CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2013)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3163

A comparative study of parallel software SURF implementations

Alejandro Hidalgo-Paniagua [1,*,†], Miguel A. Vega-Rodríguez [1], Nieves Pavón [2] and Joaquín Ferruz [3]

[1] *Department of Technologies of Computers and Communications, University of Extremadura, Polytechnic School, Cáceres, Spain*
[2] *Department of Information Technology, University of Huelva, Higher Technical School of Engineering, Huelva, Spain*
[3] *Department of Systems Engineering and Automation, University of Sevilla, Higher Technical School of Engineering, Sevilla, Spain*

SUMMARY

Nowadays, it is common to find problems that require recognizing objects in an image, tracking them along time, or recognizing a complete real-world scene. One of the most known and used algorithms to solve these problems is the *Speeded Up Robust Features (SURF) algorithm*. *SURF* is a fast and robust local, scale and rotation invariant, features detector. This means that it can be used for detecting and describing a set of *points of interest* (*keypoints*) from an image. Because of the importance of this algorithm and the rise of the parallelism-based technologies, in the last years, diverse parallel implementations of SURF have been proposed. These parallel implementations are based on very different techniques: Compute Unified Device Architecture, OpenMp, OpenCL, and so on. In conclusion, we think valuable a comparative study of all of them highlighting the advantages and disadvantages of each parallel implementation. To our best knowledge, this article is the first attempt to do this comparative study. In order to make this study, we have used the standard metrics and image collection in this field, as well as other important metrics in parallelism as speedup and efficiency. Copyright © 2013 John Wiley & Sons, Ltd.

Received 1 March 2013; Revised 13 September 2013; Accepted 14 September 2013

KEY WORDS: parallel implementations; SURF; comparative study; CUDA; OpenCL; OpenMP; OpenCV

1. INTRODUCTION

In computer vision and robotics, it is becoming more common to find problems that require real-time image analysis ([1] and [2]). Normally, these problems need to recognize objects, track them along time, or recognize a complete real-world scene.

In order to solve these problems, a common approach involves selecting from each image a set of pixels by their properties, location, and neighborhood structure that characterize the image. These points are known as *keypoints* or *points of interest* ([3] and [4]). Furthermore, extracting keypoints from an image is not enough to solve the problem, and it is necessary to label them in a unique way to identify them in future matches and searches. This characterization is known as obtaining *descriptors of keypoints* ([5] and [6]).

One of the most commonly used algorithms for this task is the Speeded Up Robust Features (SURF) algorithm. SURF is a fast and robust local, scale and rotation invariant, features detector. This means that it can be used for detecting and describing a set of keypoints from an image ([7] and [8]).

*Correspondence to: Alejandro Hidalgo-Paniagua, Department of Technologies of Computers and Communications, University of Extremadura, Polytechnic School, Cáceres, Spain.

†E-mail: ahidalgop@unex.es

Currently, vision devices deliver high resolution images, and because of this, more computational resources are now needed to process them in real time or at least in acceptable time ([9] and [10]). For this reason, the use of sequential algorithms to solve the previous problem has started to have a high processing time. The solution to this is to use parallelism techniques that reduce the execution time and increase the algorithm performance. In this sense, diverse parallel SURF implementations exist in literature. However, to our best knowledge, there is not any previous work in literature that analyzes, surveys, and compares these parallel implementations. This is the main contribution of our paper. Our work not only compares the runtime and speedup of the different parallel implementations but also studies the stability of both the detection and the description. Different hardware platforms (multi-core CPUs and graphics processing units (GPUs)) are used in our analysis in order to present a more complete comparison. Finally, a scalability and efficiency study has also been included showing the behavior of the different parallel implementations when the number of cores increases.

The rest of the paper is organized as follows. In Section 2, a brief description of SURF algorithm can be found. In Section 3, the related work is described. Section 4 explains the methodology we used to evaluate all parallel SURF implementations. Section 5 presents the evaluation results. Finally, section 6 explains the main conclusions obtained by the study.

2. THE SURF ALGORITHM

The *SURF* algorithm was proposed by H. Bay *et al.* in 2006 [11]. This algorithm is both a detector and descriptor of keypoints from an image, and it can be used to track a set of keypoints from an image changing throughout time ([3] and [4]).

The SURF algorithm relies on the use of integral images. By using integral images for image convolution it is faster to compute than other features detection algorithms. Integral image is a data structure and algorithm to quickly and efficiently generate sum values in a rectangular subset of a grid. The value of each pixel *(x,y)* in the integral image $I_{\sum}$ is computed as follows ([7, 8] and [12]):

$$I_{\sum}(x,y) = \sum_{i=0}^{i\leqslant x}\sum_{j=0}^{j\leqslant y} I(i,j) \tag{1}$$

Then to find the sum of pixel values contained in a rectangle area $R_{\sum}$ between points (x_1,y_1) and (x_2,y_2) where $(x_1,y_1)\leqslant(x_2,y_2)$ the following equation is used ([7] and [12]):

$$R_{\sum}(x_1,y_1,x_2,y_2) = I_{\sum}(x_2,y_2) - I_{\sum}(x_2,y_1-1) - I_{\sum}(x_1-1,y_2) + I_{\sum}(x_1-1,y_1-1) \tag{2}$$

More concretely, SURF algorithm achieves its goal by executing two well-defined steps:

1. Find image keypoints.
2. Calculate keypoints descriptors.

In order to make image correspondences or object tracking, an additional step can be considered, although it is not part of the SURF algorithm. This step is *the matching step*.

2.1. Finding keypoints

In the SURF algorithm, *keypoints* or *points of interest* refer to a set of image pixels selected by their properties, location, and neighborhood structure, that characterize the image ([3–5] and [6]).

In order to detect keypoints, the SURF detector is based on *determinant* of the *Hessian matrix*. Given a point $p = (x, y)$ from an image I, the Hessian matrix $H(p,\sigma)$ in p at scale σ is defined as follows ([7] and [8]):

$$H(p,\sigma) = \begin{bmatrix} L_{xx}(p,\sigma) & L_{xy}(p,\sigma) \\ L_{xy}(p,\sigma) & L_{yy}(p,\sigma) \end{bmatrix} \tag{3}$$

DOI: 10.1002/cpe

In the Hessian matrix, L_{xx} (p,σ), L_{xy} (p,σ), and L_{yy} (p,σ) are approximations to the *convolution of the Gaussian second order derivative* with the image I in point p:

$$L_{xx}(p,\sigma) = I_{\sum}(p) * \frac{\partial^2}{\partial x^2} g(\sigma) \tag{4}$$

$$L_{xy}(p,\sigma) = I_{\sum}(p) * \frac{\partial^2}{\partial xy} g(\sigma) \tag{5}$$

$$L_{yy}(p,\sigma) = I_{\sum}(p) * \frac{\partial^2}{\partial y^2} g(\sigma) \tag{6}$$

Once the Hessian matrix has been defined, the determinant of the Hessian matrix can be calculated as follows:

$$Det(H) = L_{xx}(p,\sigma) * L_{yy}(p,\sigma) - (w * L_{xy}(p,\sigma))^2, \tag{7}$$

where w is a weight factor to compensate the approximation error.

2.2. Calculating keypoints descriptors

In order to identify or track objects, or complete scenes, keypoints are not enough. Besides keypoints, it is necessary to calculate *descriptors* of them. *Descriptors* label keypoints with the aim of identifying them in future matches and searches.

In order to calculate descriptors, orientation and neighborhood description is mainly considered. To obtain pixels orientation, the algorithm uses *Haar filters* that give a response of image points in x and y axes. In the response function, orientation is set to the treated pixel. In this way, the SURF manages to be a rotation invariant ([3, 4, 7] and [8]).

2.3. Matching descriptors and getting counterpart points

As previously said, although the matching step is not part of the SURF algorithm, when the goal is to identify objects or complete scenes between two different images, it is necessary to compare the calculated descriptors from the two source images. This step is often based on a distance between descriptors, for example, the Mahalanobis or Euclidean distance ([3, 4, 7, 8] and [13]).

3. RELATED WORK

The article's intention is to conduct a comparative study among different parallel implementations of the SURF algorithm.

SURF is one of the most frequently used algorithms when it is necessary to detect and describe a set of key points from an image. For this reason, we have chosen SURF for our comparative study. This study could be used as a reference, by anyone who needs to use SURF, to select the best parallel implementation.

Currently, we can find several sequential and parallel software implementations of this well-known computer vision algorithm. Regarding sequential versions, some comparative studies which measure runtime, stability of descriptors, and stability of detection among several implementations can be found ([14–16] and [17]). As for parallelism, we found a pseudo-study in which the author only compares with their own SURF implementation (written by using POSIX Threads) [18]. However, we were unable to access or download the source code presented in this article to compare it with other implementations. In conclusion, to the best of our knowledge, no similar studies for parallel software implementations exist.

The implementations selected for this comparative study are developed by using the most commonly used and best known parallel programming languages, for example, NVIDIA Compute

DOI: 10.1002/cpe

Unified Device Architecture (CUDA), OpenCL, and OpenMP. These implementations are the following:

- *GPUSURF 1.2.0*: is an implementation of SURF using CUDA. This version support both CUDPP-1.x and CUDPP-2.x. It was developed by A. Schulz *et al*., at Max Planck Institute Informatik, Germany [19].
- *parallelsurf 0.96*: is an implementation of SURF, which uses multithreading to speed up computation on multi-core machines. It is maintained by David Gossow and is based on the SURF implementation included in Pan-o-matic, written by Anael Orlinski [20].
- *Speeded Up SURF*: is an implementation of the SURF using NVIDIA CUDA. It was developed in the Autonomous Space Robotics Lab, at the University of Toronto ([21] and [22]).
- *GPUSurf_HA*: is an open-source and cross-platform implementation of GPUSURF. This implementation is based on [23], and it was developed by Henri Astre. For a detailed description see [24] and [25].
- *clsurf*: is an OpenCL implementation of SURF. It was developed by the NUCAR research group, at Northeastern University, and AMD ([26] and [27]).
- *Parallel OpenCV SURF*: is a SURF implementation developed by using the parallel classes given by OpenCV. These classes use CUDA to apply the parallelism in a GPU architecture. This implementation has been developed by us.

4. METHODOLOGY

In this section, we describe the methodology used to carry out the comparative study. The following subsections detail the requirements and features of the set of parallel implementations selected for the study, the hardware systems in which the implementations were executed and evaluated, and the metrics used for measuring and comparing the performance of the different implementations.

4.1. Parallel implementations details

We have tried to study and compare all the existing parallel implementations of SURF algorithm, but finally and because of the obsolete dependencies required by some of the implementations, we studied and compared only four of them. Additionally, 3 out of 4 are complete SURF implementations. To our best knowledge, no other parallel implementations exist. As we will see, the studied implementations are developed in some of the most used parallel programming languages. Table I shows all the details.

In Table I, columns from 1 to 6 refer to the different parallel implementations of the SURF algorithm selected for the study. Rows from 1 to 11 show the software dependencies for each parallel implementation. Rows 1 and 2 indicate if an implementation uses the parallel programming

Table I. Parallel implementations details.

Dependencies	GPUSURF 1.2.0	Parallelsurf 0.96	Speeded Up SURF	GPUSurf_HA	Clsurf	Parallel OpenCV SURF
nVidia CUDA	$\geqslant$ 2.2	x	3.0	3.0	x	$\geqslant$ 4.2.9
CUDPP lib	2.x	x	1.1	x	x	x
OpenCL	x	x	x	x	✓	x
OpenMP	x	✓	x	x	x	x
OpenCV	$\geqslant$ 2.0.0	x	2.1.0	x	$\geqslant$ 2.1.0	$\geqslant$ 2.4.3
OpenCV parallel classes	x	x	x	x	x	✓
Boost lib	x	✓	✓	x	x	x
Threadpool lib	x	✓	x	x	x	x
Ogre CUDA	x	x	x	✓	x	x
Ogre GPGPU	x	x	x	✓	x	x
O.S.	Linux x64	Linux	Ubuntu 9.04	Win & Linux	Linux	Ubuntu 12.10

DOI: 10.1002/cpe

language CUDA [28] and particularly the CUDPP library [29]. Row 3 specifies if a concrete implementation uses OpenCL [30] as parallel programming language. Row 4 indicates if OpenMP [31] is used as multi-core parallel programing language for any of the SURF developments. Rows 5 and 6 indicate if an implementation uses OpenCV [32], and particularly its parallel classes. Some of these implementations require using Boost and Threadpool libraries ([33, 34]). This feature is specified in rows 7 and 8. If any implementation uses Ogre [35] to show images instead of using OpenCV, it is specified in rows 9 and 10 (depending on the Ogre variant). Finally, row 11 indicates the operating system in which each implementation was tested and executed originally.

In this way, each row contains the requirements and characteristics for a particular parallel implementation. In order to complete this study, we consider necessary an implementation using the parallel classes given by OpenCV. After searching unsuccessfully for an implementation of this type, we decided to develop it. This implementation appears in the last column.

4.2. Hardware specifications

In order to conduct a more complete study, all tests applied to the parallel implementations of SURF were performed in two different hardware platforms. One of them consists of a standard hardware that any non-expert user can have currently, more particularly, a notebook ASUS series G55V, model G55VW. The other hardware platform is an advanced hardware used for high performance computing. By using several hardware platforms, we can study how both the runtime (Section 4.4.1) and the speedup (Section 4.4.4) of the algorithm vary when the parallel SURF implementations are executed in different hardware. In this way, we can obtain more complete conclusions.

4.2.1. Standard hardware specifications. This standard hardware is what a non-expert user in computation can have currently. For this study, this hardware consists of a notebook ASUS G55VW. Table II shows the hardware specifications.

As we can see in Table II, the system also incorporates a modern but standard graphic hardware. Table III shows the graphic hardware specifications.

4.2.2. Advanced hardware specifications. In order to evaluate the parallel versions of SURF in a more comprehensive way, we also execute both the *runtime speed* (Section 4.4.1) and the *speedup* (Section 4.4.4) tests in an advanced hardware used for high performance computing. This hardware

Table II. Standard hardware specifications - ASUS G55VW.

Parameter	Value
CPU	Intel Core i7 3630QM / 2.4 GHz
CPU released	Q3 2012
Cores	4
Memory	DDR3 1600 MHz SDRAM, 16 GB
Cache	6 MB L3
Graphic	NVIDIA GeForce GTX 660 M
Storage	128 GB SSD + 750 GB
O.S.	Ubuntu 12.10

Table III. Standard graphic hardware specifications - NVIDIA GEFORCE GTX 660 M.

Parameter	Value
CUDA cores	384
Frequency	835 MHz
Memory bandwidth	64.0 GB/s
Memory interface	GDDR5
Memory	2 GB
OpenGL	4.1

DOI: 10.1002/cpe

Table IV. Advanced hardware specifications.

Parameter	Value
CPU	4 x AMD Opteron 6176 / 2.3 GHz
CPU released	Q4 2010
Cores	4 x 12
Memory	DDR3 1333 MHz SDRAM, 64 GB
Cache	12 MB L3
Graphic	NVIDIA TESLA C2075
Storage	500 GB
O.S.	Ubuntu 12.10

Table V. Advanced graphic hardware specifications - NVIDIA TESLA C2075.

Parameter	Value
CUDA cores	448
Frequency	1.15 GHz
Memory bandwidth	144.0 GB/s
Memory interface	GDDR5
Memory	6 GB
OpenGL	4.1

also incorporates one of the most powerful GPU that we can find currently. Tables IV and V show a detailed description of the advanced hardware.

4.3. Image collection

For this study, we use two different sets of images. In the first place, for measuring *descriptor stability* (Section 4.4.2) and *detection stability* (Section 4.4.3), we use a collection of images widely used by other authors when evaluating the SURF algorithm for these parameters ([16, 17, 19, 27] and [8]). Figure 1 shows this image collection. We can see that these images have different properties (number of borders, number of corners, etc.) in order to perform a more complete study.

In the second place, and because of the parallelism which is really effective when data size and number of subtasks become high, we use a collection of Full-HD (1920p x 1080p) images for measuring *runtime speed* (Section 4.4.1) and *speedup* (Section 4.4.4). Each image of this collection has a high number of keypoints and descriptors in order to increase the number of subtasks in the parallel process of the SURF algorithm. Figure 2 shows the Full-HD image collection.

4.4. Metrics

For this study, in order to evaluate the performance and accuracy of all selected parallel implementations, we decided to apply to them the same tests that other authors applied to sequential implementations of the SURF algorithm ([17] and [12]), besides the *speedup* and *efficiency* metrics (typical in parallel systems). All tests were executed by using OpenCV library version 2.4.3, and CUDA SDK and CUDA toolkit version 4.2.9 (when they are applicable). In the following sections, we explain the different metrics and tests used.

4.4.1. Runtime speed. The runtime speed test measures the speed in detecting and describing the keypoints for each implementation. The process to measure them is described as follows:

1. Kill all extraneous processes.
2. Detect keypoints and compute descriptors while recording elapsed time.
3. Compute elapsed time 10 times and output best result.
4. Run the whole experiment 11 times for each implementation and record median time.

For a more detailed explanation of the runtime speed test, please see [17] and [12].

DOI: 10.1002/cpe

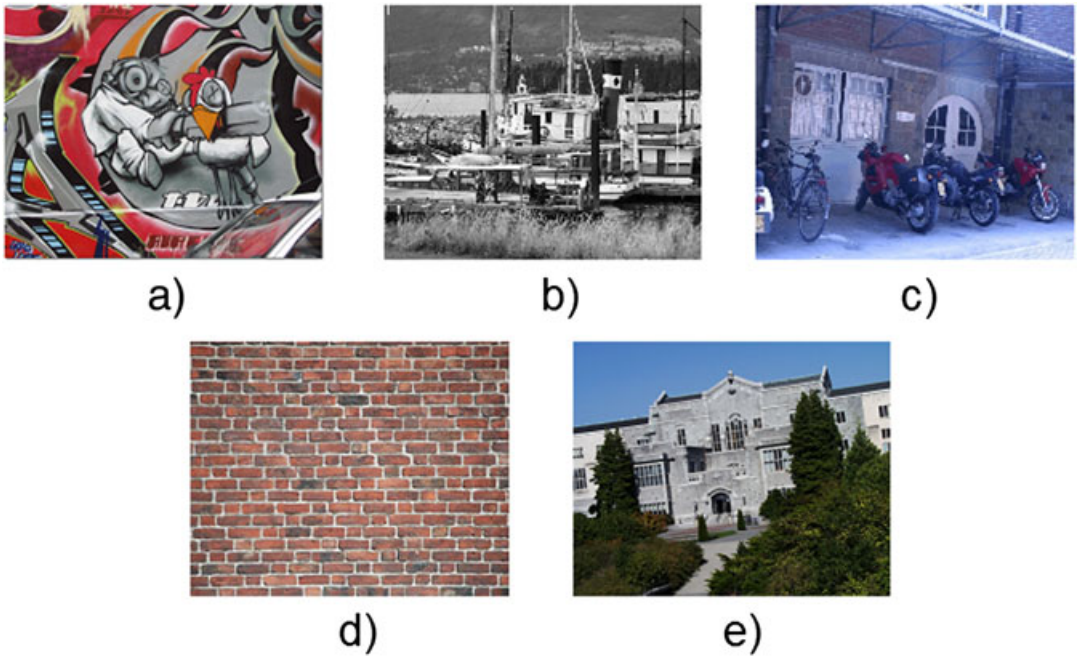

Figure 1. Image collection for measuring *descriptor stability* and *detection stability*: a) 'Graffiti' image; b) 'Boat Sequence' image; c) 'Bikes' image; d) 'Bricks' image; and e) 'UBC' image.

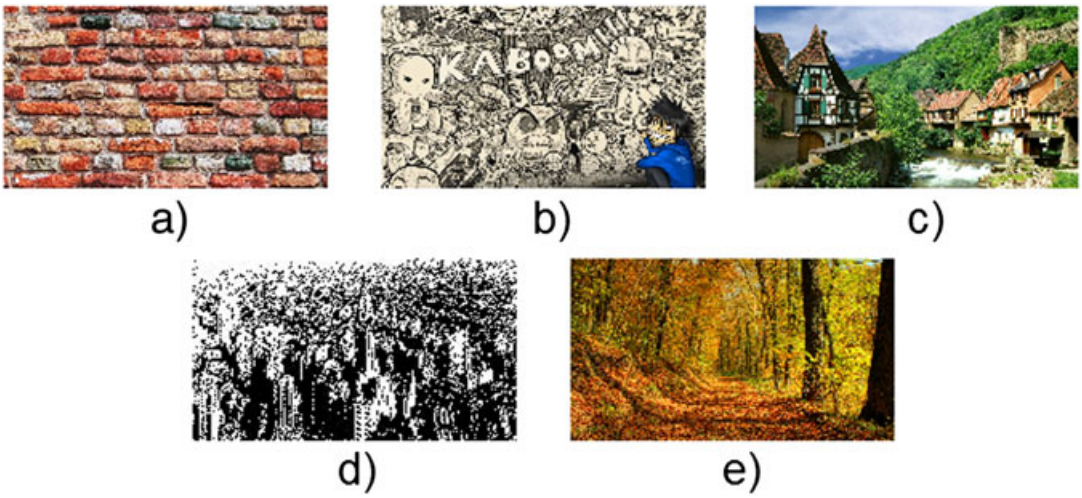

Figure 2. Full-HD image collection for measuring runtime speed and speedup: (a) 'Bricks_HD' image; (b) 'Graffiti_HD' image; (c) 'House_HD' image; (d) 'City_HD' image; and (e) 'Landscape_HD' image.

Because we have two different hardware with several cores each one (Section 4.2), a scalability study for the SURF version for multi-core (parallelsurf 0.96) has been also performed.

4.4.2. Descriptor stability. The descriptor stability test measures the stability based on the number of correct associations between two images from the data set, which have known transformations. The process to run this test is described as follows:

1. Detect features for each image.
2. For each image, compute a feature description for all features found.
3. In each image sequence, associate features in the first image to the N^{th} image, where $N > 1$.
4. Compute the number of correct associations.

DOI: 10.1002/cpe

Note that an image sequence refers to an original image and the resulting images after applying some known transformations. In this case, the transformations consist of a clockwise image rotation of 45° and two scaled images: one of them reducing the image at 50% and the other increasing it at 50%. Figure 3 shows an example of the geometric transformations applied to 'Graffiti' image.

For a more detailed explanation of the descriptor stability test, please see [17] and [12].

4.4.3. Detection stability. The detection stability test refers to how well a keypoint is detected after the image has undergone a transformation. For this reason, a good detector would detect a point in the same location as it was in the original image after applying a transformation. The process to calculate the detection stability is described as follows:

1. Detect points of interest in all images.
2. Transform interest points in image 1 to image N.
3. For each interest point in image 1:
   (a) Find all interest points in image N.
   (b) If the expected pixel location is outside the image, ignore.
   (c) If the number of matches is more than 1, ignore.
   (d) If the number of matches is one, mark the keypoint as a correct detection.
4. Count number of valid keypoints, which have 0 or 1 matches.
5. Detection metrics for each image in the sequence is the total number of correct detections divided by the number of valid keypoints.

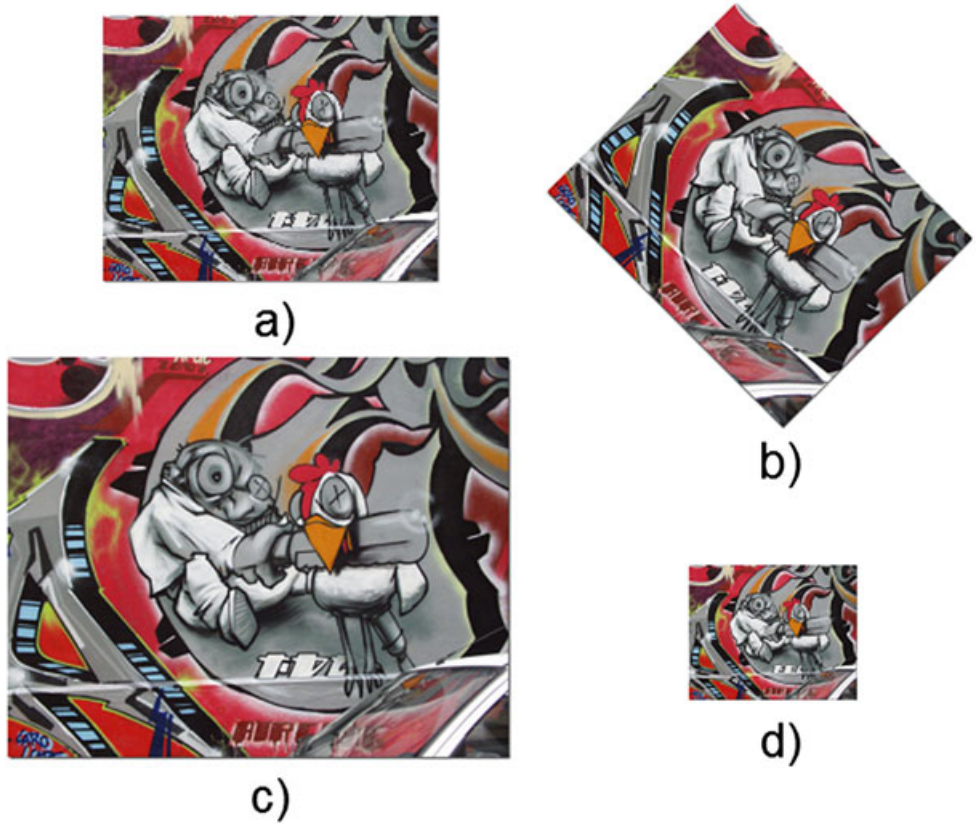

Figure 3. Geometric transformations applied to 'Graffiti' image: (a) original image; (b) clockwise rotation of 45°; (c) scaled image increasing 50% from original one; and (d) scaled image reducing 50% from original one.

DOI: 10.1002/cpe

Note that the image sequence has the same meaning as at descriptor stability section (Section 4.4.2), and the transformations applied to an image are the same too (see Figure 3 for an example).

For a more detailed description of the detection stability test, please see [17] and [12].

4.4.4. SpeedUp and efficiency. SpeedUp has as a goal to measure how many times a parallel implementation of an algorithm is faster than the sequential version.

The speedup for a particular algorithm is measured by using the following equation (Equation 8):

$$S = \frac{sequential\_execution\_time}{parallel\_execution\_time} \tag{8}$$

For measuring this, and because the selected versions are from different authors and are written by using very different languages, we used the slower sequential version of all parallel candidates. That is, we executed all the parallel implementations by using a single CPU core and then selected the slower version in terms of runtime. The worst results were obtained by using *parallelsurf 0.96*, so we selected it as the sequential version for measuring the speedup.

The efficiency may be defined as how effectively we are making use of multiple processors/cores. The efficiency is measured by using Equation 9.

$$E = \frac{SpeedUp}{number\_of\_cores} \tag{9}$$

At this point, and because we have two different hardware with several cores each one (see Section 4.2), a scalability study for the SURF version for multi-core (parallelsurf 0.96) has been performed.

5. RESULTS

In this section, we present and analyze the results obtained by applying the previous tests and metrics to the different parallel SURF implementations (by using the methodology described in Section 4).

Speeded Up SURF and *GPUSURF_HA* could not be evaluated because they are too old to run on the systems used for the study. The remaining implementations were executed and evaluated successfully. The following subsections present the obtained results.

5.1. Runtime speed results

In this section, we explain the results obtained by applying the *Runtime Speed* test to the parallel implementations of SURF. Furthermore, a scalability study for the multi-core CPU version of the SURF algorithm has been performed by increasing the number of cores to execute it. Note that SURF is a deterministic algorithm; this means that the number of cores used to execute it does not influence the results for both the descriptor stability (see Section 4.4.2) and detection stability (see Section 4.4.3). The following subsections show the results obtained for each hardware type (see Section 4.2).

5.1.1. Runtime speed results by using the standard hardware. As we said previously, a scalability study for the multi-core CPU version of SURF has been carried out. For this reason, the results appear separated into two different tables. On the one hand, Table VI shows the runtime speed and the scalability study for the multi-core version of SURF (*parallelsurf 0.96*). On the other hand, Table VII shows the results by applying the test in the standard GPU.

Note that the matching step is not part of the SURF algorithm, and for this reason, the measured times only take into account the keypoints detection and description steps.

As we can see, executions, which use *City_HD* image, require more time because this image has a higher number of keypoints due to the high number of lines, intersections among them, and corners.

DOI: 10.1002/cpe

Table VI. parallelsurf 0.96. Runtime speed and scalability results by using the standard CPU hardware.

Cores	Bricks_HD (ms)	Graffiti_HD (ms)	Landscape_HD (ms)	City_HD (ms)	House_HD (ms)	Average (ms)
1	2332.82	2715.78	2454.85	4074.04	2384.83	2792.47
2	1320.15	1443.00	1264.26	2149.04	1250.41	1485.37
4	781.56	850.98	721.96	1209.14	752.92	863.31

Table VII. Runtime speed results by using the standard GPU.

Implementation	Bricks_HD (ms)	Graffiti_HD (ms)	Landscape_HD (ms)	City_HD (ms)	House_HD (ms)	Average (ms)
GPUSURF 1.2.0	2325.66	2347.09	2341.25	2349.83	2351.02	2342.97
Parallel OpenCV SURF	753.31	785.34	712.41	815.13	628.08	738.85
clsurf	87.48	90.60	89.39	91.68	79.48	87.72

Table VIII. parallelsurf 0.96. Runtime speed and scalability results by using the advanced CPU hardware.

Cores	Bricks_HD (ms)	Graffiti_HD (ms)	Landscape_HD (ms)	City_HD (ms)	House_HD (ms)	Average (ms)
1	4729.28	5165.23	4465.86	7693.36	4436.24	5297.99
2	2574.81	2781.00	2440.64	4095.13	2411.56	2860.63
4	1405.43	1521.22	1333.26	2232.38	1315.69	1561.60
8	796.17	865.17	758.99	1275.40	741.60	887.47
16	523.07	569.28	496.63	832.72	492.53	582.85
32	485.68	512.90	446.34	762.97	458.74	533.32

Table IX. Runtime speed results by using the advanced GPU.

Implementation	Bricks_HD (ms)	Graffiti_HD (ms)	Landscape_HD (ms)	City_HD (ms)	House_HD (ms)	Average (ms)
GPUSURF 1.2.0	864.41	863.35	856.50	852.97	851.65	857.78
Parallel OpenCV SURF	219.74	232.83	200.94	234.55	162.14	210.04
clsurf	47.02	47.95	47.54	48.17	44.79	47.09

Regarding the best result in terms of GPU, it is obtained when using the *clsurf* implementation, followed by *Parallel OpenCV SURF*, and then by *GPUSURF 1.2.0*. Note that for the multi-core version (*parallelsurf 0.96*), when the number of used cores grows, we obtain similar values to the *Parallel OpenCV version* (GPU-based version). Despite this, the best results are obtained with the GPU-based implementation, in particular with *clsurf* (written in OpenCL). It is important to observe that *parallelsurf 0.96* only uses the CPU.

5.1.2. Runtime speed results by using the advanced hardware. In the same way as in Section 5.1.1, a scalability study for the multi-core CPU version of SURF has been performed in the advanced hardware too. Again, the results appear separated into two different tables. Table VIII shows the runtime speed and the scalability study for the multi-core version of SURF (*parallelsurf 0.96*), and Table IX shows the results by applying the test to the GPU-based implementations in the advanced GPU.

As said, the matching step is not part of the SURF algorithm, and for this reason, the measured times only take into account the keypoints detection and description steps.

Note that the sequential times by using the advanced hardware for the multi-core version of SURF, *parallelsurf 0.96*, are greater than the values obtained when the standard hardware is used. This is due to the technological age (see Tables II and IV) of each one and because the performance obtained by using Intel hardware, in this case, is higher than when using AMD hardware.

DOI: 10.1002/cpe

In terms of results, a similar behavior to the one obtained by using standard hardware occurs; the best is obtained when using the *clsurf* implementation, followed by *Parallel OpenCV SURF*, and then by *GPUSURF 1.2.0* (GPU-based implementations). For the multi-core version (*parallelsurf 0.96*), when the number of used cores grows, we obtain similar, or even lesser, values than the results obtained with *GPUSURF 1.2.0* (GPU-based version). Despite this, the best results are obtained with the GPU-based implementations, in particular with *clsurf* again. As in the standard hardware, *parallelsurf 0.96* (based on OpenMP and only CPU usage) has longer runtimes.

5.2. Descriptor stability results

In this section, we explain the results obtained by applying the *descriptor stability* test to the parallel implementations of SURF.

Descriptor stability was always measured by using as source the original image and as destination all the variants of this original image (all the geometric transformations and also the own original image, see Section 4.4.2). As for *clsurf* implementation, it could not be measured because this only executes the first two steps of the SURF algorithm.

With regard to the results, Table X shows that the best implementation in terms of descriptor stability is *parallelsurf 0.96* (based on OpenMP), followed by the GPU-based implementations: *GPUSURF 1.2.0*, and then *Parallel OpenCV SURF*.

5.3. Detection stability results

In this section, we explain the results obtained by applying the *detection stability* test to the parallel implementations of SURF.

Table XI shows that the best implementation in terms of detection stability is *parallelsurf 0.96*, followed (closely) by *GPUSURF 1.2.0*, and finally by *Parallel OpenCV SURF* implementation. As for *clsurf*, it was not possible to measure it at this point because it only executes the first two steps of the SURF algorithm. In conclusion, again, from the stability point of view the OpenMP-based implementation obtains better results than the GPU-based implementations.

5.4. SpeedUp and efficiency results

In this section, we explain the results obtained by calculating both the *speedup* and the *efficiency* (only for the multi-core CPU implementations) of the parallel SURF implementations. The *speedup* and *efficiency* are calculated by applying equations 8 and 9, respectively. For this calculation, we

Table X. Descriptor stability results.

Implementation	Graffiti images (%)	Boat S. images (%)	Bikes images (%)	Bricks images (%)	UBC images (%)	Average (%)
parallelsurf 0.96	83.30	86.63	87.99	87.87	89.30	87.01
GPUSURF 1.2.0	46.00	25.90	45.03	28.85	44.27	38.25
Parallel OpenCV SURF	12.51	31.87	7.98	68.44	13.68	26.89
clsurf	*	*	*	*	*	*

Table XI. Detection stability results.

Implementation	Graffiti images (%)	Boat S. images (%)	Bikes images (%)	Bricks images (%)	UBC images (%)	Average (%)
parallelsurf 0.96	99.35	99.25	99.24	99.59	99.62	99.41
GPUSURF 1.2.0	98.21	99.01	99.52	98.51	99.39	98.93
Parallel OpenCV SURF	81.98	75.92	59.78	76.97	70.46	73.02
clsurf	*	*	*	*	*	*

DOI: 10.1002/cpe

take into account the results presented in Section 5.1 (runtime test). The following subsections show the results obtained for both the standard and advanced hardware.

5.4.1. SpeedUp and efficiency by using the standard hardware. In this subsection, the results obtained by calculating the speedup for the standard hardware are presented. Table XII shows the speedup and efficiency for the multi-core CPU version of the SURF algorithm (*parallelsurf 0.96*). Table XIII shows the speedup for the GPU-based implementations of the algorithm.

In terms of parallelism in the multi-core CPU implementation (*parallelsurf 0.96*), for all the cases, the *speedup* and *efficiency* have good values when the number of cores grows (*efficiency* is always higher than 80%). Furthermore, the results are similar for all the images of the collection.

For the GPU-based implementations, except for *GPUSURF 1.2.0*, the *speedup* is higher in comparison with the values calculated for the CPU-based implementation (*parallelsurf 0.96*). For all GPU-based implementations, the maximum speedup is obtained with the *City_HD* image, which is the image of the collection with more keypoints and descriptors.

5.4.2. SpeedUp and efficiency by using the advanced hardware. In this subsection, the results for both the *speedup* and *efficiency* in the advanced hardware are presented. Again, results appear divided into two different tables. Table XIV shows the *speedup* and *efficiency* for the multi-core version of the SURF algorithm (*parallelsurf 0.96*).

The *speedup* and *efficiency* are very similar to the ones obtained for the standard hardware for 1, 2, and 4 cores. Good values of these metrics are also obtained for higher number of cores. However, with the highest number of cores, the results obtained for *speedup* and *efficiency* are not so good.

In terms of parallelism, in multi-core CPU implementation, the highest *speedup* is reached for the image *City_HD*, which is the image with more points of interest.

Table XV shows the *speedup* for the GPU-based implementations of the algorithm.

Table XII. parallelsurf 0.96. SpeedUp and efficiency by using the standard CPU hardware.

Cores	Bricks_HD (%)	Graffiti_HD (%)	Landscape_HD (%)	City_HD (%)	House_HD (%)	Average SpeedUp (%)	Average Efficiency (%)
1	100.00	100.00	100.00	100.00	100.00	100.00	100.00
2	176.71	188.20	194.17	189.58	190.72	187.88	93.94
4	298.48	319.14	340.02	336.94	316.74	322.26	80.57

Table XIII. SpeedUp by using the standard GPU.

Implementation	Bricks_HD (%)	Graffiti_HD (%)	Landscape_HD (%)	City_HD (%)	House_HD (%)	Average (%)
GPUSURF 1.2.0	100.31	115.71	104.85	173.38	101.44	119.14
Parallel OpenCV SURF	309.68	345.81	344.58	499.80	379.70	375.92
clsurf	2666.77	2997.77	2746.12	4443.72	3000.72	3171.00

Table XIV. parallelsurf 0.96. SpeedUp and efficiency by using the advanced CPU hardware.

Cores	Bricks_HD (%)	Graffiti_HD (%)	Landscape_HD (%)	City_HD (%)	House_HD (%)	Average SpeedUp (%)	Average Efficiency (%)
1	100.00	100.00	100.00	100.00	100.00	100.00	100.00
2	183.68	185.73	182.98	187.87	183.96	184.84	92.42
4	336.50	339.55	334.96	344.63	337.18	338.56	84.64
8	594.01	597.02	588.40	603.21	598.20	596.17	74.52
16	904.14	907.33	899.24	923.88	900.70	907.06	56.69
32	973.75	1007.07	1000.55	1008.35	967.06	991.35	30.98

DOI: 10.1002/cpe

Table XV. SpeedUp by using the advanced GPU.

Implementation	Bricks_HD (%)	Graffiti_HD (%)	Landscape_HD (%)	City_HD (%)	House_HD (%)	Average (%)
GPUSURF 1.2.0	547.11	598.28	521.41	901.95	903.35	694.42
Parallel OpenCV SURF	2152.18	2218.50	2222.44	3280.01	2736.03	2521.83
clsurf	10057.80	10772.61	9393.25	15972.83	9904.79	11220.25

For the GPU-based implementations, again, the *speedup* is much higher in comparison with the values calculated for the CPU-based implementation (*parallelsurf 0.96*), except for the *GPUSURF 1.2.0* implementation in which the *speedup* can be exceeded by the multi-core CPU version when the number of cores used for executing it is greater than or equal to 16. For this advanced GPU, the results are much better in comparison with the standard GPU. Anyway, the GPU-based implementations can be ordered in the same way: first *clsurf*, followed by *Parallel OpenCV SURF*, and then *GPUSURF 1.2.0*.

6. CONCLUSIONS

In this paper, we have presented a comparative study of all the software parallel implementations of the SURF algorithm that we have found. SURF is an important algorithm in computer vision and robotics, and its parallel implementations are based on very different techniques: CUDA, OpenMP, OpenCL, and so on. In order to conduct this study, we have used standard metrics and image collections.

As we can see throughout the study, in general, the execution time increases with the number of keypoints of each image of the collection. For example, the '*City_HD*' is the image with more features, and '*House_HD*' is the image with fewer features, this affecting their runtimes (see Section 5.1).

Regarding *descriptor stability* and *detection stability*, the best results are obtained by *parallelsurf 0.96* implementation. These results depend on how the features detector and matching process were developed, and note that these processes can vary among the different implementations.

As for *Runtime Speed*, the best result is obtained when executing *clsurf*, which uses OpenCL language. Regarding *SpeedUp* metric, the best result was obtained by *clsurf* as expected after the results obtained for *Runtime Speed* test. For these two metrics, parallel implementations executed on GPUs are better than the parallel implementations which use multithreading techniques in a multi-core CPU.

As a general conclusion, this study shows that the faster parallel implementations are less stable, and vice versa; the more stable parallel implementations are slower. More concretely, to apply the SURF algorithm to a specific problem, those implementations with better *descriptor stability* and better *detection stability* should be selected when the execution time is not so important. In this case, we should use *parallelsurf 0.96* (based on OpenMP). However, if the execution time is a very important parameter, we should use one of the GPU-based implementations (particularly, *clsurf*), that is, a faster implementation but with lesser stability in the algorithm. All this will depend on the particular problem and the requirements to fulfill.

ACKNOWLEDGEMENTS

This work was partially funded by the Projects of Excellence from the Junta de Andalucía (Spain) ROMOCOG I and ROMOCOG II (TEP-4479 and TEP-6412). The work also was partially funded by the Spanish Ministry of Economy and Competitiveness and the ERDF (European Regional Development Fund), under the contract TIN2012-30685 (BIO project).

DOI: 10.1002/cpe

REFERENCES

1. Nixon M, Aguado A. *Feature Extraction & Image Processing for Computer Vision*, Thirth edn. Academic Press: Waltham, Massachusetts, 2012.
2. Demaagd K, Oliver A, Oostendorp N, Scott K. *Practical Computer Vision with SimpleCV: The Simple Way to Make Technology See*. O'Reilly Media: Sebastopol, CA, 2012.
3. Bay H, Ess A, Tuytelaars T, Van Gool L. Speeded-Up Robust Features (SURF). *Computer Vision and Image Understanding* June 2008; **110**(3):346–359.
4. Pedersen JT. Study group SURF: Feature detection and description. *Technical Report*, Aarhus University, 2011.
5. Mikolajczyk K, Schmid C. Indexing based on scale invariant interest points. *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on,* vol.1, Vancouver, BC, Canada, 2001; 525–531.
6. Mikolajczyk K, Schmid C. An affine invariant interest point detector. *In Proceedings of the 7th European Conference on Computer Vision*, Copenhagen, Denmark, 2002; 128–142.
7. Evans C. Notes on the opensurf library. *Technical Report*, University of Bristol, 2009.
8. Oyallon E, Rabin J. An Analysis and Implementation of the SURF Method, and its Comparison to SIFT. *IPOL Journal - Image Processing On Line, ISSN 2105-1232,* preprint February 2013.
9. Fung J, Mann S. Using graphics devices in reverse: GPU-based Image Processing and Computer Vision. *Multimedia and Expo, 2008 IEEE International Conference on*, Hannover, Germany, 2008; 9–12.
10. Slabaugh G, Boyes R, Yang X. Multicore image processing with OpenMP [Applications Corner]. *Signal Processing Magazine, IEEE* March 2010; **27**(2):134–138.
11. Bay H, Tuytelaars T, Van Gool L. SURF: Speeded Up Robust Features. *Computer Vision-ECCV 2006* 2006; **3951**(3):404–417.
12. Abeles P. Resolving implementation ambiguity and improving SURF. *CoRR* 2012; **abs/1202.0492**.
13. Maesschalck RD, Jouan-Rimbaud D, Massart D. The Mahalanobis distance. *Chemometrics and Intelligent Laboratory Systems* 2000; **50**(1):1–18.
14. Gossow D, Decker P, Paulus D. An Evaluation of Open Source SURF Implementations. In *Robocup 2010: Robot Soccer World Cup XIV*, Vol. 6556, Ruiz-del Solar J, Chown E, Plger PG (eds), Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011; 169–179.
15. Maji S. A comparison of feature descriptors, 2006. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.141.4980.
16. Khvedchenia I. Feature descriptor comparison report, 2011. Available from: http://computer-vision-talks.com/2011/08/feature-descriptor-comparison-report/.
17. Abeles P. Performance: SURF, 2012. Available from: http://boofcv.org/index.php?title=Performance:SURF.
18. Zhang N. Computing Parallel Speeded-Up Robust Features (P-SURF) via POSIX Threads. In *Emerging Intelligent Computing Technology and Applications*, Vol. 5754, Huang DS, Jo KH, Lee HH, Kang HJ, Bevilacqua V (eds), Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009; 287–296.
19. Schulz A, Jung F, Hartte S, Trick D, Wojek C, Schindler K, Ackermann J, Goesele M. CUDA SURF. A real-time implementation for SURF, March 2012. Available from: http://www.d2.mpi-inf.mpg.de/surf.
20. Gossow D. ParallelSURF, 2010. Available from: http://sourceforge.net/apps/mediawiki/parallelsurf/.
21. Furgale P, Tong CH, Kenway G. Speeded-Up Speeded-Up Robust Features. *Technical Report*, University of Toronto, 2009.
22. Furgale P, Tong CH. Speeded up SURF, 2010. Available from: http://asrl.utias.utoronto.ca/code/gpusurf/index.html.
23. Cornelis N, Van Gool L. Fast scale invariant feature detection and matching on programmable graphics hardware. *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, Anchorage, AK, USA, 2008 june; 1–8.
24. Astre H. GPU-Surf with OGRE3D. Open-source implementation of Surf on GPU. *Technical Report*, Visual Experiments, 2010.
25. Astre H. GPUSurf, 2010. Available from: http://www.visual-experiments.com/demos/gpusurf/.
26. Mistry P, Gregg C, Rubin N, Kaeli D, Hazelwood K. Analyzing program flow within a many-kernel OpenCL application. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4. ACM: New York, NY, USA, 2011; 10:1–10:8.
27. NUCAR, AMD. clsurf. OpenCL implementation of the Speeded Up Robust Features (SURF) algorithm, 2011. Available from: http://code.google.com/p/clsurf/.
28. Sanders J, Kandrot E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley: Boston, Massachusetts, 2011.
29. CUDPP - CUDA Data Parallel Primitives Library, 2011. Available from: http://code.google.com/p/cudpp/.
30. Munshi A, Gaster B, Mattson T, Ginsburg D. *OpenCL Programming Guide*, OpenGL Series. Pearson Education: Upper Saddle River, New Jersey, 2011.
31. Chapman B, Jost G, Van Der Pas R. *Using OpenMP: Portable Shared Memory Parallel Programming*, Scientific and Engineering Computation Series. Mit Press: Cambridge, Massachusetts, 2008.
32. Bradski G, Kaehler A. *Learning OpenCV: Computer Vision with the OpenCV Library*, Software that sees. O'Reilly Media: Sebastopol, CA, 2008.
33. Schäling B. *The Boost C++ Libraries*. XML Press: California, USA, 2011.
34. Henkel P. Threadpool Library, 2008. Available from: http://threadpool.sourceforge.net/index.html.
35. Kerger F. *OGRE 3D 1.7 Beginner's Guide*, Learn by doing : less theory, more results. Packt Publishing, Limited: Birmingham, 2010.

DOI: 10.1002/cpe