The *geometric figures* represent the position of pixels contained in the original image and its reallocation in the new subimages

4.2 Intercalation-based decomposition

This decomposition method was used in [17], where a parallel implementation using four CPUs was presented. This decomposition technique guarantees homogeneous load across each processor, reducing the total run time compared with both the segmentation method and the non-decomposition method. This technique, known as decimation technique, decomposes the image in four subimages when a factor of 2 is considered (see Fig. 5). The model of segmentation-based decomposition is given by

$$\begin{aligned}
 Im_1 &= \left\{ Ib(2i - 1, 2j - 1), \quad i = 1, \dots, \left\lceil \frac{N}{2} \right\rceil; j = 1, \dots, \left\lceil \frac{N}{2} \right\rceil \right\}, \\
 Im_2 &= \left\{ Ib(2i - 1, 2j), \quad i = 1, \dots, \left\lceil \frac{N}{2} \right\rceil; j = \left\lceil \frac{N}{2} \right\rceil + 1, \dots, N \right\}, \\
 Im_3 &= \left\{ Ib(2i, 2j - 1), \quad i = \left\lceil \frac{N}{2} \right\rceil + 1, \dots, N; j = 1, \dots, \left\lceil \frac{N}{2} \right\rceil \right\}, \\
 Im_4 &= \left\{ Ib(2i, 2j), \quad i = \left\lceil \frac{N}{2} \right\rceil + 1, \dots, N; j = \left\lceil \frac{N}{2} \right\rceil + 1, \dots, N \right\}.
 \end{aligned} \tag{3}$$

Where $N \times N$ is the size of input image I_b . For example, if this method receives as input an image I_b of 256×256 pixels, it split the image as follows. The first subimage consists of the subarray of pixels $I_b[1, 3, \dots, 253, 255]$ $[1, 3, \dots, 253, 255]$. The second subimage consists of the subarray of pixels $I_b[1, 3, \dots, 253, 255][2, 4, \dots, 254, 256]$. The third subimage consists of the subarray of

pixels $I_b[2, 4, \dots, 254, 256][1, 3, \dots, 253, 255]$. Finally, the fourth subimage consists of the subarray of pixels $I_b[2, 4, \dots, 254, 256][2, 4, \dots, 254, 256]$. Implementation of HT algorithm using the four subimages obtained using the intercalation-based decomposition method is detailed in Sect. 4.3.

4.3 Parallel algorithm

The parallel algorithm implemented on GPU with CUDA is described next. It is worth mentioning that for the implementation three versions of the HT algorithm were considered, the first one using the sequential, non-decomposition method, the second using the technique of segment decomposition in parallel, and the third one using the intercalation-based decomposition in parallel. A comparison of these three implementations is presented in Sect. 5. Before applying the HT to the input image, we apply the Canny filter [5, 18]. The Canny filter returns a binary edge image. We select the Canny filter since this filter is robust and low error rate in the digital image processing.

The steps of the algorithm are as follows:

1. The input is an image I with a size of $N \times N$ pixels, where N is known. Then, image is binarized (I_b) with the *Canny* filter, to obtain the main edges of the image.
2. Binarized image is decomposed using a factor $L = 2$. At this point, we apply the first optimization [17]: they use four processors $p_i, i = 1, \dots, 4$, where each processor is responsible for generating one of the four subimages Im_1, Im_2, Im_3 and Im_4 , achieving a reduced time compared with the decomposition of the image using a single processor. However, it is possible to generate the four subimages much more quickly using the GPUs. In this case, we decided to use $2N$ threads. Every $N/2$ thread generates one subimage, where each thread is responsible for filling one row from each subimage, allowing the generation time for the four subimages to be reduced.
3. Four accumulator matrices are created to simulate the parameter space. These matrices are initialized to zero, acting as voting matrices $M_i, i = 1, \dots, 4$, one for each subimage. Each M_i has the following ranges: $-90 < \theta < 90, -D < \rho < D$, where D is the largest distance in the image (diagonal) [7].
4. Then, for improved treatment of the subimages, a series of processes are applied to obtain only those points containing values equal to one (remember that the subimages are binarized). Eight vectors, $(\vec{X}_i, \vec{Y}_i), i = 1, \dots, 4$, are obtained from this step. Each pair of vectors (\vec{X}_i, \vec{Y}_i) stores the points of the corresponding subimage.
5. HT is obtained from each pair of vectors $\vec{X}_i, \vec{Y}_i, i = 1, \dots, 4$. This step represents the most important optimization. Next, we describe this optimization. Given the huge quantity of threads contained within a GPU, it is possible to process multiple points at the same time, improving the total processing time. The four *kernels* are created, each one containing an individual *stream*, this to avoid the implicit synchronization mentioned earlier. Each *kernel* will be responsible for one of the aforementioned pairs of vectors. However, due to differences among GPUs, some

possess greater capacity for parallel computing than others. For this reason MPQ (Maximum Point Quantity) variable is defined, which will determine the maximum number of points responsible for analyzing a thread. In this case, the maximum value for MPQ is $\vec{X}_i.length$ as this would determine that a single thread would be responsible for processing all the points. The minimum value for MPQ would be 1, which would mean that each point would be processed by one thread. This variable is very important in that it provides for granularity of parallelism in the algorithm. In this way, the number of threads assigned to each *kernel* will be given by Eq. (4). Figure 6 illustrates the functionality of this step, using the intercalation method to decompose the image.

6. After the previous processing, accumulator matrices $M_i, i = 1, \dots, 4$, are obtained. However, to work with one single accumulated matrix, we proceed to add the four previous ones forming a new matrix M_t . To increase the processing of this stage, we used a *kernel* with $\rho \times \theta$ threads.
7. Finally, from the accumulator M_t matrix we proceed to find the peaks of the cells using a predefined threshold, obtaining updated parameters which are verified with the input image.

$$threadnumber_i = \frac{\vec{X}_i}{MPQ}. \tag{4}$$

The Pseudocode 1 presents the proposed algorithm. The algorithm receives as input the image \mathbf{I} , of dimension $\mathbf{N} \times \mathbf{N}$, \mathbf{I}_b is the binary representation obtained with the *Canny* filter. $\mathbf{Im}_i, i = 1, \dots, 4$ represents the four subimages that are generated upon decomposing the initial image, binarized through a CUDA *kernel* with $2N$ threads. $\mathbf{VecX}_i, \mathbf{VecY}_i, i = 1, \dots, 4$, are the vectors that store the points for each corresponding subimage. $\mathbf{M}_i, i = 1, \dots, 4$ are the corresponding accumulator matrices for each subimage, which are filled by calling each *kernel* that executes the Hough Transform for each subimage. Values ρ and θ are the ranges (dimension) of the voting matrix described in step 3 of the algorithm. \mathbf{M}_t is the accumulator matrix where the four previous matrices are added through a *kernel* with $\rho \times \theta$ threads. \mathbf{P} is a vector with the parameters necessary to trace the lines in the original image. \mathbf{I}_f is the graphic of the initial image \mathbf{I} and the detection of the lines \mathbf{P} .

The functions with the *kernel* prefix are run on the GPU in lines 4-10. The number of threads that will be used with the respective *kernel* are specified between the symbols “<<<” and “>>>”. The lines 6 to 9 of Pseudocode 1 make reference to the *kernels* kernel_HT1, kernel_HT2, kernel_HT3 and kernel_HT4. These *kernels* perform the parallelization of the voting phase (see the step number 4 of basic sequential algorithm of HT in Sect. 2) for each pair of vectors ($\mathbf{VecX}_i, \mathbf{VecY}_i$) associated with the subimage \mathbf{Im}_i , where $i = 1, \dots, 4$. Notice that the *kernels* kernel_HT1, kernel_HT2, kernel_HT3 and kernel_HT4 are executed concurrently. Finally, the *kernel* kernel_adds perform the parallelization of the sum of the accumulator matrices M_1, M_2, M_3 , and M_4 obtained from the previous *kernels* to generate the final accumulator matrix M_t .

Through parallelization of the method, the complexity of the algorithm in terms of size is reduced, achieving a result in lower time. The sequential time complexity of the Hough Transform without decomposition is $O(N^2m)$ [12]. The sequential algorithm

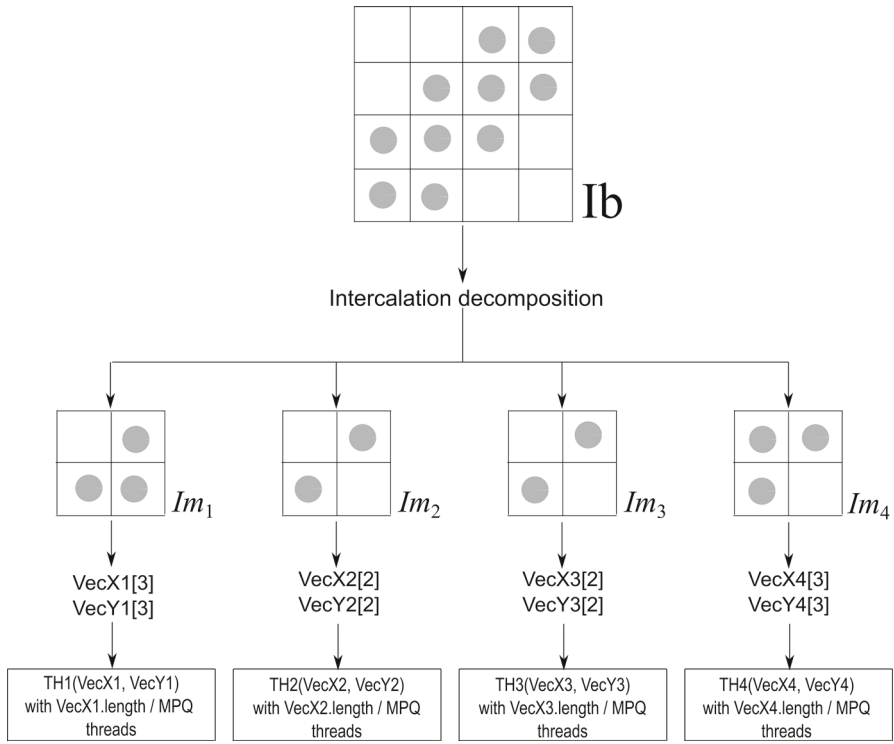


Fig. 6 The image I_b is decomposed using the intercalation method generating four subimages Im_1 , Im_2 , Im_3 and Im_4 . Vectors are generated to a size based on the number of points that each subimage contains. Following this, a *kernel* is run, based on the corresponding number of threads given by Eq. 4

Pseudocode 1: Pseudocode of the proposed algorithm

```

Input : The initial image
Output: The graphic of the input image with the straight lines detected in the image
1 begin
2    $[I, N] \leftarrow \text{InputImage}()$ ;
3    $[I_b] \leftarrow \text{CannyFilter}(I)$ ;
4    $[Im_1, Im_2, Im_3, Im_4] \leftarrow \text{kernel\_decompose} \lll 2N \text{ threads} \ggg (I_b)$ ;
5    $[\text{VecX}_1, \text{VecY}_1, \text{VecX}_2, \text{VecY}_2, \text{VecX}_3, \text{VecY}_3, \text{VecX}_4, \text{VecY}_4] \leftarrow \text{getPoints}(Im_1, Im_2, Im_3, Im_4)$ ;
6    $[M_1] \leftarrow \text{kernel\_TH1} \lll \text{VecX}_1.\text{length}/MPQ \text{ threads, stream1} \ggg (\text{VecX}_1, \text{VecY}_1)$ ;
7    $[M_2] \leftarrow \text{kernel\_TH2} \lll \text{VecX}_2.\text{length}/MPQ \text{ threads, stream2} \ggg (\text{VecX}_2, \text{VecY}_2)$ ;
8    $[M_3] \leftarrow \text{kernel\_TH3} \lll \text{VecX}_3.\text{length}/MPQ \text{ threads, stream3} \ggg (\text{VecX}_3, \text{VecY}_3)$ ;
9    $[M_4] \leftarrow \text{kernel\_TH4} \lll \text{VecX}_4.\text{length}/MPQ \text{ threads, stream4} \ggg (\text{VecX}_4, \text{VecY}_4)$ ;
10   $[M_t] \leftarrow \text{kernel\_addMs} \lll \rho \times \theta \text{ threads} \ggg (M_1, M_2, M_3, M_4)$ ;
11   $[P] \leftarrow \text{hough\_peaks}(M_t, N)$ ;
12   $[I_f] \leftarrow \text{hough\_lines}(I, P)$ ;
13 end

```

traverses all the $N \times N$ positions of the matrix (each position corresponds to a pixel of the input image). For each pixel with value higher than zero, the algorithm estimates the value m which is the resolution of the parameter θ . Using our proposed optimization, the value of $m = 180$ and $MPQ = 1$. The variable MPQ determines the number of points to be processed by thread; i.e., the smaller the value of MPQ , the lower the time complexity. Therefore, the time complexity of our voting phase is a theoretical order of $O(1)$. This is consistent with the fact that the variable MPQ preserves the level of parallelism of the main part of the method. Experimentally, the execution time increases slightly, since there exist certain operations that are not under the control of our algorithm. Such situations include for example the time it takes to copy data onto memory, the writing time and delays for synchronization of several processors [14]. Therefore, the optimum value of MPQ will be given in accordance with the characteristics of the GPU that is being used.

5 Experimental results

Now, we present the results obtained from experiments on a designed image of $N \times N$ pixels, and two real images with different values for $N = \{128, 256, 512, 1024, 2048, 4096\}$. These experiments were conducted on a computer with an Intel Xeon processor with 4 cores, 8 GB of RAM, an NVIDIA Quadro K4000 card with 768 CUDA cores. The parallel programming platform on GPU CUDA version 7 was used, with the debug tools from Visual Studio 10.0 with a plugin for CUDA. Matlab software was used to generate binarized images and CUDA-C for data processing.

The three algorithms were implemented. First, the sequential HT with no segmentation. Next, the standard method consisting in dividing the image in four quadrants (segments), this method also performs the optimization. Finally, we implement the optimization method proposed in [17] referred as division by intercalation. The sequential HT was implemented in C; the second and the third algorithms were implemented in CUDA-C. We also programmed the methods in Matlab, to conduct the pertinent comparisons. A comparison of our parallel implementation using GPU against the parallelization of [17] is presented below.

Figure 9 is designed to compare the performance of our optimized method using both decomposition methods, since the intercalation method using CPU works better on images where part of the image is empty or almost empty. In Tables 1, 2, and 3, we present the results of the times for the four implementations using the previous figures. In the case of the parallel implementations, the best times are obtained through the $MPQ = 1$ variable.

Note that the parallel intercalated and segmented methods are very similar in its execution time, and both present better results when compared to the sequential methods. In [17], the times of the parallel intercalated method are better than those of the segmented method, since this method only uses four threads. Additionally, we can compute the throughput of our parallel algorithm using the speedup when comparing with the best sequential algorithm. The speedup is obtained through Eq. 5, where $S_p(n)$ is the speedup of the parallel algorithm for an input size of n , $T(n)$ is the time it takes the best known sequential algorithm to resolve a problem of size n , $T_p(n)$ and

Table 1 Experimental results using Fig. 7

Dimension in pixels	Intercalated parallel in seconds	Segmented parallel in seconds	Sequential C in seconds	Sequential Matlab in seconds	Intercalated parallel Matlab (four cores) in seconds	Segmented parallel Matlab (four cores) in seconds
128 × 128	0.008	0.012	0.027	0.029	0.151	0.154
256 × 256	0.010	0.013	0.073	0.071	0.181	0.199
512 × 512	0.020	0.021	0.216	0.196	0.251	0.292
1024 × 1024	0.051	0.053	0.980	0.776	0.564	0.739
2048 × 2048	0.155	0.159	2.541	2.229	1.440	2.056
4096 × 4096	0.524	0.533	5.812	5.628	3.984	4.816

Table 2 Experimental results using Fig. 8

Dimension in pixels	Intercalated parallel in seconds	Segmented parallel in seconds	Sequential C in seconds	Sequential Matlab in seconds	Intercalated parallel Matlab (four cores) in seconds	Segmented parallel Matlab (four cores) in seconds
128 × 128	0.007	0.009	0.045	0.036	0.298	0.279
256 × 256	0.01	0.014	0.16	0.127	0.335	0.334
512 × 512	0.022	0.023	0.52	0.402	0.456	0.495
1024 × 1024	0.051	0.053	1.557	1.191	0.842	1.057
2048 × 2048	0.169	0.165	3.704	3.204	2.228	2.858
4096 × 4096	0.566	0.548	8.271	7.973	5.984	7.011

Table 3 Experimental results using Fig. 9

Dimension in pixels	Intercalated parallel in seconds	Segmented parallel in seconds	Sequential C in seconds	Sequential Matlab in seconds	Intercalated parallel Matlab (four cores) in seconds	Segmented parallel Matlab (four cores) in seconds
128 × 128	0.010	0.010	0.026	0.020	0.266	0.290
256 × 256	0.013	0.012	0.087	0.070	0.318	0.308
512 × 512	0.025	0.022	0.314	0.244	0.436	0.438
1024 × 1024	0.047	0.053	1.285	0.984	0.771	0.835
2048 × 2048	0.184	0.168	5.078	4.374	2.285	2.559
4096 × 4096	0.797	0.697	20.290	18.234	9.186	10.137



Fig. 7 Image 1 used in the experiments. Image taken from [7]



Fig. 8 Image 2 used in the experiments

is the time it takes the parallel algorithm with p processors to resolve a problem of size n [19]. In Fig. 10, the speedup of the division by segmentation and division by intercalation methods are presented, when image of Fig. 9 is used as input.

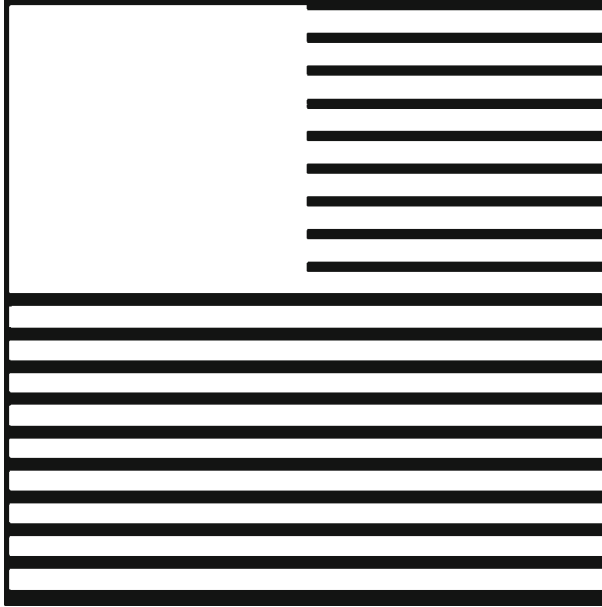


Fig. 9 Image 3 used in the experiments

$$S_p(n) = \frac{T(n)}{T_p(n)}. \quad (5)$$

The Performance Ratio (PR) is a measure that compares the speed using the total time taken by CPU for a specific input data (computational task) using a constant number of processors with the total time taken by GPU performing the same input data using a constant number of threads. The PR is defined as follows:

$$\text{PR}(n) = \frac{T(n, \#\text{processors})_{\text{CPU}}}{T(n, \#\text{threads})_{\text{GPU}}}. \quad (6)$$

Figure 11 shows the PR relating to the parallel methods implementation using CPUs as proposed in [17] when Fig. 9 is the input. Note that the parallel versions on GPU achieve speeds gain to 10 times better than the parallel implementations with CPUs.

In [17], they obtained a better performance using the decomposition method known as *decimation technique* (intercalation), than the traditional method of segment decomposition. In contrast, in our implementation with GPU we note that the times of both decomposition methods are very similar. An experiment was carried out using Fig. 7 (Image taken from [7]) with a dimension of 1024×1024 pixels, assigning different numbers of job threads. The results suggest that for few job threads, the intercalated method obtains better times.

We use the NVIDIA Visual Profiler (nvvp) to perform the analysis of the use of bandwidth of the *kernels* when it uses both decomposition methods (segmentation and intercalated) with $\text{MPQ} = 1$ and $\text{MPQ} = 20$. Note in Fig. 12a, b that the performance

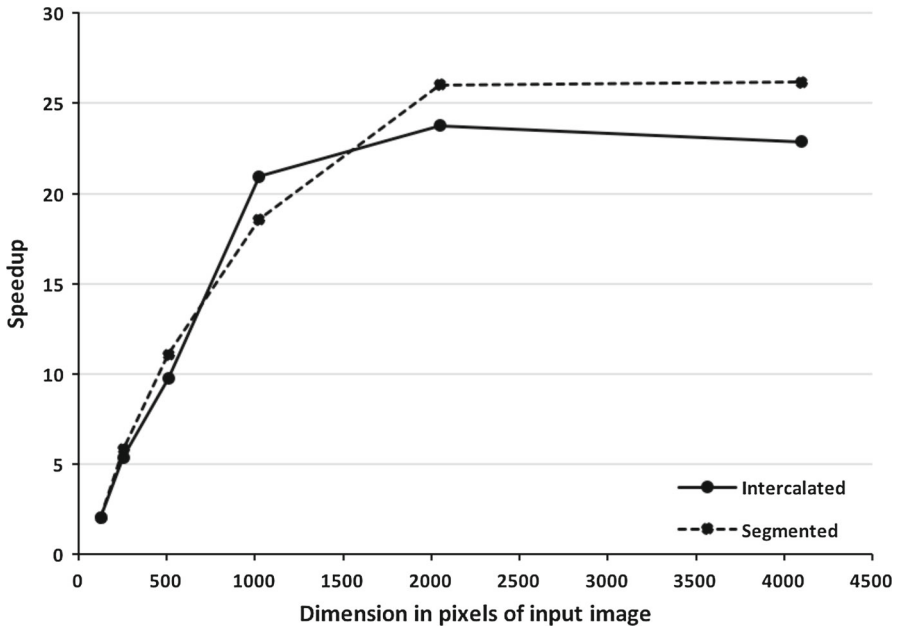


Fig. 10 Performance of the Hough Transform with implementation of the methods of decomposition by intercalation and segmentation in Fig. 9. As can be observed, both methods are up to 20 times faster than the sequential implementation

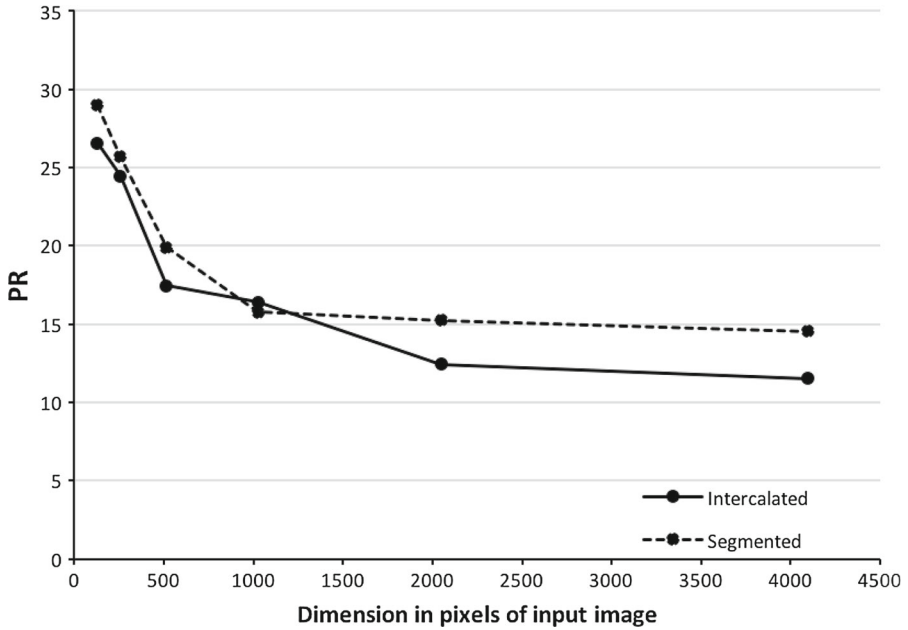


Fig. 11 PR of the GPU methods in relation to the methods propose in [17] using four CPUs

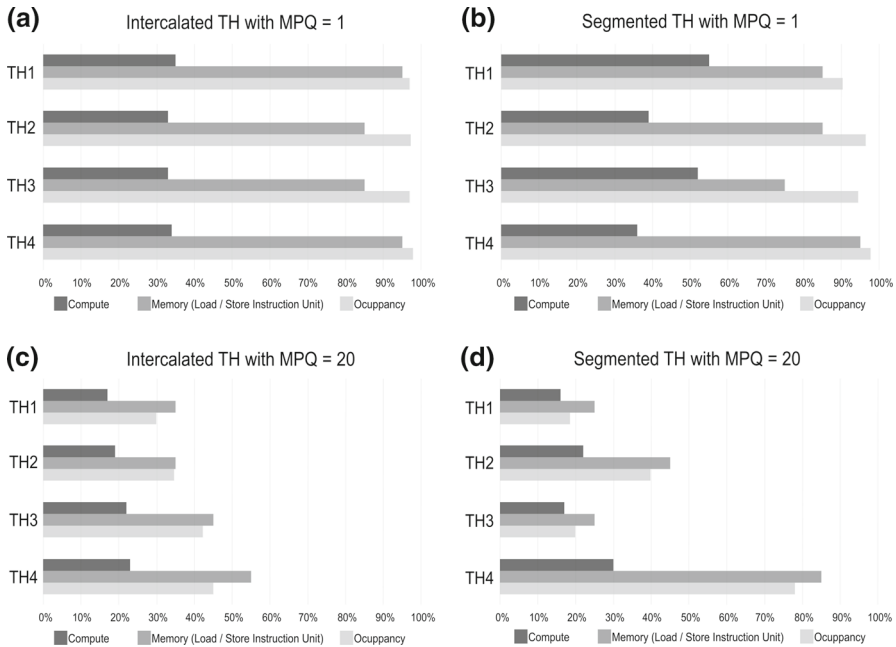


Fig. 12 Comparison of the profile of using the compute, the bandwidth memory and the occupancy of the *kernels* kernel_HT1 (HT1), kernel_HT2 (HT2), kernel_HT3 (HT3) and kernel_HT4 (HT4) of the Fig. 7, using both decomposition methods (segmentation and intercalated) with MPQ = 1 and MPQ = 20

of *kernels* kernel_HT1, kernel_HT2, kernel_HT3 and kernel_HT4 is restricted by the bandwidth of the GPU card of the experiment (Quadro K 4000). Therefore, the *kernel* is using the maximum bandwidth and the warps, maintaining a 100% of occupancy. When MPQ = 20, note in Fig. 12b, c that it is possible to increase the number of threads in execution. Thus, when MPQ = 20, there exists problems with the latency. Therefore, it is possible to define a tradeoff (relation) between the use of bandwidth and the dimension of the input image, when we consider the features of the GPU card.

When the thread number increases, the execution time decreases and these execution times are similar on both implementations. Table 4 lists the results.

Decomposition by intercalation method is recommended when the available architecture possess low processing power, and then, it is able to create homogeneous workloads in the job threads, allowing results to be obtained in less time in comparison with the decomposition by segments method. However, when there is a higher level of computing power available, both implementations generate much better times than those obtained with sequential methods, depending on the number of points to be dealt with by each thread, which is defined through the MPQ variable.

6 Concluding remarks

In this paper, an optimization method using parallel computing on a GPU for the Hough Transform algorithm for straight lines recognition in an image is presented.

Table 4 Experimental results with different numbers of job threads using Fig. 9 with a dimension of 1024×1024

Job threads per image	Intercalated time (s)	Segmented time (s)
4	2.248	4.894
8	2.13	2.977
16	0.604	2.038
24	0.42	0.744
40	0.272	0.461
200	0.119	0.143
400	0.076	0.105
4000	0.068	0.068
8000	0.06	0.055
16000	0.061	0.055
36000	0.06	0.054

The optimization is carried out via a method of decomposition into subimages known as decimation technique presented in [17] and also with the traditional method of division by segments. Both methods provide good results as long as the number of job threads is large. However, in the absence of a GPU with high level of processing, the best option would be the method of division by intercalation, since it provides the job threads with homogenous workloads, achieving better times. Implementation was done on an NVIDIA GPU through the parallel programming platform CUDA-C to maximize all the processing potential of the graphics card. The simulations show that both methods (intercalation and segmentation-based) achieve better response times on a GPU than the non-decomposition sequential method as well as the implementation using CPUs [17], reaching speeds up to 20 times better than those of the sequential method.

References

1. Duda RO, Hart PE (1975) Use of the Hough transformation to detect lines and curves in pictures. *Commun ACM* 15(1):11–15
2. Satzoda RK, Suchitra S, Srikanthan T (2008) Parallelizing the Hough transform computation. *IEEE Signal Process Lett* 15:297–300
3. Ito Y, Ogawa K, Nakano K (2011) Fast ellipse detection algorithm using Hough transform on the GPU. In: *Proceedings of the Second International Conference on Networking and Computing (ICNC)*, pp 313–319
4. Mukhopadhyay P, Chaudhuri BB (2015) A survey of Hough transform. *Pattern Recognit* 48(3):993–1010
5. Parker J (2011) *Algorithms for image processing and computer vision*, 2nd edn. Wiley, London
6. Xu Z, Shin B-S, Klette R (2015) Closed form line-segment extraction using the Hough transform. *Pattern Recognit* 48:4012–4023
7. Gonzalez RC, Woods RE (2008) *Digital image processing*, 3rd edn. Pearson, Upper Saddle River
8. Ji J, Chen G, Sun L (2011) A novel Hough transform method for line detection by enhancing accumulator array. *Pattern Recogn Lett* 32(11):1503–1510
9. Atiqzaman M (1992) Multiresolution Hough transform—an efficient method of detecting patterns in images. *IEEE Trans Pattern Anal* 14(11):1090–1095

10. Vladimir T, Jeon D, Kim DH (2013) Hough transform with Kalman filter on GPU for real-time line tracking. In: Proceedings of the Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), pp 212–216
11. Chen ZH, Su AWY, Sun MT (2012) Resource-efficient FPGA architecture and implementation of Hough transform. *IEEE Trans Very Large Scale Integr Syst* 20:1419–1428
12. Chen L, Chen H, Pan YI, Chen Y (2004) A fast efficient parallel Hough transform algorithm on LARPBS*. *J Supercomput* 29:185–195
13. Braul T, Feyrer S, Rapf W, Reinhardt M (2000) *Parallel image processing*. Springer, New York
14. Nvidia C (2015) *CUDA C programming guide*. NVIDIA Corporation. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Accessed 30 Nov 2016
15. Cook S (2012) *CUDA programming: a developer's guide to parallel computing with GPUs*. Morgan Kaufmann, Los Altos
16. Harris H (2015) GPU Pro Tip: CUDA 7 streams simplify concurrency. NVIDIA Corporation. <http://devblogs.nvidia.com/paralleforall/gpu-pro-tip-cuda-7-streams-simplify-concurrency/>. Accessed 30 Nov 2016
17. Arceo C, Lopez-Martinez JL, Narvaez-Diaz L (2015) Fast algorithm of the Hough transform to straight lines detection in an image. *Program Mat Softw* 7(2):8–13
18. Canny J (1986) A computational approach to edge detection. *IEEE Trans Pattern Anal Mach Intell* 8:679–698
19. JaJa J (1992) *An introduction to parallel algorithms*. Addison-Wesley Publishing Company, Reading