The Journal of Supercomputing (2023) 79:9915–9931
https://doi.org/10.1007/s11227-023-05052-2

Parallel border tracking in binary images for multicore computers

Victor M. Garcia-Molla[1] · Pedro Alonso-Jordá[1]

Accepted: 9 January 2023 / Published online: 23 January 2023
© The Author(s) 2023

Abstract
Border tracking in binary images is an important operation in many computer vision applications. The problem consists in finding borders in a 2D binary image (where all of the pixels are either 0 or 1). There are several algorithms available for this problem, but most of them are sequential. In a former paper, a parallel border tracking algorithm was proposed. This algorithm was designed to run in Graphics Processing units, and it was based on the sequential algorithm known as the Suzuki algorithm. In this paper, we adapt the previously proposed GPU algorithm so that it can be executed in multicore computers. The resulting algorithm is evaluated against its GPU counterpart. The results show that the performance of the GPU algorithm worsens (or even fails) for very large images or images with many borders. On the other hand, the proposed multicore algorithm can efficiently cope with large images.

Keywords Border tracking · Computer vision · Parallel computing · GPU computing · OpenMP · Multicore 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 [5]. It can be used to find borders in color images or in grayscale images by applying appropriate thresholds to the image [6].

✉ Victor M. Garcia-Molla
vmgarcia@dsic.upv.es

Pedro Alonso-Jordá
palonso@upv.es

[1] Department of Information Systems and Computation, Universitat Politècnica de València, Camino de Vera s/n, Valencia 46022, Spain

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 OpenCV, the well-known library for computer vision [8].

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. The Suzuki algorithm is a sequential process.

A GPU parallel border tracking algorithm was proposed in [9]. This algorithm was developed to suit the requirements of a company that is devoted to the automatic detection of defects in car bodyworks. This algorithm, written in CUDA [10], was based on the Suzuki algorithm. The parallel algorithm proposed in [9] (which we will call the GPU version) proved to be very efficient for images with a small or medium number of borders (1–500). However, when the number of borders grows, the performance deteriorates. It has also been detected that, for large binary images (larger than 10000×10000 pixels), the memory needed to run the GPU version (on our computer) was too large. The main reason for the low performance for large images is that (as was acknowledged in [9]) some of the phases of the parallel algorithm cannot take full advantage of the GPU. Some of the phases of the algorithm had to be implemented using CUDA blocks of a single thread.

In this paper, we study how to correct these shortcomings, by modifying the GPU version to run in multicore CPUs, using the standard parallelization library OpenMP [11]. (We call this our OpenMP version or CPU version). The OpenMP version obtained runs faster than the GPU version in some phases of the parallel algorithm, but slower in others. Overall, the performance of this new version improves on the GPU version when the images processed are large.

The structure of the paper is as follows. First, in Sect. 2, we describe the problem of border tracking for binary images, outline the original Suzuki method, and describe the GPU version. In Sect. 3, we describe the proposed multicore parallel algorithm. Section 4 is devoted to the evaluation of the proposed algorithm. Finally, the conclusions and possible future work are discussed in Sect. 5.

2 Motivation and definition of the problem

The work described in [9] was driven by the need for a GPU implementation of border tracking in a real-time system for automatic detection of defects in car bodyworks. The detection of the defects requires large binary images to undergo several processes; an important part of these processes is to obtain the borders on the images. In that particular application, for the sake of efficiency, the whole process is carried out in the GPU.

The modifications proposed in this paper are based on the GPU version described in [9]. Therefore, we start by describing the GPU version with enough detail so that this paper is self-contained. We only consider rectangular binary images and 8-connectivity, that is, the pixel (i, j) is neighbor (is connected) to every pixel that touches one of its edges or corners [12].

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 padded with zeros. As a consequence, all of the borders of the image are closed.

The sequential Suzuki algorithm described in [7] examines all of the pixels in the input image using a standard double loop. (see the high-level description in Algorithm 1). The origin of coordinates is the top left corner of the image. When the standard Suzuki sequential algorithm finds a border pixel *P* in a border not yet tracked, 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.

Algorithm 1 Suzuki sequential algorithm for finding all of the borders in a binary image A with $M \times N$ pixels

```
1:  for i = 1 to M do
2:      for j = 1 to N do
3:          if A(i, j) is in a border not tracked yet then
4:              while border not closed do
5:                  Search next Pixel, add it to the border
6:                  mark pixel as visited
7:              end while
8:          end if
9:          Store border
10:     end for
11: end for
```

Now, we turn to the parallel algorithm proposed in [9]. The main idea behind the proposal in [9] is to split the image into *NX* by *NY* rectangles of the same size. If the dimensions of the image are not multiples of *NX* and/or *NY*, then the dimensions of the image are increased (with zero-valued pixels) until the next multiples of *NX* and *NY*. Therefore, the new dimensions of the image divide the parameters *NX* and *NY* exactly. Figure 1 shows a typical test image, and Fig. 2 displays a possible splitting of the same image.

Then, similarly to the Suzuki algorithm, a process is launched for each rectangle, which tracks and stores all of the borders in its rectangle. Clearly, some borders can belong to more than one rectangle. The final step is to connect the borders from the different rectangles. A high level description of the algorithm proposed in [9] is portrayed in Algorithm 2.

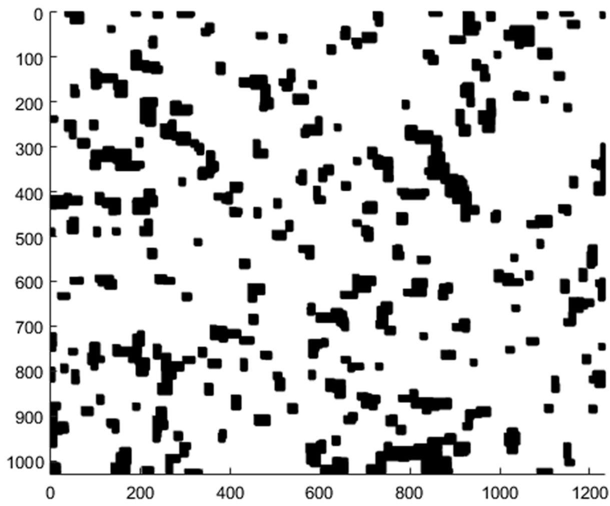

Fig. 1 Image 1, obtained synthetically, generating random dark zones

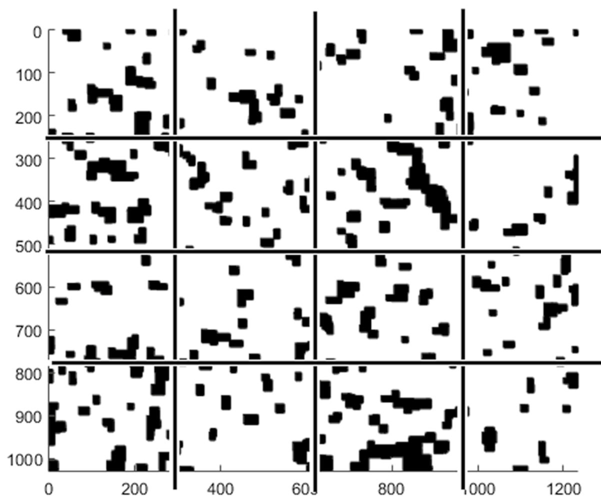

Fig. 2 Image 1, split into 4×4 rectangles

Algorithm 2 Parallel algorithm for finding all of the borders in a binary image.

1: split the image into NX by NY rectangles
2: parallel preprocessing of all of the pixels
3: parallel tracking of the borders in each rectangle
4: parallel connection of all of the borders of the image

Algorithm 2 has two different algorithmic levels: The "image" level (or high level) scheme (division of the image in rectangles, parallel processing of the borders in each rectangle, and organization of the connection of all rectangles), and the "rectangle" level (or low level) details (preprocessing in each rectangle, tracking in each rectangle, and connection of the borders of two neighbor rectangles). In this paper, we are interested in obtaining an OpenMP version. For the OpenMP version, the low-level processing is virtually the same as in the GPU version. Therefore, we will only focus on the high-level details of the algorithm, which are the parts that need to be changed. The three main phases of the GPU version are briefly described below. These are preprocessing (Sect. 2.1), border tracking in rectangles (Sect. 2.2), and connection of the borders of all of the rectangles (Sect. 2.3).

2.1 Preprocessing in GPU

The first step is 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 a value of 0 in any of the positions $(i+1, j)$, $(i-1, j)$, $(i, j-1)$, $(i, j+1)$. This check can be performed independently for all of the pixels in the image. Therefore, this check can be carried out in parallel for all pixels, which is very appropriate for GPU computing. This check is carried out easily and efficiently in the GPU by using a CUDA kernel called *preprocessing_gpu*. This kernel uses blocks of 32 per 32 threads (each thread checks a single pixel) and as many blocks as needed to process the whole image. 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. This array is used to speed up the second phase, the parallel tracking.

2.2 GPU border tracking in rectangles

As mentioned above, the key idea in the GPU version is to divide the image into small rectangles. The tracking of the borders in each rectangle can be carried out independently of the tracking in any other rectangle; hence, the tracking in all of the rectangles can be computed in parallel. In the GPU algorithm, we used a block of threads for the tracking of each rectangle. However, since the tracking must be done sequentially within each rectangle, we use a single thread for tracking the borders in each rectangle. Assuming a division of the image in *NX* per *NY* rectangles, the

tracking is launched with a kernel (called *parallel_tracking*) of *NX* per *NY* blocks with one thread in each block.

The implementation of the tracking phase in the parallel case is far more complex than in the sequential case. 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 so that (in borders distributed across several rectangles) they can be properly connected in the final phase. The labeling needed is quite complex because a single pixel can be part of up to four different borders. In addition, it must be ensured that borders already tracked are not tracked again.

Another problem that can arise with large images is the potentially large storage required. It can happen that a given set of pixels can be part of two different borders. This can happen when the pixels are tracked n different order. This requires the borders to be stored as a sequence of "triads" (ordered sequences of coordinates of pixels), including the coordinates of the present pixel, the former pixel and the next pixel, plus another integer number pointing to the next triad. Then, the storage needed for a single "triad" is 7 integer values, (of 4 each), for a total of 28 bytes. Each pixel of the binary image is stored as an unsigned integer of 8 bits (1 byte), that is, each triad needs 28 times more storage than a pixel. Furthermore, for efficiency in GPUs, it is necessary to allocate enough static memory to hold all of the (possibly many) borders. Because of this, the storage needed to store the borders in large images can be much larger than the storage needed for a binary image. This may be a limiting factor of the usability of the algorithm. All of the fine implementation details are described in [9].

2.3 GPU connection of the borders of all of the rectangles

After the tracking stage, the thread that processes a given rectangle will have generated a data structure where the borders in that rectangle are stored as ordered sequences of "triads". When the borders of all of the rectangles have been computed, the connection between the open borders from different rectangles starts. The key for the parallel connection algorithm is that the connections between borders in two neighbor rectangles can be established independently from any other connection between other pair of neighbor rectangles. However, as in the tracking phase, the connection of two neighbor rectangles must be carried out sequentially. Therefore, we again used CUDA blocks with just one thread. The low-level process of connecting the borders from two neighbor rectangles is quite complex, especially because some borders can exit and re-enter a rectangle several times. The low-level process of connecting the borders of two rectangles is described in detail in [9].

As described in [9], we wrote a CUDA kernel for the vertical connection *vert_connection* $<<< X, Y >>>$ such that the only thread of the block $(i,j), 1 \leq i \leq X, 1 \leq j \leq Y)$ connects the borders of rectangle $(2(i-1)+1, j)$

with the borders of the rectangle ($2i$, j). We wrote a similar CUDA kernel for the horizontal connection: *horz_connection* $<<< X, Y >>>$ such that the only thread of the block $(i,j)(1 \leq i \leq X, 1 \leq j \leq Y)$ connects the borders of rectangle $(i, 2(j-1)+1)$ with the borders of the rectangle (i, $2j$).

There are many possible arrangements for a parallel connection. We have chosen to use numbers of rectangles *NX*, *NY* 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 has $log_2(NX)$ stages and the horizontal sweep will have $log_2(NY)$ stages. Algorithm 3 outlines the high-level structure of the connection. When Algorithm 3 concludes, a single structure (that stores all of the borders of the image) is obtained.

Algorithm 3 Connection of the borders of all of the rectangles in an image divided in $NX \times NY$ rectangles

```
1:  NX = NX/2
2:  while NX > 1 do
3:      vert_connection <<< NX, NY >>>
4:      NX = NX/2
5:  end while
6:  NY = NY/2
7:  while NY > 1 do
8:      horz_connection <<< 1, NY >>>
9:      NY = NY/2
10: end while
```

For the sake of clarity, we want to show how the parallel connection would proceed using an example where the image is divided into 4×4 rectangles.

Figure 3, left, depicts a possible image divided into 4×4 rectangles. The borders of all of the rectangles in the image on the left have been obtained, so that the vertical connection can start. In the first stage of the vertical connection, the thread of block (1,1) would connect the borders from rectangles (1,1) and (2,1) and would store them as the borders in a larger rectangle union of rectangles (1,1) and (2,1) (rectangle (1,1) in the new structure, Fig. 3 right). Similarly, the thread of block (2,1) would connect the borders from rectangles (3,1) and (4,1), and store them as the borders in the rectangle union of rectangles (3,1) and (4,1) (rectangle (2,1) in the image on the right). Then, in the example, the full stage can then be carried out with 8 blocks (8 threads) in parallel, resulting in 8 structures holding the borders of the new 8 rectangles depicted on the right in Fig. 3.

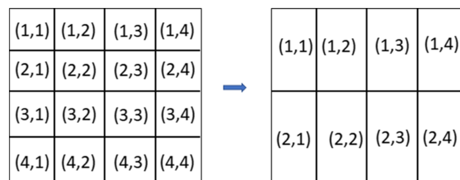

Fig. 3 Example of a vertical connection applied to an image divided into 4×4 rectangles, 1st stage

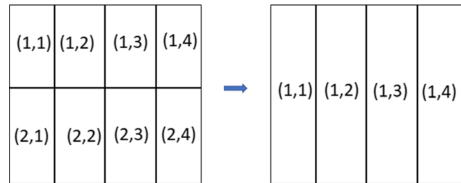

Fig. 4 Example of a vertical connection applied to an image divided into 4 × 4 rectangles, 2nd stage

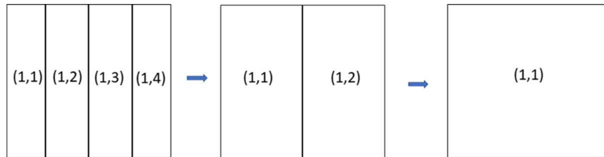

Fig. 5 Example of a horizontal connection (final two stages) applied to an example image

Another stage of the vertical connection can be applied, now using just 4 blocks (4 threads) in parallel, resulting in 4 structures, holding the borders of the 4 rectangles on the right of the Fig. 4. This concludes the vertical connection.

Now, the horizontal connection must start (Fig. 5). The procedure is very similar to the vertical connection, connecting borders from neighbor rectangles, but now the horizontal neighbors will be connected. Since only 4 rectangles are left, only two stages are needed: the first stage involves only 2 blocks (2 threads) and the second (and last) stage involves only one block (one thread). The final structure holds all of the borders of the image.

3 OpenMP implementation

The algorithm proposed in [9] can be modified to generate an OpenMP version through relatively straightforward changes. However, careful programming is needed in order to obtain an efficient version.

3.1 Preprocessing in CPU

In the GPU version of preprocessing, each block of threads processes a rectangle. Our first approach for the CPU version was to use a similar structure, dividing the image in rectangles, and creating a high-level loop that runs through all of the rectangles. This loop can be parallelized using the OpenMP construct "parallel for".

However, we found out through experimentation that, in the CPU version, the subdivision into rectangles did not provide the most efficient organization. In this case, it was faster to use a double loop running through all of the pixels of the image, parallelizing the outer loop with OpenMP and vectorizing the inner loop by using AVX vector instructions [13].

In order to obtain an efficient implementation, an appropriate ordering of the loops must be chosen, so that the cache misses are minimized. This was not

important in the GPU version (because of the special features of the memory accesses in GPU), but it is crucial in the OpenMP version. In this case, the image is stored by columns; therefore, the inner loop must be the one that runs through rows of pixels (and, therefore, accesses the pixels of the image sequentially). Furthermore, it is important to ensure the use of the best vectorizing instruction set available (AVX-512 in our computer). In this case, we were using the g++ compiler version 7.50, and we realized (using the compiler flag "-fopt-info-vec-optimized") that the standard optimizing flag "-O3" does not enforce the use of the instruction set AVX 512. We achieved this by additionally using the "-mavx512f" compiler flag.

3.2 CPU border tracking in rectangles

In the GPU version, the border tracking in rectangles is launched with a kernel call, which uses one thread per rectangle. If the image is split into $NX \times NY$ rectangles, then the kernel call creates $NX \times NY$ blocks of threads, each of which has a single thread. Each thread processes and stores the borders in a rectangle. In order to perform the same task in a multicore computer, this kernel launch can be replaced with a loop that runs through the $NX \times NY$ rectangles. Since the tracking in a rectangle is independent from the tracking in any other rectangle, this loop can be parallelized using OpenMP.

The cost of tracking borders in a rectangle may vary strongly from one rectangle to another (unlike in the preprocessing phase, where all of the pixels of all of the rectangles are processed, and the computational cost should be very similar for all of the pixels) if the number of borders or/and the length of the borders varies from one rectangle to another. Assuming that the number of rectangles is larger than the number of threads available, the default work distribution of the parallel loop assigns a fixed set of rectangles to each thread. If some rectangles with many borders are assigned to a thread, this thread may require a long time to process its set of rectangles, penalizing the overall computational cost. Therefore, it becomes necessary to use the "dynamic" pragma of OpenMP. By using this pragma, the assignment of rectangles to threads is automatically readjusted so that the processing of rectangles not yet processed may be reassigned to threads that have already completed their assignment.

3.3 CPU connection of the borders of all of the rectangles

The GPU connection phase described in Sect. 2.3 can also be adapted for execution in multicore CPUs by substitution of the kernel calls (*vert_connection* and *horz_connection*) with "parallel for" loops. For example, if the code for computing the vertical connection between rectangles $(2(i-1)+1, j)$ and $(2i, j)$ is embodied in the function *vert_connection_cpu*(i,j), then the kernel call *vert_connection* in Algorithm 3 can be replaced with the double loop in Algorithm 4:

Algorithm 4 CPU version of vertical connection

```
1: for j ← 1 to NY do
2:     for i ← 1 to NX do
3:         vert_connection_cpu(i, j)
4:     end for
5: end for
```

Both loops can be simultaneously parallelized using OpenMP. We collapsed both loops in one, in order to avoid unnecessary synchronization points.

Algorithm 5 Parallelized CPU version of vertical connection

```
1: # pragma omp parallel for
2: for rect ← 1 to (NX * NY) do
3:     i=1+((rect-1)/NX)                  ▷ Integer division
4:     j=1+((rect-1)% NX)                 ▷ Remainder of Integer division
5:     vert_connection_cpu(i, j)
6: end for
```

The call to *horz_connection* can be similarly replaced. In the connection phase, the "dynamic" pragma has hardly any effect.

4 Evaluation of the proposed OpenMP algorithm and comparison with the GPU algorithm

The experimental evaluation of the proposed algorithm (and the comparison with the GPU version) was carried out in our main computer (named Server1). This computer is equipped with an Intel(R) Core(TM) i9-7960X CPU @ 2.80GHz (Turbo Boost enabled) with 16 cores and 64 GB and a Nvidia Quadro RTX 5000 GPU (with 48 multiprocessors and 64 CUDA cores per multiprocessor, for a total of 3072 CUDA cores; the base clock frequency is 1620 MHz and the total memory is 16 GB). The operating system in Server1 is Ubuntu 18.04.04 LTS, and the CUDA toolkit version is 10.2. The CPU version was compiled using *g++* version 7.50 (the same version used by the *nvcc* CUDA compiler).

The comparison of performance between these two algorithms is troublesome because there are many factors that may influence the performance:

- The hardware used.
- The size of the images.
- The number of borders in the images.
- The number of nonzero pixels in the images.
- The borders may be concentrated in a few rectangles, affecting the work distribution.

- The number of rectangles used to split the image.
- The inclusion (or not) of memory transfers from/to the GPU.

Furthermore, the proportional weight of the computational cost of the three phases (preprocessing, tracking, connection) is different in the two versions.

We have chosen to include only the computing times in CPU or in GPU, without including the memory transfers to/from GPU from/to CPU. We think that this is consistent with the study in [9], where the computation was part of other computations carried out in the GPU. We have chosen to split the image using 32×32 rectangles in all of the experiments. We have tested this choice experimentally and, in our test cases, it is an optimal or nearly optimal choice. Interestingly enough, this result is similar for the CPU and for the GPU versions.

Our initial experiment was to evaluate both algorithms using two images of different properties, obtained in different sizes. This experiment was designed to bring out the differences in performance between the two algorithms. The images, (Image 1 and Image 2), are shown in Figs. 1 and 6. Image 1 has 163 borders and 134398 nonzero elements, while Image 2 is more complex. It has 502,26 borders and 853,717 nonzero elements. The original size of the images is 1028×1232 pixels. We will name this initial image size as the 1× size. Using the function *imresize* from Matlab [14], we obtained versions of both images in six different sizes, i.e., 2× versions of both images were obtained by doubling the size to 2056×2464. Then, we obtained 4× versions, of size 4112×4928, and so on, until 12×.

The images are stored using a “byte” data type for each pixel. Therefore, the memory needed to store each image in original format is around 1 MB.

Both versions were tested with the two images in the six sizes. In Server 1, the GPU version failed to allocate enough GPU memory for the 12× images (and larger), while the CPU version worked correctly up to 24× size. In all of the cases without memory allocation issues, the borders were correctly obtained. For each case, the borders were obtained 20 times. The times obtained are the average times of the 20 experiments.

Figure 7 shows the average computing time of both versions, for sizes of the images 1×, 2×, up to 12×. These times are the aggregated computing times of

Fig. 6 Image 2, obtained combining a picture of a car bodywork with a synthetically obtained image with circles

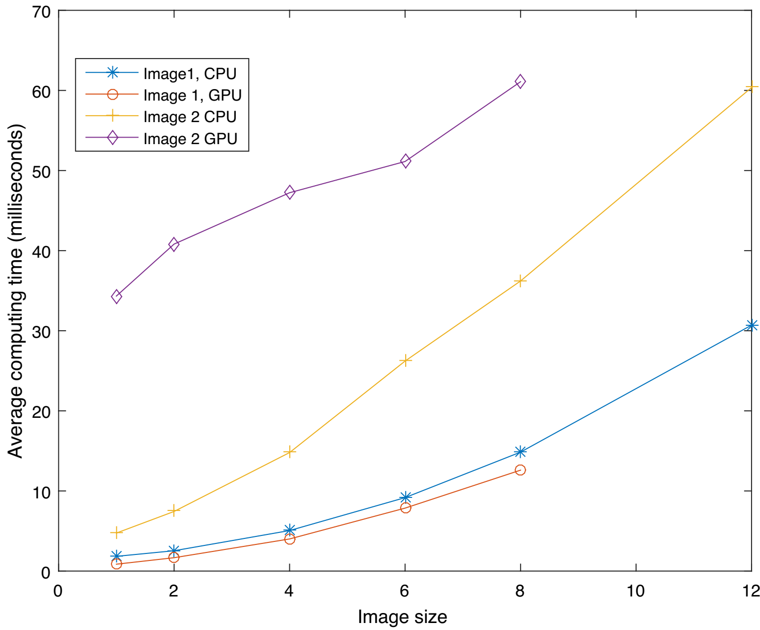

Fig. 7 Computing times of the GPU and CPU versions applied to Image 1 and Image 2

preprocessing, tracking, and connection. Tables 1, 2, and 3 show the separate computing times of each phase for each image.

Many insights can be obtained from this experiment. First, when looking at the aggregated computing times (Fig.7), it can be observed that the GPU version does obtain some advantage for the simpler Image 1. However, the CPU version is clearly faster when both versions are applied to Image 2.

As can be seen in Table 3, the reason for the CPU version being faster when applied to Image 2 (Fig. 6) is that the connection phase is very slow in the GPU when the

Table 1 Average computing times of the preprocessing phase (milliseconds) in Server 1

	1×	2×	4×	6×	8×	12×
GPU Image 1	0.04	0.16	0.58	1.23	2.23	F
CPU Image 1	0.14	0.35	1.24	2.72	4.8	12.48
GPU Image 2	0.04	0.14	0.55	1.23	2.2	F
CPU Image 2	0.14	0.35	1.45	3.44	6.0	15.53

Table 2 Average computing times of the tracking phase (milliseconds) in Server 1

	1×	2×	4×	6×	8×	12×
GPU Image 1	0.44	1.23	3.15	6.34	10.09	F
CPU Image 1	0.37	0.82	2.47	5.0	8.7	16.87
GPU Image 2	1.94	5.38	11.18	19.9	28.1	F
CPU Image 2	2.85	5.08	11.44	20.8	28.3	42.87

Table 3 Average computing times of the connection phase (milliseconds) in Server 1

	1×	2×	4×	6×	8×	12×
GPU Image 1	0.38	0.29	0.28	0.29	0.28	F
CPU Image 1	1.33	1.37	1.35	1.45	1.35	1.35
GPU Image 2	32.35	35.3	35.5	30	30.8	F
CPU Image 2	1.8	2.06	1.95	2.02	1.96	2.0

number of borders is large. This is consistent with the conventional wisdom regarding GPU programming because these kernels were launched using blocks of a single thread. Furthermore, in the last stages of the connection phase, the number of threads diminishes. This clearly underutilizes the GPU. Additionally, it can also be observed that for both versions the connection time is independent from the size and greatly depends on the number of borders.

On the other hand, the computing times of the parallel tracking (see Table 2) are quite similar in both versions. This is an interesting result because this phase was also implemented in GPU using blocks with only one thread. The difference with the connection phase lies in the fact that, in the final stages of the connection phase (horizontal connection in Algorithm 3), fewer and fewer threads are used, until only one is used. On the other hand, in our experiments, the parallel tracking kernel is always launched with 32×32 blocks of one thread. It seems that as long as the number of one-thread blocks launched is large, the performance of the one-thread blocks is acceptable and (to some extent) it compares reasonably well with the performance of CPU cores.

Table 1 shows that the preprocessing phase is faster in the GPU, but not enough to counter the slowness of the GPU connection phase. We considered the possibility of creating a "hybrid" version, with the preprocessing being executed in the GPU and the tracking and connection phases being executed in the CPU. However, in that case, the cost of the memory transfers (sending the image to the GPU and sending back the *is_border* array to the CPU) is too large. Table 4 displays the times needed to upload or download images of the considered sizes. The table shows that the upload/download times are larger than the preprocessing times in the CPU version in all of the cases.

The next experiment aims to highlight the effect on the computing times of varying the number of borders. For this purpose, we generated five synthetic images, all of which had the same size 4× and similar structure, with black squares that were regularly spaced. An example of this kind of image is shown in Fig. 8.

Since the number and size of the black squares can be changed, the number of borders can be changed. We generated images with the following number of borders: 5576, 12,546, 22,468, 50,430, 89,872. The computational times are shown in Fig, 9. It is quite clear that an increase in the number of borders affects the

Table 4 Average image uploading/downloading times (milliseconds)

	1×	2×	4×	6×	8×
Server1	0.21/0.19	0.70/0.52	2.36/2.53	5.56/5.37	8.94/8.48

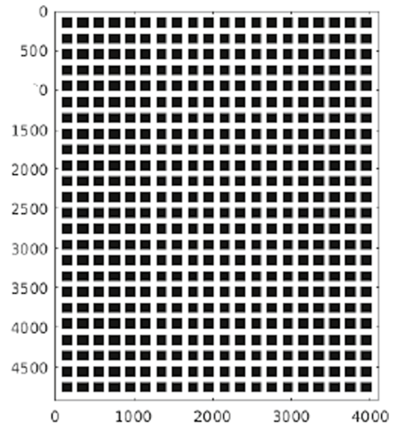

Fig. 8 Sample synthetic image with regularly spaced small black squares

GPU version much more than the CPU version. Again, this is due to the larger cost of the connection phase in the GPU version.

We also studied the scalability of the CPU algorithm. For this experiment, we used a computer (named Server 2) equipped with 2 Intel(R) Xeon(R) E5-2698 CPUs of 20 cores each, with hyperthreading activated, and 512 GB. We tested the scalability of the CPU algorithm by running it with 1, 2, 4, 8, 16, 32, 40 and 80 OpenMP threads, using Image 2 in 4× size. Table 5 shows the computing times

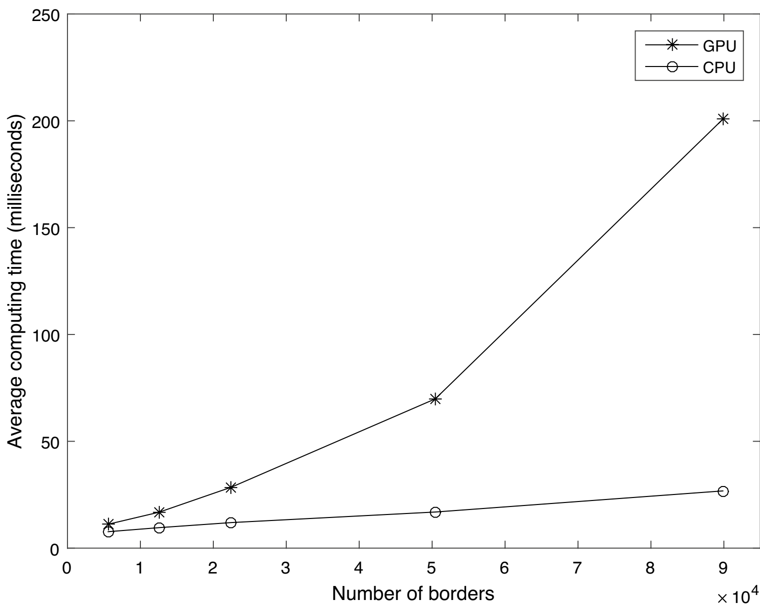

Fig. 9 Computing times of the GPU and CPU versions for synthetic images with varying number of borders

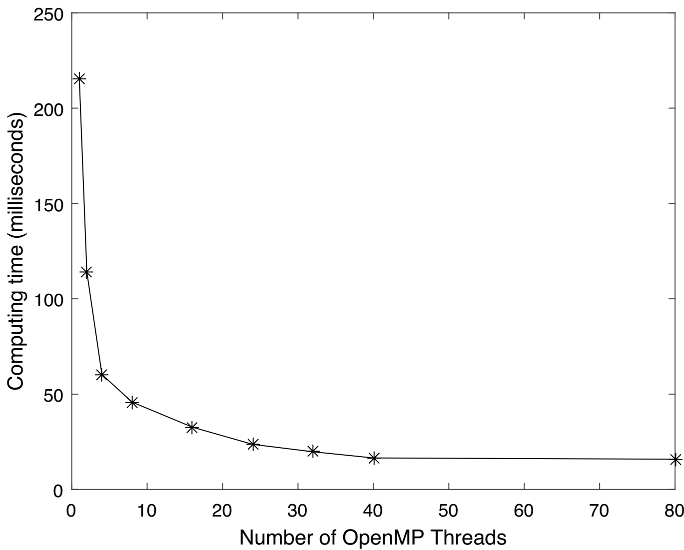

Fig. 10 Computing times of the CPU version varying the number of OpenMP threads (in Server 2)

Table 5 Detailed computing times (milliseconds) for the CPU version varying the number of OpenMP threads (in Server 2)

Phase/threads	1	2	4	8	16	24	32	40	80
Preproc.	167.2	89.0	46.6	34.1	23.0	16.7	13.0	10.4	9.8
Tracking	43.3	21.4	11.3	8.5	6.1	4.1	3.1	2.5	1.9
Connect	5.6	4.1	3.2	3.3	3.5	3.5	3.6	3.6	4.1

of the different phases when the number of threads varies. Figure 10 shows the aggregated computing times.

It can be observed that the CPU version scales quite well. The case with 80 threads makes use of hyperthreading.

Finally, we want to justify some of the choices described in the paper experimentally, using Images 1 and 2 in 4× size.

The difference caused by the use of AVX512 instructions is especially relevant in the preprocessing phase. As an example, the computing time of the preprocessing phase using the flags "-O3" and "-mavx512f" is 1.24 ms. in Image 1 and 1.45 ms in Image 2, compared with 2.37 ms in Image 1 and 2.88 ms in Image 2 when only "-O3" is used.

The effect of the "dynamic" pragma in the tracking phase can be observed by checking the computational times with the "dynamic" pragma (2.51 ms in Image 1, and 11.4 ms in Image 2) or without it (2.88 ms in Image 1, and 14.28 ms in Image 2).

We have generated a working version of the code so that readers can examine and execute it. The images used in the experiments are included in the downloadable file. The link can be found in Sect. 6.

5 Conclusion

In this paper, we have described the implementation of a parallel border-tracking method for use in multicore machines. For images of small to moderate size, the GPU version described in [9] is faster than well-known sequential CPU implementations. However, the tracking and connection phases of the GPU algorithm could only be implemented in CUDA by using blocks of only one thread. This works quite well for moderate size images, but the situation is different when the images are very large or with many borders. In these cases, the performance of the GPU version deteriorates. This fact motivated the development of an OpenMP version that designed to run on multicore computers. The experiments described in Sect. 4 show that the OpenMP version can cope efficiently with large images and also with images with a large number of borders.

As future work, we think that it is possible to improve the parallel efficiency of the OpenMP version by removing the implicit synchronization point between the parallel tracking and the connection. It should be possible to start the connection phase before finishing the tracking phase.

Acknowledgements The authors would like to thank the AUTIS, S.L. company for their support of this work.

Author contributions VMG-M and PA-J developed and programmed the algorithm. Both authors were also involved in the writing and editing of the paper.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This work has been partially supported by the Spanish Ministry of Science, Innovation, and Universities, jointly with the European Union, through Grants RTI2018-098085-BC41, PID2021-125736OB-I00 and PID2020-113656RB-C22 (MCIN/AEI/10.13039/501100011033/, "ERDF A way of making Europe"). Also, the GVA has partially supported this research through project PROMETEO/2019/109.

Availability of data and materials Code available in http://personales.upv.es/vmgarcia/borders_cuda.tar.gz and http://personales.upv.es/vmgarcia/borders_omp.tar.gz.

Declarations

Conflict of interest The authors declare no competing interest.

Ethical approval not applicable.

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, 748–756. https://doi.org/10.1007/978-3-030-13469-3_87
2. Arbelaez 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. Thida M, Chan KL, Eng H-L (2006) An improved real-time contour tracking algorithm using fast level set method. In: Chang L-W, Lie W-N (eds) Advances in image and video technology. Springer, Berlin, pp 702–711
6. Yang S, Xu S, Zeng X, Pan Y (2018) Multi-objective boundary tracking method in grayscale image. In: 2018 joint international advanced engineering and technology research conference (JIAET). https://doi.org/10.2991/jiaet-18.2018.8
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. Bradski G (2000) The OpenCV library. Dr. Dobb's J Software Tools 25:120–125
9. Garcia-Molla VM, Alonso-Jordá P, García-Laguía R (2022) Parallel border tracking in binary images using GPUs. J Supercomput 78(7):9817–9839. https://doi.org/10.1007/s11227-021-04260-y
10. NVIDIA Corporation: CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [Online; November, 2022] (v11.8.0, last updated November 9, 2022)
11. OpenMP v 4.5 specification (2015) http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
12. Pavlidis T (1982) Algorithms for graphics and image processing, 1st edn. Springer-Verlag Berlin. https://doi.org/10.1007/978-3-642-93208-3
13. Intel AVX. Accessed: 2022-08-02. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html
14. MATLAB: (R2018b). The MathWorks Inc., Natick (2018)

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