



Parallel border tracking in binary images using GPUs

Victor M. Garcia-Molla¹ · Pedro Alonso-Jordá¹ · Ricardo García-Laguía¹

Accepted: 21 December 2021
© The Author(s) 2022

Abstract

Border tracking in binary images is an important kernel for many applications. There are very efficient sequential algorithms, most notably, the algorithm proposed by Suzuki et al., which has been implemented for CPUs in well-known libraries. However, under some circumstances, it would be advantageous to perform the border tracking in GPUs as efficiently as possible. In this paper, we propose a parallel version of the Suzuki algorithm that is designed to be executed in GPUs and implemented in CUDA. The proposed algorithm is based on splitting the image into small rectangles. Then, a thread is launched for each rectangle, which tracks the borders in its associated rectangle. The final step is to perform the connection of the borders belonging to several rectangles. The parallel algorithm has been compared with a state-of-the-art sequential CPU version, using two different CPUs and two different GPUs for the evaluation. The computing times obtained show that in these experiments with the GPUs and CPUs that we had available, the proposed parallel algorithm running in the fastest GPU is more than 10 times faster than the sequential CPU routine running in the fastest CPU.

Keywords Border tracking · Parallel computing · GPU computing

1 Introduction

Finding borders in a 2d binary image (where all of the pixels are either 0 or 1) is an important tool for many applications of image processing, e.g., segmentation in medical applications [1, 2], automatic recognition of handwriting [3, 4], and many other applications, including applications with real-time requirements. It can be used

✉ Victor M. Garcia-Molla
vmgarcia@dsic.upv.es
Pedro Alonso-Jordá
palonso@upv.es
Ricardo García-Laguía
rgletters@gmail.com

¹ Universitat Politècnica de València, Valencia, Spain

to find borders in color images, in grayscale images [5], or in images with uncertainties [6] (by applying appropriate thresholds to the image).

There are many algorithms in the literature for border tracking. One of the most popular is the algorithm proposed in [7], which is commonly known as the Suzuki algorithm. This algorithm has been implemented in the *findcontours* function, which is part of the well-known library for computer vision OpenCV [5].

The main idea of the Suzuki algorithm is to loop over all of the pixels in the image looking for pixels belonging to a (previously unexplored) border. When such a “border” pixel is found, the Suzuki algorithm provides a mechanism to follow this border until it has been fully tracked.

Since the Suzuki algorithm follows the borders sequentially and has a strong sequential nature, it is not easy to parallelize. This is usually not very important because the sequential implementations for CPU are quite fast. However, there are more and more applications where most of the computations related to images are carried out in GPUs, using CUDA [8] or related libraries. (There are several functions in OpenCV that are implemented for execution in GPUs.) There are many filtering operations over images that are extremely efficient when computed in GPUs. It would be quite useful to be able to perform some operations in CPUs and others in GPUs; however, at present, this idea is deterred by the large cost of memory transfers between CPUs and GPUs. In order to avoid the cost of extra memory transfers between CPUs and GPUs, it seems natural to look for GPU implementations of border tracking.

In this paper, we describe a parallel border tracking algorithm that is based on the Suzuki algorithm but which has important algorithmic differences. This algorithm was developed for a specific real-time industrial application, which is described in Sect. 2. However, we believe that the parallel algorithm proposed is an interesting general contribution that allows the borders of a binary image to be computed in GPUs.

We have tested our implementation with a number of images of different sizes; some come from the industrial application described in Sect. 2, and some are generated synthetically. The results were compared with the results obtained with the OpenCV CPU implementation of the Suzuki algorithm in the *findcontours* function. The borders obtained were the same in all of the cases. The CUDA implementation of the proposed parallel algorithm cannot take full advantage of the computing power of the GPU because there are parts of the algorithm that are intrinsically sequential. However, the computing times obtained by our algorithm are similar to or better than the computing times of the OpenCV *findcontours* routine when using CPUs and GPUs of similar price for the comparison.

The structure of the paper is as follows. First, we describe the industrial application that motivated the work along with the state-of-the-art methods. In Sect. 3, we describe the problem of border tracking for binary images and outline the original Suzuki method. In Sect. 4, we describe the proposed parallel algorithm, and, in Sect. 5, some GPU implementation details are discussed. Section 6 is devoted to the evaluation of the proposed algorithm, comparing it with the

OpenCV *findcontours* routine. Finally, the conclusions and possible future work are discussed in Sect. 7.

2 Motivation and state of the art

The work described in this paper has been driven by the need for a GPU implementation of border tracking in a real-time system for automatic detection of defects in car bodyworks. This system was developed in Autis S.L.¹, a company that works on the integration of industrial systems, with emphasis on computer vision applications. The system that was developed in Autis for detection of defects in car bodyworks has several light sources and several cameras that are controlled by a workstation. Each camera takes images of light reflections on a part of the bodywork. The images must be processed in real time in order to avoid delays on the manufacturing line. Each CPU core of the workstation controls a camera. The detection of the defects requires the images to undergo several processes; an important part of these processes is to obtain the borders on the images. As mentioned above, there are fast CPU routines that can carry out this process (e.g., the *findcontours* routine of the OpenCV library). However, in the Autis system, the CPU cores of the workstation are heavily loaded with other parts of the process.

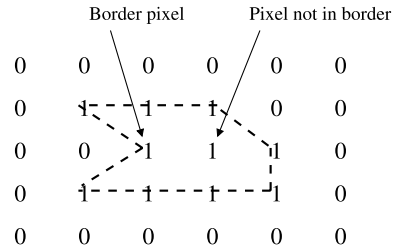
One way of relieving the load of the CPU cores was to move part of the image processing to a GPU. Some of the required operations (“erode” or “dilate” operations [5]) were very efficient when carried out on the GPU. In order to avoid the cost of extra memory transfers between the CPU and the GPU, we looked for GPU CUDA [8] implementations of border tracking. However, there was no software available for border detection on the GPU. Therefore, it was a natural idea to try to port the Suzuki algorithm to GPUs. Several algorithmic approaches have been investigated in Autis. One of these approaches led to the algorithm for border tracking described in this paper.

2.1 State of the art

There are many methods for border tracking. These include the simple boundary follower [9], the improved single boundary follower [10], the Moore neighbor tracing algorithm (MNT) [11], the radial sweep algorithm (RSA) [12], the method proposed in [13], the contour tracing method proposed in [14], etc. However, of the few implementations available, only two take into account the reliability required for an industrial application. The candidate routines were the above-mentioned *findcontours* routine and the *bwboundaries* routine, which is part of the image processing toolbox of MATLAB [15]. The *bwboundaries* routine is based on the Moore neighbor tracing algorithm described in [11]; it is very similar to the Suzuki algorithm and has similar performance.

¹ <https://autis.es/en/>

Fig. 1 Example of a border pixel (with a zero-valued neighbor pixel in a *cross* position) and an example of a pixel with a zero-valued neighbor pixel that is not on the border (the zero-valued neighbor pixel is not in a *cross* position)



The situation is more complicated in the case of parallel implementations of border tracking for binary images. The work in [16] presents a theoretical description of a model for parallel updating of a single border. Despite having a similar name, the problem studied is very different from the problem of finding all of the borders in a binary image. There are several parallel border tracking implementations for multi-valued (not binary) images, such as those described in [17–19]. In those cases, the parallelism is extracted by detecting borders at different values (or levels). This is not a feasible procedure for parallelizing the detection of borders in binary images.

3 Definition of the problem

In this work, we are only interested in obtaining a parallel algorithm with the same functionality as the Suzuki algorithm. Throughout this paper, we mostly use terminology from the Suzuki paper. As in this original work by Suzuki et al. [7], we only consider rectangular binary images.

In this work, we consider only the case of 8-connectivity, that is, the pixel (i, j) is neighbor (is connected) to every pixel that touches one of its edges or corners [20]. If 8-connectivity is considered, it can be easily shown that the pixel with coordinates (i, j) is part of a border if its value is larger than 0 and if there is a pixel with a value of 0 in at least one of these positions: $(i + 1, j)$, or $(i - 1, j)$, or $(i, j - 1)$, or $(i, j + 1)$. An example is shown in Fig. 1. The pixel labeled as “Border Pixel” has a zero-valued neighbor pixel in the position $(i, j - 1)$, and, therefore, it is part of a border, which is shown by the discontinuous line. However, the pixel labeled as “Pixel not on Border” is not a border pixel. It has a zero-valued neighbor pixel, but it is not in one of the four positions mentioned. The zero is the position $(i - 1, j + 1)$ relative to the pixel considered. Please note that since 8-connectivity is being considered, the border follows the discontinuous line without touching that pixel. These four positions $(i + 1, j)$, $(i - 1, j)$, $(i, j - 1)$, $(i, j + 1)$ are very important in the following, and we will name them as the four *cross* positions that are relative to the pixel with coordinates (i, j) .

The goal of border tracking (or border following) in 2d binary images is to obtain the borders (sequences of nonzero pixels separating zones filled with pixels larger than zero, from zones filled with zeros). We assume that the “frame” of the image (the first and last rows, and the first and last columns) is filled with zeros. This usually implies that the background of the image is filled with zeros. There are two

possible types of borders. The first type are outer borders, which are sets of nonzero pixels between a zone that is filled with pixels that are larger than zero and a zone that is filled with zeros (when the zone filled with zeros surrounds the zone filled with pixels larger than zero). The second type are hole borders, which are sets of nonzero pixels between a zone that is filled with pixels that are larger than zero and a zone that is filled with zeros (when the zone filled with pixels larger than zero surrounds the zone filled with zeros).

The sequential Suzuki algorithm described in [7] examines all of the pixels in the input image, using a standard double loop. The origin of coordinates is the top left corner of the image. The row index increases when going toward the bottom part of the image, while the column index increases when going toward the right part of the image. For the description in this work, we have chosen vectors and matrices with numeration starting with 1, that is, the top left pixel of the image is the pixel (1, 1).

When the Suzuki sequential algorithm finds a border pixel P , the tracking starts by searching for its “former pixel” by rotating clockwise around the pixel P and then searching for the “next” pixel by rotating counterclockwise around the pixel P . Then, the new “next” pixel is added to the border and now becomes the center pixel, which is used as above to find a new “next” pixel. This procedure follows the border until it gets back to the initial pixel (i.e., the border is followed until it is closed). This way of following the border is clearly sequential and is difficult to parallelize.

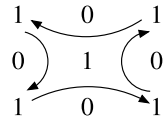
4 The proposed parallel algorithm

The main idea behind the proposed algorithm is to split the image into small rectangles of the same size (as much as possible). Then, a process is launched for each rectangle, which tracks and stores all of the borders in its rectangle, similarly to the Suzuki algorithm. Clearly, there will be borders that belong to more than one rectangle. Then, the next step will be to connect the borders from the different rectangles. Our complete proposal has three main steps: preprocessing (4.1), border tracking in rectangles (4.2), and connection of the borders of all of the rectangles (4.3).

4.1 Preprocessing

As a first step, we want to determine which pixels are part of at least one border. The pixel with coordinates (i, j) is part of a border if its value is greater than 0 and if there is a pixel with value 0 in any of the cross positions. This check is performed easily and efficiently in the GPU. In our CUDA implementation, we wrote a kernel called *border_preprocessing*. This kernel is launched with as many threads as possible and needs very little computing time. The result of this check is stored in an array of the same size as the image. We call this array “*is_border*,” so *is_border*(i, j) is equal to 1 if the pixel (i, j) is in a border, and 0 otherwise.

Fig. 2 Center pixel belonging to four different borders



4.2 Border tracking in rectangles

As mentioned above, the key idea in our algorithm is to divide the image into small rectangles. Within each rectangle, the tracking must be done sequentially; therefore, we use a single thread for tracking the borders in each rectangle.

The division of the image must be made using powers of two as the number of divisions in each axis in order to carry out the connection phase efficiently. In the current version, we are using divisions of 32×32 and 64×64 rectangles, although any other product of powers of two could be used. The thread responsible for a rectangle will examine all of the pixels in the rectangle with a standard double loop over rows and columns checking whether or not the current pixel belongs to a border.

When the thread finds a border pixel that has not been followed previously, the associated border must be followed and stored. In the sequential version of the Suzuki algorithm, all of the borders are “closed” (due to the “frame” being filled with zeros), that is, all of the borders are fully included in the image.

However, in the parallel version, a given border may be fully contained in a rectangle (in this case, we say that this border is “closed”), or it may be distributed in several rectangles, passing through the limits of the rectangles. Each one of these pieces of a border is called an “open” border, which enters and leaves the rectangle. In order to obtain a full connection later, all of the borders in a rectangle (closed or open) must be tracked and stored.

Furthermore, a pixel may belong to several borders. This is especially troublesome when the pixel in the border has several neighbors that are zero in cross positions. Potentially, there may be as many different borders passing through a pixel as neighbors with zero value in cross positions. In Fig. 2, the pixel in the center can belong to up to four different borders. These borders are obtained as in the sequential version of the Suzuki algorithm, and, in the example, each zero in a cross position gives rise to a different border. For example, the left border in Fig. 2 is obtained by selecting the zero in cross position (2, 1), then rotating around the center pixel clockwise starting from the zero in (2, 1) to find the “former” pixel (1, 1), and then rotating counterclockwise starting from the zero in (2, 1) to find the “next” pixel (3, 1). The other three borders are similarly obtained with rotations with the center in the current pixel and starting from the other three zeros in cross positions. It is important to clarify that these counterclockwise and clockwise rotations are local and relative to the center pixel.

It is clear that a pixel may belong to different borders; however, each ordered triad composed of the center pixel, the former pixel, and the next pixel can only belong to one border. Note that the same triad with reversed order may belong to a different border; therefore, the ordering of the triad is important. In the following, we will use

the term “triad” meaning “ordered triad.” We have devised a procedure for tracking and storing borders as a sequence of triads. The whole procedure is shown in the form of pseudocode in Algorithm 1. First, we show the full algorithm, and then we proceed to describe it step by step.

Algorithm 1 Procedure Track_borders_in_rectangle

```

1: Input:  $i\_ini, i\_end, j\_ini, j\_end$ 
2: for  $j = j\_ini$  to  $j\_end$  do
3:   for  $i = i\_ini$  to  $i\_end$  do
4:     if  $is\_border(i, j) == 1$  then
5:       Find the first triad  $pix\_form, pix\_curr, pix\_next$ , obtain its triad number
6:       if  $P(i, j) \% triad\_number != 0$  then
7:          $P(i, j) = P(i, j) * triad\_number$ 
8:         Call  $tracking\_forward\_backward(pix\_form, pix\_curr, pix\_next)$ 
9:       end if
10:      Finished=False
11:      while Finished==False do
12:         $pix\_next = pix\_form$ 
13:        Find a new triad (rotating clockwise, starting from  $pix\_next$ ) and compute
        triad number
14:        if new triad==first triad then
15:          Finished=True
16:        else if  $P(i, j) \% triad\_number != 0$  then
17:           $P(i, j) = P(i, j) * triad\_number$ 
18:          Call  $tracking\_forward\_backward(pix\_form, pix\_curr, pix\_next)$ 
19:        end if
20:      end while
21:    end if
22:  end for
23: end for

```

Fig. 3 Clockwise rotation to obtain former pixel

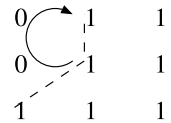
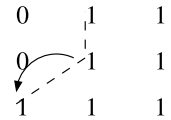


Fig. 4 Counterclockwise rotation to obtain the next pixel



Lines 2 and 3 in Algorithm 1 are the double loop for examining all of the pixels in the rectangle. Only the pixels belonging to a border (using the array *is_border* obtained in the preprocessing) are processed. The procedure for finding a first triad passing through a border pixel $P(i, j)$ is quite similar to the procedure of the sequential Suzuki algorithm, which is as follows:

1. Locate a zero in any of the cross positions (there must be at least one for $P(i, j)$ to be a border pixel). We have arbitrarily chosen to start from the cross position $(i, j - 1)$ (we could start from any cross position) and to search clockwise looking for a zero in one of these four positions. In the example in Fig. 3, there is only a 0 in position $(i, j - 1)$; therefore, this zero pixel is selected.
2. Find the “former” pixel. The former pixel in the border (relative to the current border pixel $P(i, j)$) can be determined by rotating clockwise around the center pixel (starting from the zero selected in Step 1) until a nonzero pixel is found. In the example in Fig. 3, the selected zero is in position $(i, j - 1)$. In the example, by rotating clockwise from position $(i, j - 1)$ around the center pixel $P(i, j)$, the former pixel is found in position $(i - 1, j)$.
3. Find the “next” pixel. The next pixel in the border (relative to the current border pixel $P(i, j)$) can be found by rotating counterclockwise (starting from the zero selected in Step 1) until a nonzero pixel is found. In the example, the next pixel is the pixel in position $(i + 1, j - 1)$ (Fig. 4).

After Steps 1, 2, and 3, we have obtained an “ordered” triad: former pixel, current (or center) pixel, and next pixel, which can be the starting triad of the border. However, before tracking this border, we must determine whether or not this border has already been tracked and stored. To do that, we have devised a labeling procedure.

4.2.1 Avoid tracking borders already tracked: Labeling

It is important to devise a mechanism to avoid threads that are responsible for a rectangle tracking a border that may already have been tracked.

The devised method for detecting that a border has already been tracked is based on the following encoding procedure. The first four prime numbers greater than one (2, 3, 5, 7) have been assigned to each one of the cross positions as follows: Position

Fig. 5 Numbers (labels) assigned to zeros in the cross positions

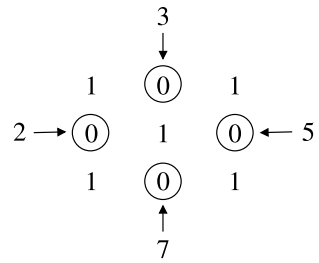


Fig. 6 Example, single zero in a cross position

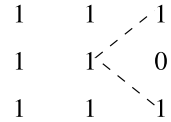
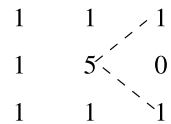


Fig. 7 Modification of the example in Fig. 6 after tracking the border passing through the center pixel



$(i, j - 1)$ is assigned the value 2, position $(i - 1, j)$ is assigned the value 3, position $(i, j + 1)$ is assigned the value 5, and position $(i + 1, j)$ is assigned the value 7. See the example in Fig. 5.

Let us consider the center pixel, the $P(i, j)$ pixel. Since we are dealing with binary images, initially the value of all nonzero pixels is 1. For each triad, we want to compute a positive integer value that identifies the zeros visited when obtaining a triad; therefore, this number (called the “triad number”) identifies the triad. The triad number is obtained as the product of the labels of the zeros visited during the computation of the triad.

Then, the value of the center pixel $P(i, j)$ is modified as the product of the triad numbers, for all of the triads passing through $P(i, j)$. For example, in Fig. 5, the triad formed by $pix_form(i - 1, j - 1)$, $pix_curr(i, j)$, and $pix_next(i + 1, j - 1)$ starts with the zero labeled as 2 and does not pass through more zeros. Therefore, its triad number will be 2. Similarly, the triad formed by $pix_form(i - 1, j + 1)$, $pix_curr(i, j)$, and $pix_next(i - 1, j - 1)$ starts with the zero labeled as 3, so its triad number is 3. Similarly, we can find the triad number of the triads associated with zero pixels labeled as 5 or 7.

Once a given triad (not yet followed) is obtained, the value of the center pixel is multiplied by the triad number. As an example, given the situation shown in Fig. 6, when tracking the borders passing through the center pixel, the only triad passing through this pixel is $pix_form(i + 1, j + 1)$, $pix_curr(i, j)$, $pix_next(i - 1, j + 1)$, and its label is “5.” Therefore, when tracking, the value of the center pixel is multiplied by 5, obtaining the situation shown in Fig. 7.

There may also be triads that are associated with two or more cross zeros. In this case, the label of the triad will be the product of the labels of these zeros.

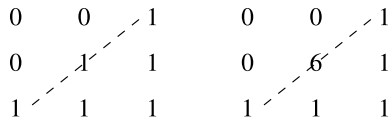
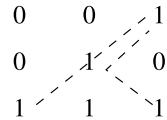


Fig. 8 Left: triad passing through 2 zeros in cross positions. Right: modification after tracking the border passing through the center pixel

Fig. 9 Pixel that is part of two triads



Consider Fig. 8. In this example, the only counterclockwise triad would be $\text{pix_form}(i-1, j+1)$, $\text{pix_curr}(i, j)$, $\text{pix_next}(i+1, j-1)$, which starts with the cross zero with label “2” and passes through the cross zero with label “3.” Therefore, the label of the triad would be $2 \cdot 3 = 6$. Since there are no more triads, the center pixel must take the value 6.

There may also be several triads associated with the same center pixel. Let us consider the rectangle in Fig. 9. In this case, we obtain two triads: one with triad number 6 (as in Fig. 8), and the other that visits only the cross zero in position 5. Therefore, the number of the second triad is 5 and the center pixel will have the value $6 \cdot 5 = 30$.

The maximum value of any pixel would be the situation shown in Fig. 5; the center pixel has the value $2 \cdot 3 \cdot 5 \cdot 7 = 210$, which is smaller than 256. This is an interesting detail because many images are stored with 8-bit integers.

The procedure for deciding whether or not a triad must be followed now depends on the remainder of the integer division $P(i, j)/\text{triad_number}$. If the remainder of this division is zero, then this triad has already been tracked. If the remainder is not zero, then the border has not been tracked. It must be tracked, and the value of the pixel $P(i, j)$ must be updated by multiplying it by the triad number.

The tracking of the border is described below.

4.2.2 Tracking of borders after obtaining a starting triad that is not yet tracked

The complete procedure is given as Algorithm 2 and is used in lines 8 and 18 of Algorithm 1. The tracking of a border starting with a given triad may have two phases. First, the tracking proceeds forward, which is similar to the sequential Suzuki algorithm (*while* starting in line 5 in Algorithm 2). The new pixel is obtained by rotating counterclockwise around the “next” pixel. If the border is fully contained in the rectangle, then the border will be fully tracked and then stored. However, if this is an open border, it will leave the rectangle at some

stage. In this case, the border may have a previous part that must be tracked backward. The backward part starts in line 11 and is quite similar to the forward part, with the only change being that now the rotations are clockwise.

Algorithm 2 Procedure Tracking_forward_backward

```

1: Input:  $pix\_form$ ,  $pix\_curr$ ,  $pix\_next$ 

2:  $pix\_curr\_ini=pix\_curr$ 

3:  $pix\_next\_ini=pix\_next$ 

4:  $pix\_form\_ini=pix\_form$ 

5: while  $pix\_next$  in current rectangle, and  $pix\_next \neq pix\_curr\_ini$  do

6:   store triad ( $pix\_form$ ,  $pix\_curr$ ,  $pix\_sig$ )

7:    $pix\_form=pix\_curr$ 

8:    $pix\_curr=pix\_next$ 

9:   Search next  $pix\_next$  rotating counterclockwise around  $pix\_curr$ , starting the rotation
   from  $pix\_form$  ; update value of  $pix\_curr$ 

10: end while

11: if  $pix\_next$  out of rectangle then

12:    $pix\_form=pix\_form\_ini$ ,  $pix\_next=pix\_next\_ini$ ;

13:    $pix\_curr=pix\_curr\_ini$ 

14:   while  $pix\_form$  in current rectangle do

15:     store triad ( $pix\_form$ ,  $pix\_curr$ ,  $pix\_sig$ )

16:      $pix\_next=pix\_curr$ 

17:      $pix\_curr=pix\_form$ 

18:     Search next  $pix\_form$  rotating clockwise around  $pix\_curr$ , starting the rotation
     from  $pix\_next$  ; update value of  $pix\_curr$ 

19:   end while

20: end if

```

The updating of the pixel value in lines 9 and 18 of Algorithm 2 is carried out according to the labeling procedure described above. It is necessary to ensure that this border is not tracked again. The updating is shown as Algorithm 3.

Algorithm 3 Updating of pixel value

```

1: if when rotating, a zero in position  $(i, j - 1)$  is visited then
2:    $pix\_curr.value = pix\_curr.value * 2$ 
3: end if
4: if when rotating, a zero in position  $(i - 1, j)$  is visited then
5:    $pix\_curr.value = pix\_curr.value * 3$ 
6: end if
7: if when rotating, a zero in position  $(i, j + 1)$  is visited then
8:    $pix\_curr.value = pix\_curr.value * 5$ 
9: end if
10: if when rotating, a zero in position  $(i + 1, j)$  is visited then
11:    $pix\_curr.value = pix\_curr.value * 7$ 
12: end if

```

It is important to note that when a border in a rectangle is open it enters the rectangle through the first triad and leaves the rectangle through the last triad. The first triad has its former pixel outside of the rectangle, and the last triad also has its next pixel outside of the rectangle. This is used in the connection phase to ensure that the borders from different triangles are connected properly.

The final explanation needed for Algorithm 1 is the loop starting in line 11. The previous lines in Algorithm 1 process the first triad found. However, there may be more than one triad passing through a pixel (see, for example, Fig. 9). The *while* loop starting in line 11 searches and processes other triads that may pass through the actual pixel.

4.3 Connection of the borders of all of the rectangles

After the tracking stage, each rectangle will have generated a data structure where the borders are stored as sequences of triads. When the borders of all of the rectangles have been computed, the connection between the open borders from different rectangles can start. The key for the parallel connection algorithm is that the connections between borders in two neighbor rectangles can be established independently from any other connections between other rectangles. We describe the overall process first, and then we describe the process of connecting the borders of two rectangles in detail.

Fig. 10 Image divided into 4×8 rectangles

(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)
(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)
(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)
(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	(4,8)

Fig. 11 Image from Fig. 10 divided into 2×8 rectangles after the first stage of the vertical sweep

(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)
(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)

4.3.1 Overall connection process

In order to describe the overall procedure graphically, we will show the process for a simpler example, assuming a division of the input image in 4×8 rectangles, as shown in Fig. 10.

The connection of the borders of rectangle (1, 1) with the borders of its horizontal neighbor, the rectangle (1, 2), can be computed in parallel with the connection of the borders of any other pair of neighbor rectangles. This can also be done vertically. For example, the connection of borders of rectangle (1, 1) and rectangle (2, 1) can be done in parallel with the connection of borders of rectangle (1, 2) and rectangle (2, 2). There are many possible arrangements for a parallel connection. We have chosen to use numbers of rectangles in powers of two for ease of programming and to use two sweeps, first a vertical sweep and then a horizontal sweep. If the number of rectangles is $NX \times NY$, with NX and NY power of two, then the vertical sweep will have $\log_2(NX)$ stages and the horizontal sweep will have $\log_2(NY)$ stages.

In the first stage of the vertical sweep, $(NX/2) \times NY$ threads are launched. The borders of the rectangle $(2(i - 1) + 1, j)$ and the borders of the rectangle $(2i, j)$ will be connected by the thread (i, j) . This means that a new data structure is created where the closed borders from both rectangles are included, and the open borders that go from one of the two rectangles to the other are connected. By gathering all of these borders, a structure is obtained with all of the borders from the union of both rectangles. Of course, there may still be open borders. In our example, we would execute 2×8 threads in this first stage, each of which connects two rectangles. After all of them complete their tasks, we would obtain a similar data structure, but for an image divided in 2×8 rectangles. See Fig. 11.

The next vertical stage is similar; 1×8 threads are executed and the (i, j) thread connects the borders of rectangle $(2(i - 1) + 1, j)$ with the borders of rectangle $(2i, j)$. This stage creates a data structure corresponding to an image with 1×8 rectangles (See Fig. 12).

This ends the vertical sweep. The horizontal sweep is similar. In the first stage of the example, just four threads are needed, and the j th thread connects the borders

(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)
-------	-------	-------	-------	-------	-------	-------	-------

Fig. 12 Image from Fig. 10 divided into 1×8 rectangles after two stages of the vertical sweep

(1,1)	(1,2)	(1,3)	(1,4)
-------	-------	-------	-------

Fig. 13 Image from Fig. 10 divided into 1×4 rectangles after two stages of vertical sweep and one stage of the horizontal sweep

of rectangle $2(j - 1) + 1$ with the borders of rectangle $2j$. In the example, the result would be like the one shown in Fig. 13.

The process is repeated until the structure for a single final rectangle (the whole image) is obtained.

4.3.2 Connection of borders of two neighbor rectangles

The connection of the borders of two neighbor rectangles must be done sequentially. We consider that two rectangles are neighbors if they share a side. The procedure is the same whether they share a vertical or a horizontal side.

Each border is stored in a data structure that includes the sequence of triads forming the border, a pointer to the first triad (the beginning of the border), and a pointer to the last triad (the end of the border). The final goal is to obtain a vector of closed borders called *CONTG* (initially empty), which includes all of the borders in the whole image. The borders of the first rectangle are stored in a vector of *NC1* borders called *CONT1*, and the borders of the second rectangle are stored in a vector of *NC2* borders called *CONT2*. When a closed border is found in *CONT1* or in *CONT2*, it is copied to the global vector called *CONTG*. However, it must be noted that, as described above, there will be several simultaneous threads connecting the borders of neighbor rectangles. Therefore, the access to *CONTG* must always be done using the mutual exclusion *atomicAdd* directive to prevent two different threads from accessing the *CONTG* vector at the same time.

The open borders obtained after processing *CONT1* and *CONT2* will be stored in a new local output vector of borders that is called *CONTN*. Since some of the borders in *CONT1* or *CONT2* can collapse into a single border, it cannot be known in advance how many borders will be stored in the vector *CONTN*.

The idea is to start with a loop over the borders of *CONT1*, examine all of the connections of the ends of the borders of Rectangle 1, and, when finished with the borders of *CONT1*, loop over the borders of *CONT2*. Some of the borders of *CONT1* or *CONT2* may have been connected before the loop arrives to them. To avoid considering any border twice, we need a variable for each border in order to mark each border in *CONT1* and in *CONT2* as used or unused. Initially, all of the borders in *CONT1* and *CONT2* are marked as unused.

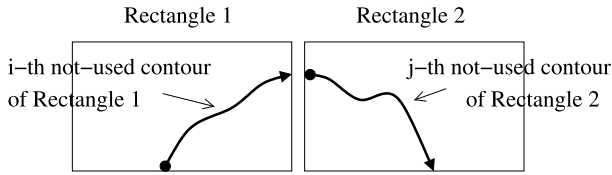


Fig. 14 Two open borders appended, resulting in a new open border copied to *CONTN*

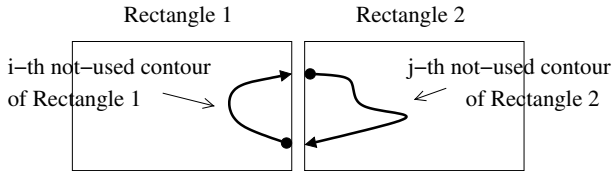


Fig. 15 Two open borders appended, resulting in a new closed border, copied to *CONTG*

The loop for borders in *CONT1* needs to take into account the different possibilities for the *i*th border in *CONT1*. Initially, there are two easy possibilities:

-The *i*th border is closed. In this case, the *i*th border is copied directly to the global output vector *CONTG*. As mentioned above, care must be taken to prevent several threads from accessing the *CONTG* vector at the same time.

-The *i*th border is an open border that does not begin or end on the common side. Then, the *i*th border is copied to *CONTN*.

The borders in *CONT1* that are marked as used are skipped, and the open borders that begin on the common side but do not end on the common side are also skipped. (These will be connected later in the main loop over the borders of *CONT2*.)

Now consider that the *i*th border of *CONT1* is an unused open border, whose last triad ends on the common side (i.e., there must be a border in *CONT2* whose first triad is on the common side and connects with the last triad of the *i*th border of Rectangle 1). Then, the code searches for this border in Rectangle 2; let this border be the *j*th border of Rectangle 2. There are different possibilities:

1. If the *j*th border of Rectangle 2 ends on a side other than the common side, we have the situation shown in Fig. 14. Then, both borders are marked as used, and a new border is generated (appending both borders). This new open border is copied to the *CONTN* output borders structure.
2. If the *j*th border of Rectangle 2 ends on the common side (goes back to Rectangle 1) and it connects to the first triad of the *i*th border in *CONT1*, then the situation would be the one shown in Fig. 15. In this case, the *i*th border in Rectangle 1 and the *j*th border in Rectangle 2 are marked as used, and a new closed border that is formed by appending both borders is copied to the global output vector *CONTG*.
3. The most troublesome situation is when the *j*th border of Rectangle 2 ends on the common side (goes back to Rectangle 1) but it connects to the first triad of a different border from Rectangle 1, which has not been used yet. This might happen repeatedly, as in the cases shown in Fig. 16. Clearly, this must be handled

size of the image exactly. Therefore, the size of all of the rectangles will be the same.

- The triads (or border points) are stored as structs that hold the current pixel, the next pixel, and the former pixel. All of the triads are stored in a vector, which resides in global GPU memory. In this implementation, the dimension of this vector is quite large so that it can cope with images with many borders. This large vector is logically split so that each thread has its own part of the vector, where the thread writes the triads that are found. The storage of the borders in a triad vector is very convenient during the search, but the resulting vector is quite large. This may be troublesome if this vector must be downloaded to the CPU. (The downloading time can be of the same order or larger than the computing time of extraction of borders.) This is not important in our target application because the outcome of the full process of the GPU is a modified binary image. However, for other applications where the contours need to be downloaded, it might be better to devise a more compact scheme for border storage.
- The tracking and contour phases have been implemented without using shared memory. The reason for this is that these phases are implemented using blocks of one thread. With only one thread per block, there is no collaboration between threads of the block. Furthermore, the shared memory that is not used in a block is automatically used as cache memory, which is as fast as the shared memory. Therefore, the use of shared memory in these kernels does not provide any advantage. On the other hand, the preprocessing phase uses shared memory and is implemented in a standard CUDA kernel, with blocks of 1024 threads.

The code is quite long and has many details that are difficult to cover in a paper. We have generated a simplified version of the code so that readers can examine and execute it. The code can be downloaded from personales.upv.es/vmgarcia/borders_cuda.tar.gz.

6 Evaluation of the proposed algorithm

We have carried out an empirical evaluation of the proposed algorithm, comparing the execution times of our algorithm with the execution times of the *findcontours* function of the OpenCV library (version 4.5). We used a set of 19 binary images, with different features and different numbers of borders, ranging from 1 to 163. Some of these images are actual binary images from car bodyworks (such as the one in Fig. 17), and some have been generated synthetically (such as the one in Fig. 18). The original size of the images is 1232×1028 (1 \times). The images are stored using a “byte” data type. Therefore, the memory needed to store each image in original format is around 1 MB. It might be possible to store the binary images as matrices of bits. However, the Suzuki algorithm (and our parallel version) needs to store values that are larger than 1 in the image (see Sect. 4.2.1), hence the need for using a “byte” data type.

Using the function *resize* from MATLAB, we obtained all of the images in two different sizes, 2464×2056 (2 \times) and 4928×4112 (4 \times). The codes were tested on

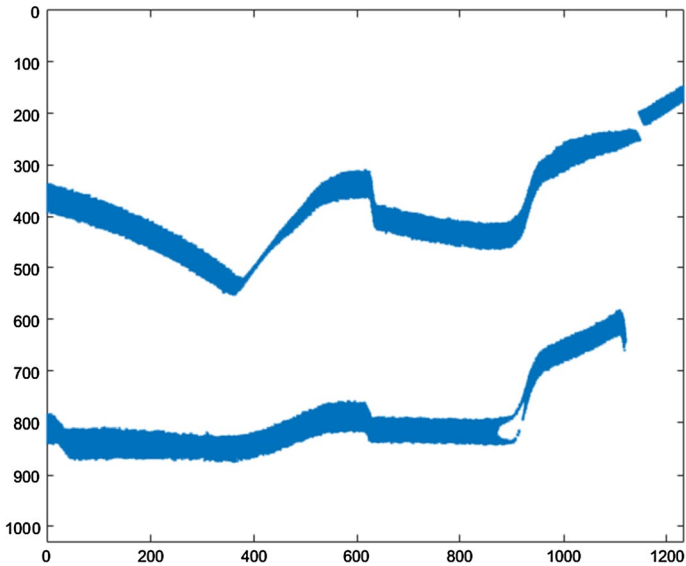


Fig. 17 Test image obtained from the defect detection system

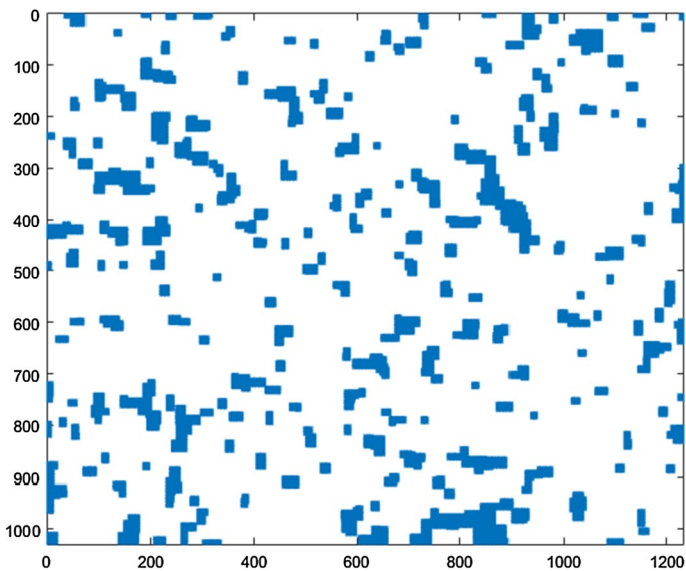


Fig. 18 Test image obtained synthetically

two different computers. The first computer (Server1) is a server that is equipped with an Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz (Turbo Boost enabled) and a Tesla K40c GPU card (with 15 multiprocessors and 192 CUDA cores per multiprocessor, for a total of 2890 CUDA cores; the base clock frequency is 745

Table 1 Average computing times (milliseconds) in Server1

	1×	2×	4×
Preprocessing	0.29	1.10	4.41
Tracking of borders	3.79	11.17	40.78
Connection	4.81	2.51	2.57
Total GPU	8.89	14.79	47.76
OpenCV	22.06	38.27	54.00

Table 2 Average computing times (milliseconds) in Server2

	1×	2×	4×
Preprocessing	0.07	0.25	0.95
Tracking of borders	0.36	0.63	1.93
Connection	0.37	0.69	0.77
Total GPU	0.80	1.57	3.65
OpenCV	22.47	48.46	120.78

MHz). The operating system in Server1 is Ubuntu 16.04.6 LTS, and the CUDA toolkit version is 8.0. The second computer (Server2) is a recent acquisition that is equipped with an Intel(R) Core(TM) i9-7960X CPU @ 2.80GHz (Turbo Boost enabled) and a Nvidia Quadro RTX 5000 (with 48 multiprocessors and 64 CUDA cores per multiprocessor, for a total of 3072 CUDA cores; the base clock frequency is 1620 MHz). The operating system in Server2 is Ubuntu 18.04.04 LTS, and the CUDA toolkit version is 10.2. It must be noted that the *findcontours* function of OpenCV implements the sequential Suzuki algorithm and therefore uses a single CPU core.

The borders in each image were extracted 20 times, and the average times were recorded. The results (borders obtained) were identical with the *findcontours* function and with our CUDA code. The images in original size were divided into 32×32 rectangles in the CUDA code, while the images in sizes 2× and 4× were divided into 64×64 rectangles.

Tables 1 and 2 show the average execution time of all of the 19 images of each of the 3 sizes on Server1 and Server2, respectively. We include the average times of each phase of the GPU algorithm: preprocessing, tracking, and connection.

The results shown in Tables 1 and 2 can be analyzed from different perspectives. First, it is clear that the proposed algorithm is faster than the OpenCV *findcontours* function, especially in Server2.

With regard to usage of the GPU, the proposed parallel algorithm uses blocks with only one thread in the tracking and connection phases. This means that the proposed algorithm underutilizes the GPU. (CUDA blocks of threads usually have hundreds and even thousands of threads.) It is difficult to measure this underutilization because the computations in Suzuki's algorithm and the parallel version are mostly memory accesses and integer operations, with hardly any floating point operations. This makes it difficult to evaluate the performance of

Table 3 Average computing times (milliseconds) from images in 4× size, varying the number of threads/rectangles

Number of threads/rectangles	1 × 1	2 × 2	4 × 4	8 × 8	16 × 16	32 × 32	64 × 64	128 × 64
Track Serv1	8888.33	2295.67	590.33	157.83	52.50	42.00	40.78	40.53
Connect Serv1	0.00	0.17	0.40	0.60	0.87	1.40	2.51	2.81
Track Serv2	1092.67	286.33	77.58	22.48	7.59	2.92	1.93	2.08
Connect Serv2	0.00	0.03	0.08	0.12	0.19	0.53	0.77	1.09
Ratio track Serv1 vs Serv2	8.13	8.02	7.61	7.02	6.92	14.40	21.13	19.07

this algorithm compared to the maximum performance of the GPU since the maximum performance of GPUs (and of CPUs) is usually given in terms of Gflops.

Another interesting matter is the difference in performance of the CUDA algorithm when running on both servers, which is around 10 times faster on the GPU of Server2 than on the GPU of Server1. In order to clarify the cause of this difference in computing times, we have executed the parallel CUDA code to find the borders of the 4× images, varying the number of rectangles, and, consequently, the number of blocks of one thread used in the tracking and the connection phase. We recorded the average tracking and connection times in both servers as well as the average ratio between the tracking times of both servers. The results are displayed in Table 3.

It must be observed that when using a single thread, the connection phase is not executed because all of the borders are fully tracked by the same thread. As expected, the tracking times obtained with one block of one thread were very slow. However, the computing times in this case were quite stable: The tracking with a single thread in Server2 was consistently around 8 times faster than the tracking with a single thread in Server1. (This was also tested with the images with 1× and 2× sizes.) This difference in speed between the cores of the two servers must be related to the clock frequency, but also to other improvements in the cores of the RTX 5000 GPU. It is also interesting to note that, for larger numbers of rectangles, the weight of the connection phase is small and only starts to be significant for numbers of rectangles larger than 32 × 32.

It can also be observed that the sum of tracking times and connection times in both servers decreases when the number of rectangles increases, until the number of rectangles (i.e., the number of blocks of one thread) is larger than 64 × 64. No advantage is obtained by increasing the number of rectangles further. For every image size, there seems to be an optimal number of rectangles, such that no advantage is obtained by exceeding that number of rectangles.

When the ratios of tracking times of the GPUs of Server1 and Server2 are compared, the ratios are more or less stable (8×) except for the larger number of rectangles, when the ratios reach 21×. We believe that this increase in performance of Server2 for large number of blocks is related to the larger number of multiprocessors in the GPU of Server2 (48 vs 15). The extra number of multiprocessors allows more blocks to be executed simultaneously.

Table 4 Detailed results from 6 images in original size (1×). Average computing times (milliseconds) in Server2

Image	1	2	3	4	5	6
N borders	1	3	15	17	121	163
Nz Pixels	2E4	9E4	7E4	7E5	7E4	1E6
t Prepro.	0.07	0.07	0.07	0.07	0.07	0.07
t Tracking.	0.35	0.32	0.36	0.38	0.34	0.45
t Connect.	0.24	0.23	0.22	0.33	0.34	0.52
Total GPU	0.66	0.62	0.65	0.78	0.75	1.04
OpenCV	15.0	16.3	15.6	27.3	20.8	32.0

Table 5 Average image uploading/downloading times (milliseconds)

	1×	2×	4×
Server1	0.41/0.38	1.02/0.99	3.61/3.44
Server2	0.21/0.19	0.70/0.52	2.62/1.92

The difference in computing time of the OpenCV *findcontours* routine between the two servers should also be noted. The results of the images in original size are nearly the same, but the difference increases with the size of the images. This seems to indicate that the difference is due to the larger cache (35 MB in Server1 vs. 22 MB in Server2) and faster bus (9.6 Giga transfers per second in Server1 versus 8 Giga transfers per second in Server2) on Server1. The technical data of both CPUs can be verified in [21, 22].

In order to clarify the performance of the algorithm and the weight of the different phases, we have selected six images from the set in original size (1×), with different numbers of borders and different numbers of nonzero pixels. The results obtained in Server2 from these images are presented in Table 4 (including the number of borders and the number of nonzero pixels in each image). It can be observed that the preprocessing is quite fast and requires the same computing time for all of the images. This indicates that it does not depend on the number of borders or on the number of nonzero pixels. As expected, the tracking and the connection phases are dependent on both the number of borders and the number of nonzero pixels.

Finally, with regard to the cost of the data transfer between the CPU and the GPU, our target application includes several stages that are computed in the GPU, one of which is border tracking. The final result is a processed image of the same size and data type, which is sent back to the CPU. Table 5 shows the average times for uploading images (to GPU) and downloading images (to CPU). The time required for uploading or downloading images (relative to the processing times) is small in Server1. In Server2, the transfer times are larger (relative to the computing times), but the transfer times are still smaller than the computing times.

7 Conclusion

The algorithm presented can extract borders from binary images with similar or better computing times than the OpenCV *findcontours* function, at least with the CPUs and GPUs used for the experiments. Although the parallelized algorithm has certain limitations (because of its intrinsic sequential nature), it performs as fast as (or faster than) CPU versions and can be used in connection with other GPU functions to perform large-scale image processing in GPUs without having to send the images to the CPU to find borders.

Modern GPUs are characterized by an increase in the number of multiprocessors. We believe that the work presented shows that, thanks to these GPU improvements, some strongly sequential algorithms can be adapted to run on GPUs. This can be useful even if the algorithm infrautilizes the GPU (as in this case).

As future work, the proposed algorithm can be modified so that the detection of borders can be done in parallel using all of the CPU cores and the parallel computing library OpenMP [23]. This is not useful for our industrial application, but it may be an interesting possibility for fast extraction of borders from very large images.

Acknowledgements This work has been partially supported by the Spanish Ministry of Science, Innovation and Universities and the European Union through Grant RTI2018-098085-BC41 (MCUI/AEI/FEDER) and by GVA through PROMETEO/2019/109. The authors would also like to thank the AUTIS, S.L. company for their support of this work.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Leinio A, Lellis L, Cappabianco F (2019) Interactive border contour with automatic tracking algorithm selection for medical images: 23rd Iberoamerican Congress, CIARP 2018, Madrid, Spain, November 19–22, 2018, Proceedings, 2019, pp 748–756. https://doi.org/10.1007/978-3-030-13469-3_87
2. Arbelaz P, Maire M, Fowlkes C, Malik J (2011) Contour detection and hierarchical image segmentation. *IEEE Trans Pattern Anal Mach Intell* 33:898–916. <https://doi.org/10.1109/TPAMI.2010.161>
3. Olszewska J (2015) Active contour based optical character recognition for automated scene understanding. *Neurocomputing*. <https://doi.org/10.1016/j.neucom.2014.12.089>
4. Soares de Oliveira L, Sabourin R, Bortolozzi F, Suen C (2002) Automatic recognition of handwritten numerical strings: a recognition and verification strategy. *IEEE Trans Pattern Anal Mach Intell* 24:1438–1454
5. Bradski G (2000) The OpenCV library. *Dr Dobb's J Softw Tools* 25:120–123

6. Versaci M, Calcagno S, Morabito FC (2015) Fuzzy geometrical approach based on unit hyper-cubes for image contrast enhancement. In: IEEE International Conference on Signal and Image Processing Applications (ICSIPA) 2015:488–493. <https://doi.org/10.1109/ICSIPA.2015.7412240>
7. Suzuki S, Abe K (1985) Topological structural analysis of digitized binary images by border following. *Comput Vis Graph Image Process* 30:32–46
8. NVIDIA Corporation (2007) NVIDIA CUDA Compute Unified Device Architecture Programming Guide, NVIDIA Corporation
9. Pitas I (2000) *Digital image processing algorithms and applications*. Wiley, USA
10. Cheong C-H, Han T-D (2006) Improved simple boundary following algorithm. *J KIISE Softw Appl* 33:427–439
11. Toussaint G (2015) Grids, connectivity and contour tracing. <http://www-cgrl.cs.mcgill.ca/~godfried/teaching/pr-notes/contour.ps>
12. Reddy P, Amarnadh V, Bhaskar M (2012) Evaluation of stopping criterion in contour tracing algorithms. *Int J Comput Sci Inf Technol* 3:3888–3894
13. Seo J, Chae S, Shim J, Kim D-C, Cheong C, Han T (2016) Fast contour-tracing algorithm based on a pixel-following method for image sensors. *Sensors Basel Switz* 16:353
14. Ren M, Yang J, Sun H (2002) Tracing boundary contours in a binary image. *Image Vis Comput* 20:125–131. [https://doi.org/10.1016/S0262-8856\(01\)00091-9](https://doi.org/10.1016/S0262-8856(01)00091-9)
15. MATLABR (2018b) The MathWorks Inc., Natick, Massachusetts, 2018
16. Ferreira A, Ubéda S (1994) Ultra-fast parallel contour tracking, with applications to thinning. *Pattern Recognit* 27(7):867–878. [https://doi.org/10.1016/0031-3203\(94\)90152-X](https://doi.org/10.1016/0031-3203(94)90152-X)
17. Cao M, Zhang F, Du Z, Liu R (2016) A parallel approach for contour extraction based on cuda platform. *Int J Simul Syst Sci Technol* 17:1.1-1.5. <https://doi.org/10.5013/IJSSST.a.17.19.01>
18. Shin PJ, Gao X, Kleihorst R, Park J, Kak AC (2008) An efficient algorithm for the extraction of contours and curvature scale space on simd-powered smart cameras. In: Second ACM/IEEE International Conference on Distributed Smart Cameras 2008:1–10. <https://doi.org/10.1109/ICDSC.2008.4635714>
19. Butt MU, Morris J, Patel N, Biglari-Abhari M (2015) Fast accurate contours for 3d shape recognition. In: IEEE Intelligent Vehicles Symposium (IV) 2015:832–838. <https://doi.org/10.1109/IVS.2015.7225788>
20. Pavlidis T (1982) *Algorithms for graphics and image processing*. Springer, Berlin. <https://doi.org/10.1007/978-3-642-93208-3>
21. Xeon e5-2697 v3 @ 2.60ghz technical data. Accessed 06 Sept 2021. <https://ark.intel.com/content/www/es/es/ark/products/81059/intel-xeon-processor-e5-2697-v3-35m-cache-2-60-ghz.html>
22. Core(tm) i9-7960x cpu @2.80ghz technical data. Accessed 06 Sept 2021. <https://www.intel.es/content/www/es/es/products/sku/126697/intel-core-i97960x-xseries-processor-22m-cache-up-to-4-20-ghz/specifications.html>
23. OpenMP v 4.5 specification (2015) <http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.