Journal of Real-Time Image Processing
https://doi.org/10.1007/s11554-018-0755-2

ORIGINAL RESEARCH PAPER

Speeding up spatiotemporal feature extraction using GPU

Ahmed Mehrez[1] · Ahmed A. Morgan[2,3] · Elsayed E. Hemayed[2]

Received: 19 July 2017 / Accepted: 16 January 2018

Abstract
Spatiotemporal feature extraction algorithms are widely used in many image processing and computer vision applications. They are favored because of their robust generated features. However, they have high computational complexity. Parallelizing these algorithms, in order to speed their execution up, is of great importance. In this paper, we propose new parallel implementations, using GPU computing, for the two most widely used spatiotemporal feature extraction algorithms: scale-invariant feature transform and speeded up robust features. In our implementations, we solve problems with previous parallel implementations, such as load imbalance, thread synchronization, and the use of atomic operations. Our implementations speed up the execution by simultaneously processing all the work of each stage of the two algorithms, without dividing that stage into smaller sequential ones. The allocation of the threads in our implementations further allows them to increase the occupancy of the GPU streaming multiprocessors (SMs). We compare our presented implementations to previous CPU and GPU parallel implementations of the two algorithms. Results show that the proposed implementations could do all the processing in real time with high accuracy. They further achieve higher speedup, frame rate, and SM occupancy than the previous best-known parallel implementations of the two algorithms.

Keywords CUDA · Graphics processing unit (GPU) · Image matching · Scale-invariant feature transform (SIFT) · Speeded up robust features (SURF)

1 Introduction

The robust representation of image features is fundamental to most machine vision and image registration applications. Local approaches, which do not require image segmentation, are proved to be robust to changes that may occur in images, such as the change in the illumination and the view angle [1]. According to the changes in the image environment, local feature extraction approaches should overcome two main challenges. The first challenge is how to locate regions of the image that have different features and how to detect these features. The second challenge is how to describe the detected features in a unique way, which could be used to find a match with similar features in other images. Spatiotemporal features are shown to represent robust ones against the many variations in the image environment [2]. Accordingly, they become widely used in most image processing and computer vision applications.

Scale-invariant feature transform (SIFT) [1] and speeded up robust features (SURF) [3] are among the most robust spatiotemporal local feature extraction algorithms that are used in many computer vision techniques. For example, they are used in object recognition and tracking, image classification, face authentication, and video event classification. They could overcome illumination, scale, and rotation variations. SIFT and SURF extract features in the form of interest points, which represent special points in the image that could be used in image matching. A feature vector is created for each point, which describes the gradients in the region around that point.

✉ Ahmed Mehrez
ahmed.mehrez@feng.bu.edu.eg

Ahmed A. Morgan
ahmorgan@eng.cu.edu.eg; aamorgan@uqu.edu.sa

Elsayed E. Hemayed
hemayed@ieee.org

1 Department of Electrical Engineering, Faculty of Engineering at Shoubra, Benha University, Cairo 11614, Egypt

2 Department of Computer Engineering, Faculty of Engineering, Cairo University, Giza 12613, Egypt

3 College of Computers and Information Systems, Umm Al-Qura University, Makkah, Saudi Arabia

The SIFT algorithm uses Gaussian filters with increasing σ. As the filter size increases, the required computations and the execution time of the algorithm increase as well. This long computational time prevents the sequential implementation of the SIFT algorithm from being used in real-time applications. Alternatively, in order to use the SIFT algorithm in these real-time applications, some implementations use smaller Gaussian filters [4]. This unfortunately results in poor-quality interest points. On the other hand, the SURF algorithm is proposed to overcome the computational time problem of SIFT. The feature detection of the SURF algorithm is based on Hessian matrix and box filters approximation. Therefore, the SURF algorithm has fixed computational time throughout all points. In turn, the computation time of the SURF algorithm allows it to detect and describe interest points faster than SIFT. However, its sequential implementation is still far from being efficiently used in real-time applications.

In order to enable the use of SIFT and SURF algorithms in real-time applications, many parallel implementations of the two algorithms are presented using different hardware architectures [5–9]. These implementations, which are surveyed in Sect. 3, managed to provide a good speedup with respect to sequential implementations of the two algorithms. Nevertheless, they still have some problems that should be handled, in order to better enhance the performance of the two algorithms. First, most of previous parallel implementations suffer from a load imbalance problem. In each stage of the two algorithms, the computations are distributed in an imbalanced fashion over the processing elements (PEs) of the employed hardware architecture. Consequently, the heavily loaded PEs would need more time to finish its work than the less loaded ones. Indeed, a more balanced distribution of the work would result in a higher speedup. Second, most of the previous parallel implementations suffer from a thread synchronization problem. Computations of different stages of the two algorithms are split into smaller segments, which could be processed in parallel. However, each segment has to wait until the previous one finishes. With the aforementioned load imbalance problem, many PEs are left idle and the hardware occupancy deteriorates. This, in turn, prevents previous parallel implementations from achieving the maximum possible speedup. Finally, previous parallel implementations store interest points in a sequential manner, which depress the overall speedup that could be achieved by the parallel implementation. In this paper, we present a new parallel implementation that targets these problems. In other words, our implementation better handles the load imbalance and the synchronization between threads. It also increases the GPU occupancy and stores the detected interest points in a more efficient parallel manner.

Graphics processing units (GPUs) are mainly used for graphics. In this paper, we target NVIDIA GPUs, and hence, we would employ its terminology. A GPU consists of tens or hundreds of scalar processors (SPs) that are grouped to form streaming multiprocessors (SMs). Therefore, GPUs are cheap platforms that could result in a significant performance enhancement, if their parallelism is properly exploited. NVIDIA further develops CUDA, as a parallel computing platform, to facilitate the programming of their GPUs. This motivates many researchers to employ GPUs in speeding up general-purpose applications [10]. Interested readers would find dozens of such research work in [11].

The enormous embedded parallelism and the ease of programming encourage some researchers to speed up SIFT and SURF algorithms by using GPUs. Generally, in order to get the maximum performance from a GPU, its occupancy should be maximized. In other words, the number of threads at any time during the execution should be maximized. This maximization is indeed constrained by the GPU available resources, like registers and shared memory. However, previous GPU parallel implementations of SIFT and SURF suffer from the same problems that we mentioned in our previous paragraph. This, in turn, prevents them from reaching the maximum number of threads and an attainable speedup is therefore lost. In this paper, we seek this attainable speedup by better increasing the GPU occupancy. To this end, our contributions are two fold.

(a) Presenting a new GPU parallel implementations of both SIFT and SURF algorithms. Our implementation increases the GPU occupancy by minimizing load imbalance, thread synchronization, and sequential execution throughout all stages of the two algorithms.

(b) Evaluating the presented solution by comparing it to the sequential and the best-known parallel implementations of the two algorithms. Four GPUs and images of different resolutions and qualities are used to accurately validate the efficiency of the presented solution.

The rest of this paper is organized as follows: Sect. 2 explains the sequential implementation and the main stages of both SIFT and SURF spatiotemporal algorithms. Section 3 surveys the related work and discusses previous parallel implementations of the two algorithms. Section 4 introduces modern GPUs architecture with its basic concepts and terminology. Section 5 presents our methodology for parallelizing the two algorithms and details our implementation in different stages of the algorithms. Section 6 gives our experimental results and validates our implementation by comparing it to sequential and other related parallel algorithms. Finally, Sect. 7 concludes our work and gives directions for possible future work.

2 Interest point feature extraction algorithms

In this section, interest point extraction algorithms are discussed. In any image, edges and corners are the most likely locations where interest points may be found. Therefore, the first step in interest point detection is to identify edges and corners in a given image. Accordingly, interest points extraction algorithms could be simply considered as edge or corner detectors.

2.1 SIFT algorithm

The SIFT algorithm consists of four stages: scale space construction, interest point detection, orientation assignment, and interest point description. The first two stages are responsible for interest point localization, whereas the other two stages build the feature vector for the detected interest points, based on local gradients around these points.

Stage 1 scale space construction
This stage first constructs all scales in which interest points could be found. Scales are formed by convolving the original image by a Gaussian filter with increasing sigma (σ). Scale space consists of a number of octaves and each octave consists of a number of scales. The first octave is formed by convolving the original image. Thereafter, each subsequent octave uses downsampled images from its predecessor. Finally, the difference of Gaussians (DOG) is calculated. As shown in Fig. 1, the difference between each two consecutive layers, in each octave, constitutes one layer in the DOG.
Stage 2 interest points detection
In this stage, all points, in the DOG, are compared to their neighbors to detect whether they are local maxima or not. As shown in Fig. 2, any point is only considered

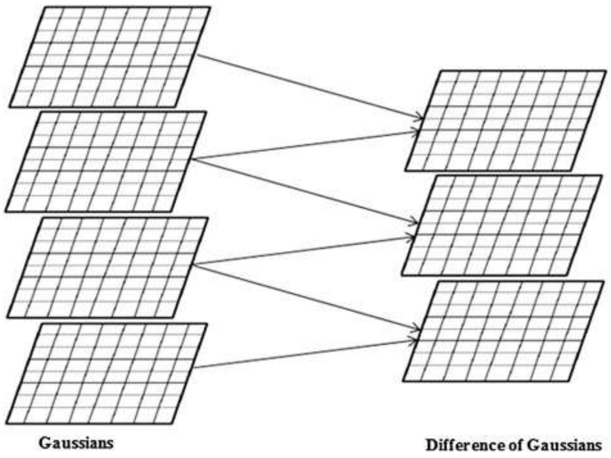


Fig. 1 Construction of difference of Gaussians (DOG) images

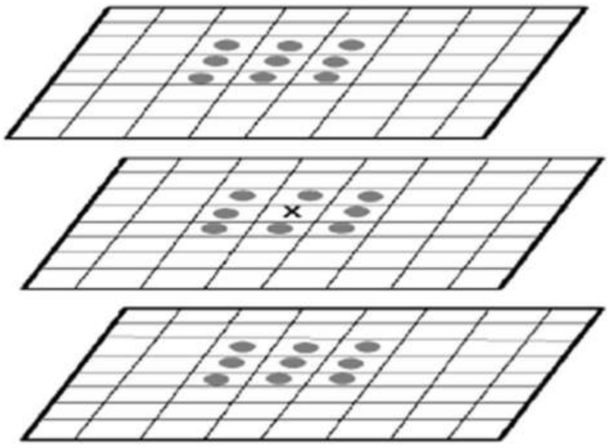


Fig. 2 Point comparisons to detect maxima of the DOG (gray dots represent the 26 neighbors of point x)

an interest point if it is a local maximum among its 26 neighbors. Thereafter, points are interpolated to sub-pixel accuracy. Points with low contrast are removed, and their responses at edges are eliminated. This method is developed by Brown and Lowe in [12] and is further used by D. Lowe in [1]. The extremum, $\hat{x}$, which represents the local maxima, is first calculated according to (1).

$$\hat{x} = -\frac{\mathrm{d}^2 D^{-1}}{\mathrm{d}x^2}\frac{\mathrm{d}D}{\mathrm{d}x}, \tag{1}$$

where D is the scale space function shifted to the sample point. D and its derivatives are calculated at the arbitrary sample point (x, y).
The approximation of the derivatives, $D(\hat{x})$, is then calculated using the difference of neighbors. According to (2), low-contrast points are removed [1]. A threshold of 0.03 is used to detect whether any point is of low contrast or not.

$$D(\hat{x}) = D + \frac{1}{2}\frac{\mathrm{d}D^t}{\mathrm{d}x}\hat{x}, \tag{2}$$

where D is again the scale space function shifted to the sample point, D^t is the function calculated at the offset from this point, and $\hat{x}$ is the extremum, as calculated in (1).
Stage 3 orientation assignment
After detecting interest points in all scales, the SIFT algorithm tries to detect the orientation of these points, based on the neighbors' gradient in the circular region around them. An orientation histogram is created for each point with 36 pins. These pins represent the 360° in the circular region. Gradients of neighbors are then added to the orientation histogram, after being weighted with a Gaussian weighted window of width 6σ, where $\sigma = 1.5 \times$ the scale of the point. Thereafter, the

orientation histogram is checked for the pin with the maximum value. This pin represents the dominant direction of the interest point. Therefore, it is assigned as its orientation. Finally, Lowe states that if another pin has a value that is more than 80% of the dominant one, it could be saved as a new interest point, with the same scale and location, but different orientation [1].

Stage 4 descriptor construction

In this final stage, the SIFT algorithm builds the feature vector. The feature vector describes the detected interest points, and hence, it is also called the descriptor vector. By comparing each interest point to its neighbors, this vector actually shows the intensity distribution around this point. For each interest point, a descriptor window of size 16×16 is formed, as shown in Fig. 3. This window is rotated to the point orientation. Gradients of other points around the interest point are then calculated. These gradients are shown as arrows in the small squares to the left of Fig. 3. Gradients are weighted with a Gaussian kernel with $\sigma = 1.5 \times$ the scale of the point. Thereafter, the descriptor window is divided into 4×4 subregions, which consist of 4×4 points. As shown to the right of Fig. 3, each region creates a histogram of 8 pins to sum the weighted gradients of its points. Accordingly, 4×4 subregions, with 8 pins each, result in a 128-pin descriptor vector.

2.2 SURF algorithm

The SURF algorithm consists of five main stages. In the first stage, the integral image is computed from the original image. In the second stage, the scale space is constructed by applying an increasing size filter. In the third stage, interest points are detected. In the fourth stage, the orientation of these interest points is found. Finally, in the fifth stage, the interest points are described. The following subsections discuss these stages in more detail.

Stage 1 integral image computation

SURF uses integral images to speed up the filters calculation. The integral image is an image whose points save the sum of its value and the values of all its upper left points. Accordingly, the value of each point could be calculated according to (3).

$$I(x,y) = \sum_{\substack{y' \leq y \\ x' \leq x}} i(x', y'), \tag{3}$$

where I is an arbitrary point, located at (x, y), and i represents points to its upper left, such that x' and y' are less than or equal to x' and y', respectively. By having each point accumulating the values of its upper left points, the summation of any rectangular area of the integral image could then be calculated by manipulating values of only four points. These are the points located at the corners of this rectangular area. Therefore, integral images reduce the number of computations required to manipulate any portion of the image to only four [3]. On the other hand, SURF uses Haar response, which is a discrete representation of the wavelet changes in the image [13]. Haar response and gradients of the image are consequently calculated using box filter approximations. Box filter approximations represent the second-order Gaussian derivatives, D_{xx}, D_{yy}, and D_{xy}, in x-, y-, and xy-directions, respectively. These approximations depend on the weighted sum of image portions, which could easily be calculated using integral images. Figure 4 shows an example of a 9×9 box filter. Approximations of second-order Gaussian derivatives, D_{xx}, D_{yy}, and D_{xy}, are shown in sub-figures (a), (b), and (c), respectively. For example, in sub-figure (a), the filter neglects upper

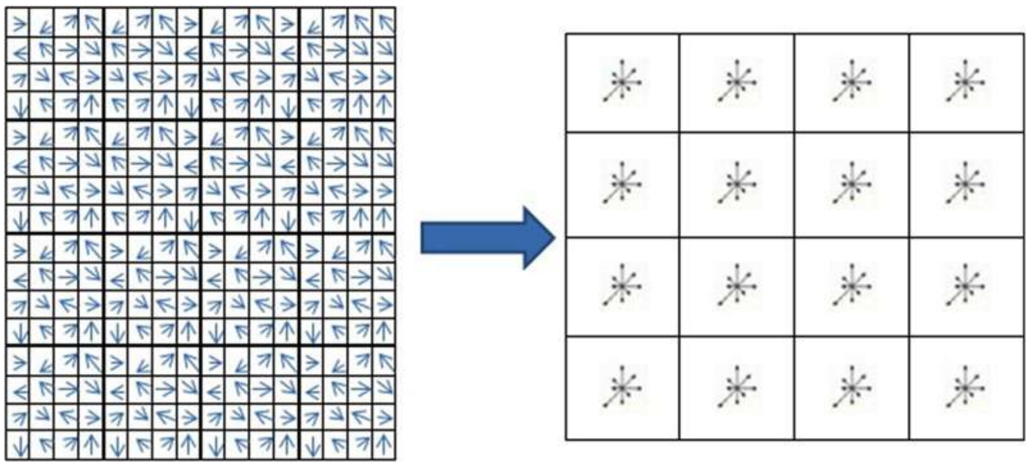


Fig. 3 Descriptor construction in the SIFT algorithm

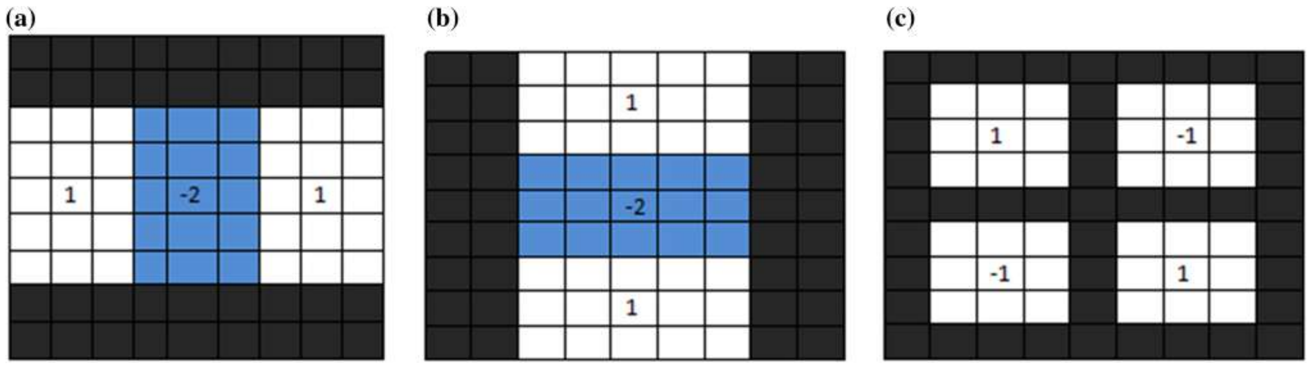


Fig. 4 Weighted box filter approximations, for a 9 × 9 filter

and lower 2 pixels. The summation of middle pixels, which are colored blue, is weighted by − 2. The summation of other pixels, which is colored white, is weighted by 1. The final value is the summation of all weighted sums.

Stage 2 scale space construction

As in the SIFT algorithm, the scale space is constructed at many octaves and scales. However, to generate new scales, SURF uses box filters with an increasing size, instead of changing the image size in each octave. Filter sizes increase both within and across octaves. This finally forms a stack of scales in the image pyramid. As shown in Fig. 5, the filter size increases from 9 × 9, in the first scale of the first octave, to 99 × 99, in the last scale of the third octave [3].

As mentioned in the previous subsection, box filter approximations result in the second-order Gaussian derivatives in *x*-, *y*-, and *xy*-directions. Accordingly, for any arbitrary point, P, located at (*x*, *y*), the Hessian matrix for point *P* at scale *σ*, *H (P, σ)*, could be defined according to (4).

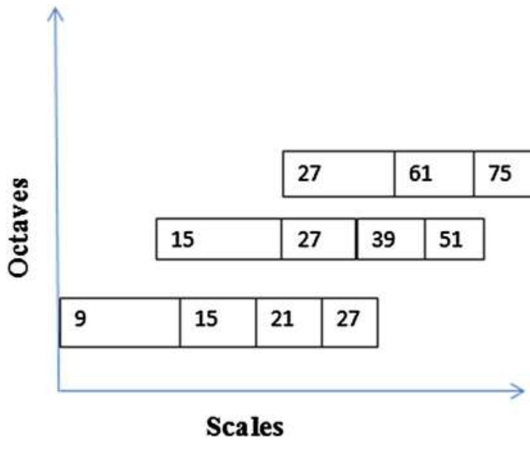


Fig. 5 Increasing the filter size in the SURF algorithm

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}, \tag{4}$$

where D_{xx}, D_{yy}, and D_{xy} represent the second-order Gaussian derivatives in the *x*-, *y*-, and *xy*-directions, respectively. Consequently, for all the scales in the image pyramid, the determinant of Hessians, Det*(H)*, is computed for all the points in the input image according to (5) [3]. Finally, computed determinants are stored in the form of a pyramid, which is called the response pyramid array.

$$\mathrm{Det}(H) = D_{xx}D_{yy} - (0.9 * D_{xy})^2. \tag{5}$$

Stage 3 interest points detection

SURF uses two steps to detect and localize interest points. In the first step, weak points, which have no significant effect on the matching process, are excluded. Points are compared with a threshold that is assigned by the algorithm designer. Those points with greater values are only passed and considered as candidate interest points. Generally, the higher the threshold value, the higher the quality of passed points. In the second step, the location and the scale of these interest points are found. As in the SIFT algorithm, non-maxima suppression is then applied. Each point is compared to its 3 × 3 × 3 neighbors to check if it is a local maximum, and hence, an accepted interest point, or not.

Stage 4 orientation assignment

In order to have rotation-invariant points, descriptors should be directed to the points' orientation. Therefore, for each point, the dominant orientation is calculated using Haar response filters. Haar wavelet filters are applied onto the neighbors of the point in a circle of radius = 6 × the scale of the point. In more detail, for each neighbor point, Haar wavelet filters, of size = 4 × the scale of the point, are applied and then weighted with a Gaussian filter to form *X–Y* responses [3]. To detect the dominant orientation, these responses are summed using a sliding orientation window of size

$\pi/3$ and a step of 0.15. This results in 42 vectors. The dominant vector is finally assigned as the point orientation.

Stage 5 descriptor construction

In the SURF algorithm, the descriptor works within a window of size 20×20. The origin of the window is at the location of the interest point. The window is also rotated to the interest point orientation. As shown in Fig. 6, each window is further divided into 4×4 subregions, which consist of 5×5 points each. Haar responses are then calculated, weighted, and summed up. For each subregion, four summations are generated $\sum dx$, $\sum dy$, $\sum |dx|$, and $\sum |dy|$, which represent summations of responses in x-direction, responses in y-direction, absolute response values in x-direction, and absolute response values in y-direction, respectively. Finally, 4×4 subregions, with four summations each, result in a descriptor vector of 64 features.

3 Related work

In this section, previous parallel implementations of SIFT and SURF algorithms are discussed. In our survey, we consider previous work that enhances the runtime of the two algorithms by using the parallel capabilities of both GPU and CPU.

3.1 SIFT parallel implementations

S. Heymann et al. presented a parallel implementation of the SIFT algorithm using GPUs, in [14]. Authors grouped every 4 pixels in the grayscale to form one RGBA pixel. Their work depended on two main GPU capabilities, the dynamic branching and the multiple render targets (MRTs). For frames of 640×480 resolution, their implementation could process up to 17 frames per second. Authors achieved a good speedup, but with very low quality interest points because of grouping 4 pixels in one RGBA pixel. So, detection of DOG extrema was not handled efficiently.

Sinha et al. [15, 16] also proposed one of the first GPU-based implementations of SIFT, in a series of publications. Precisely, authors used the GPU for the scale space construction and the interest point detection stages of the algorithm. However, the orientation assignment and parts of the descriptor construction stages were computed using the CPU. The reported speedup was 8× to 10×, with respect to the sequential algorithm that was completely run on the CPU.

Wu presented a parallel implementation, which is called SIFTPP, of the improved SIFT, in [17]. It was an enhanced version of SIFT++ sequential implementation by Vedaldi [18]. The author used CPU and GPU together to detect interest points and then sent them back to GPU to find the orientation and build descriptors of these points. Communication overhead between CPU and GPU to work together reduced the gained speedup.

In [19], Zhou et al. implemented the SIFT algorithm using CUDA. They named their implementation CUDA-SIFT. SMs occupancy was improved by calculating the maximum number of warps that could be assigned to each SM. Authors tried to achieve load balancing between threads by processing scales in parallel. In their experimental work, authors used a tesla GPU, c2050, which has 448 CUDA cores running at 1.15 GHZ and a video memory of 3 GB. The reported SM occupancy was 58%. Nevertheless, the employed grid architecture and the method, which was used to simultaneously save the

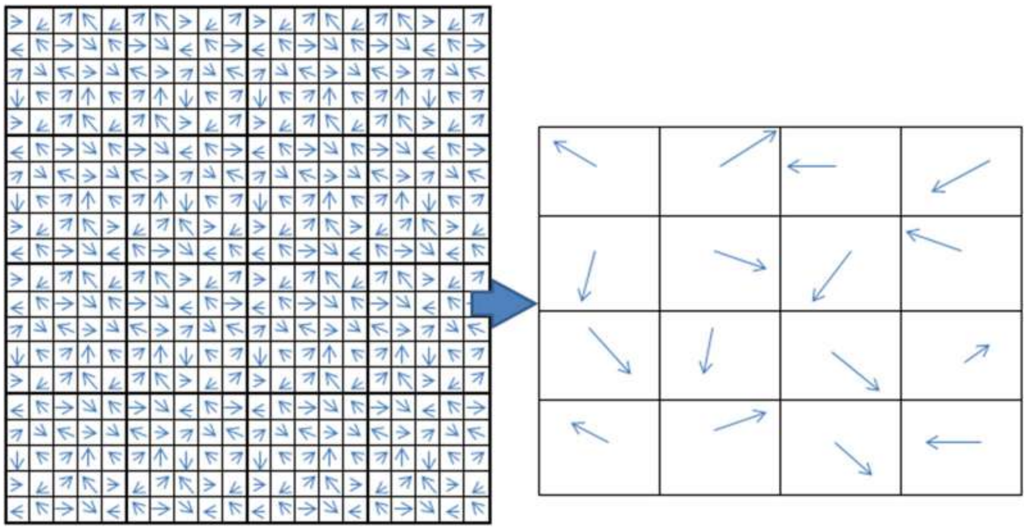


Fig. 6 Descriptor construction in the SURF algorithm

detected points of all scales, were not discussed in the paper.

In [20], Mohammadi et al. proposed another GPU-based parallel implementation of the SIFT algorithm, SIFTCU. In each scale, the first octave should be solely constructed at the beginning. Subsequent octaves could then be constructed in parallel by subsampling this first octave. In the detection stage, SIFTCU tried to speed up storing the detected points onto the interest points array. Accordingly, points of each scale were written in parallel to certain designated potions of the interest points array. Once all scales are finished, a synchronization process is done between all array portions. The speedup gained from parallelizing the writing of the interest points was unfortunately depressed by using atomic add operations. Atomic add operations actually enforced sequential access to the shared memory.

In [21], Acharya et al. implemented SIFT-GPU, which convolved all scales for all octaves at the same time. First, subsampling was done to generate the first layer in each octave. Then, all scales were convolved using the same kernel with one grid. Each block detected its scale and octave from its location within the grid. The use of a unified kernel with one grid was only limited to the scale space construction stage. To speed up the execution, SIFT-GPU also skipped some parts of the algorithm, like the sub-pixel localization. This indeed affected the quality of the detected points. Finally, for 480 $\times$ 640 images, the reported rate was 55 fps.

Parallelizing the SIFT algorithm using multicore CPUs was proposed by Zhang et al. in [5]. Authors mainly concentrated in decreasing the synchronization and enhancing the load balancing between threads. Using a 16-core CPU, the best-reported speedup was 11$\times$. In [6], Moren et al. presented platform-independent methods to parallelize the SIFT algorithm on CPUs, GPUs, or heterogeneous architectures. Therefore, authors used a pipeline-based strategy to compute different octaves and to independently calculate DOG layers. In the descriptor construction stage, they further used the read only memory instead of registers. Nevertheless, as other previous parallel implementations, presented methods employed atomic operations, which depressed some of the gained speedup. In summary, previous parallel implementations of the SIFT algorithm either resulted in low-quality interest points or could not achieve the maximum possible speedup from the underlying parallel architecture. This is mainly due to load imbalance, thread synchronization, and the use of atomic operations. In this paper, we present better solutions to these problems, which result in higher speedup, without sacrificing the quality of the generated interest points.

3.2 SURF parallel implementations

The first stage of the SURF algorithm, i.e., integral image computation, could be accomplished in parallel using prefix sum algorithms. In [22], Harris et al. employed the naïve parallel scan to calculate the prefix sum, as shown in Fig. 7a. In the first iteration, the values of any two one-step adjacent cells were added together. This step was then doubled for each subsequent iteration. The problem of naïve scan was that it took too much time to finish all required additions. Therefore, a work-efficient parallel scan was presented by Terriberry et al. in [23], based on Blelloch algorithm, in [24]. As shown in Fig. 7b and c, the work-efficient parallel scan consisted of two phases. In the up-sweep phase, an ordinary binomial summation was carried out. In the down-sweep phase, the value of the last cell was first replaced by zero. Throughout iterations, cells were manipulated in the reverse direction of the up-sweep phase. Values of associated cells were exchanged together, and their summation was saved in the cell with the higher index. In [25], Bilgic et al. used the work-efficient parallel scan to compute integral images on GPUs. In a nutshell, each row, or column, was assigned a block to find its prefix sum.

In [26], Fang et al. analyzed different types of parallelism that could be used with image retrieval algorithms, like SURF. These types were summarized as follows:

- *Pipeline parallelism* By making a pipeline of all or some stages of the algorithm, different images could be simultaneously processed, but in different stages.
- *Scale-level parallelism* Computations within the same scale were done in parallel. However, different scales were sequentially processed. As the number of cores increased, authors found that scale-level parallelism suffered from significant load imbalance and communication overhead.
- *Block-level parallelism* An image was partitioned into blocks and the whole algorithm ran on each block in parallel. Authors reported that block-level parallelism outperformed the other two types of parallelism.

Zhu et al. presented a parallel implementation of the SURF algorithm, CSURF, using block-level parallelism, in [7]. Authors further used a pipeline between the CPU and the GPU. Each kernel was divided into many smaller ones to avoid the problem of limited shared memory. Integral image computation was actually done on the CPU, whereas interest points detection and descriptor construction were accomplished on the GPU. CSURF achieved a speedup of 15$\times$ the maximally optimized CPU version. However, this speedup depended mainly on CPU capabilities rather than GPU capabilities. Similarly, in [27], Schulz et al. employed

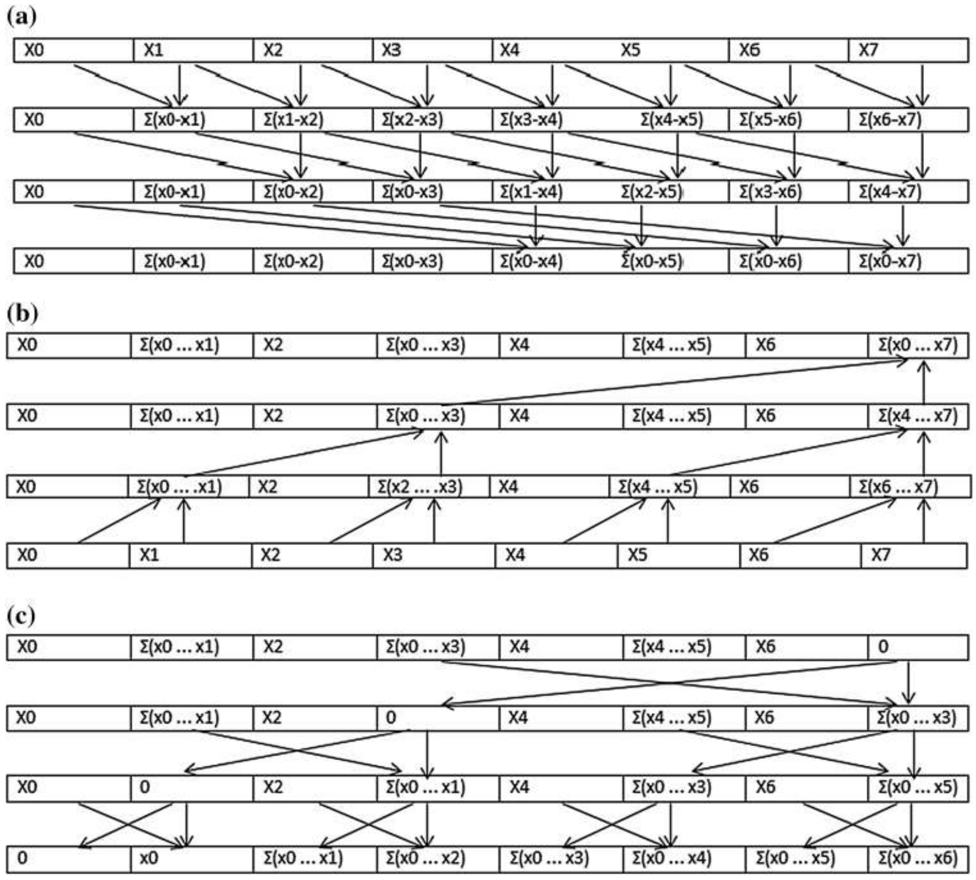


Fig. 7 Parallel prefix sum algorithms: **a** naïve parallel scan. **b** Up-sweep phase of the work-efficient parallel scan. **c** Down-sweep phase of the work-efficient parallel scan

block-level parallelism in their implementation, CUDA-SURF. CUDA-SURF was built on CUDA 2.3 and OpenCV 2.1.

Lu et al. targeted parallelizing the SURF algorithm using multicore CPUs, in [9]. Accordingly, authors proposed an adaptive pipeline parallel scheme (AD-PIPE). To overcome the load imbalance problem, they adjust the number of threads in each stage according to the workload. In [28], Cheon et al. proposed an enhanced version of the sequential SURF algorithm, eSURF. Authors made use of the repeated scales in many octaves to unify the scale space construction stage with the detection stage. They further presented methods to allocate memory efficiently. eSURF is found to be 30% faster than the original algorithm.

In [8], Yan et al. employed OpenCL to parallelize the SURF algorithm, OpenCL SURF. To speed up the execution, authors used coalesced memory access [29]. They also processed the detection stage of all octaves in parallel. However, they saved the detected point using atomic operations. For 320 × 240 images, a rate of 25 fps and a maximum speedup of 22.8× the CPU version were reported.

Like the parallel implementations of the SIFT, those of the SURF algorithm suffer from the load imbalance and the use of atomic operations. These enforce the GPU to work sequentially. This again depresses the speedup that could be attained from the employed parallel architecture. In this paper, we tackle these problems by presenting new GPU-based implementations of both SIFT and SURF algorithms. Our implementations aim at exploiting the maximum possible speedup from the employed GPU.

4 GPU: model of execution and challenges

In this section, we introduce basic concepts and definitions regarding GPUs. Parallelization challenges that prevent previous implementations from obtaining the maximum possible speedup are also discussed. From the physical perspective, a GPU simply consists of an array of SPs that are grouped to form SMs, an execution manager, and different types of storage modules, like the global memory, the constant memory, caches, and registers. From the logical perspective, functions, which are executed on the GPU, are called kernels. Each kernel constitutes a grid of blocks. Each block, in turn, is formed from a 1D, 2D, or 3D array of threads. Once a kernel is invoked, the execution manager distributes the blocks over the available SMs. In more detail, consecutive threads are grouped together to form warps. Warp is a vendor-specific concept, and it represents the basic unit that could be scheduled by the execution manager. Only one warp could physically run on an SM at a time. When a warp executes instructions with long latency, the SM switches to another ready-to-execute warp. Therefore, the presence of more warps on an SM guarantees that it would often find work to do.

GPU is an example of single instruction, multiple data (SIMD) architecture [30]. All threads within the same grid process the same instruction, but with different data. Therefore, these data are first transferred from the main memory of the computer to the global memory of the GPU card. Thereafter, they are processed using the GPU parallel capabilities. GPU cards have a high theoretical bandwidth with a rate of tens of gigabytes per second. This rate might also increase to hundreds of gigabytes per second for new memory and interconnect technologies [31]. However, the effective bandwidth depends on how memory is actually accessed. It is known that requesting a number of memory locations takes less time if these locations are in a consecutive order within the memory. A coalesced memory access is the type of memory accesses in which sequential memory locations are requested. These requests are coalesced together as a single memory access [29]. Accordingly, if concurrent threads within a warp request sequential locations from the global memory, only one coalesced memory access takes place. On the contrary, if these concurrent threads request sparse and non-adjacent memory locations, each request is processed separately and the bandwidth drops significantly. Therefore, a good programming practice, in which consecutive threads request consecutive global memory locations, results in a higher effective bandwidth.

In order to maximize the performance of the GPU, an optimal number of threads should be used. This optimal number is often limited by the resources of each SM, such as the shared memory and registers. Limitations and features of the underlying hardware are abstracted by its compute capability version. Furthermore, SM occupancy is defined as the ratio of the actual number of running warps to the maximum number of warps that could run in that SM. In order to exploit the maximum performance of the underlying hardware, SM occupancy should be increased. One goal of the work presented in this paper is to increase the SM occupancy for SIFT and SURF parallel implementations, as much as possible. To achieve this goal, we consider two SM occupancies: theoretical and runtime. On the one hand, for any logical organization, the theoretical SM occupancy could be calculated according to the employed GPU resources and limitations. On the other hand, the runtime SM occupancy is measured during the execution by the help of the underlying hardware and profiling tools. It represents the average number of active threads over a certain number of clock cycles, i.e., time.

In order to exploit the maximum performance from a GPU, concurrent threads should be almost assigned the same amount of work. This is known as load balance. Unbalanced distribution of load among threads increases the necessary time to finish the whole task and reduces the runtime occupancy. Therefore, balancing the load between threads is one challenge that, if not realized, would reduce the performance of GPUs. It also reduces the performance of previous parallel implementations of the feature extraction algorithms on GPUs. In the context of SIFT and SURF algorithms, scale space construction and interest points detection stages have scales of different dimensions. One proposal is to use a large fixed-size kernel for all stages. However, as the execution progresses, the number of idle threads increases and the optimal load balance could not be realized. Therefore, in order to reduce the load imbalance between threads, most previous parallel implementations sequentially process scales with dimension-customized independent kernels. However, this technique has two main drawbacks. First, multiple kernel invocations result in a significant time loss. Second, as the execution progresses, small kernels that are solely processed by the GPU would leave many of the processing elements doing nothing. Once again, the optimal load balance could not be reached. In this paper, we try to overcome this challenge by using one kernel for each stage, rather than for each scale. Consequently, this allows us to reach a midway between the too many kernel invocations, on one extreme, and the significant load imbalance, in the other extreme.

Another challenge that reduces the performance of GPUs, and previous parallel implementations of the feature extraction algorithms, is the use of atomic operations. Atomic operations are used by the GPU to enforce synchronization between threads. They act as semaphores, when multiple threads try to write to the same location in the global or shared memory. The location is locked, and threads are enforced to access it in a sequential manner. As a result, atomic operations serialize parallel threads and decrease the speedup that could be gained. Previous parallel implementations of SIFT and SURF algorithms use atomic add instructions to enforce threads to write the detected points in a unique index. Accordingly, all threads are queued, waiting its turn to save results. Subsequent scales could not be manipulated until the current scale is finished. This indeed slows down the execution of the interest point detection stage. In this paper, we try to overcome the atomic operation challenge by presenting a new algorithm that calculates the required unique memory index. As there is no need to arrange points while the detection stage is running, we use one grid of blocks to detect all points in parallel. We create a global memory array to write detected points in it and then do some work to arrange these points with a unique index. Detection stage is computed in parallel without any serialization, and other work is responsible for getting unique index using the prefix sum algorithm.

5 Methodology

In this section, our proposed parallel implementations of SIFT and SURF algorithms are discussed in detail. First, we demonstrate the preprocessing that is done for both SIFT and SURF. Two new algorithms, which speed up the calculation of the prefix sum for the SURF algorithm, are presented in Sect. 5.1. Section 5.2 explains our grid organization to enhance the load balance, in the scale space construction stage. Our new algorithm that replaces the use of atomic operations, in the interest points detection stage, is then proposed in Sect. 5.3. Finally, Sect. 5.4 describes our implementations of the orientation assignment and the descriptor construction stages.

5.1 Data preprocessing

5.1.1 Data preprocessing for SIFT algorithm

Subsampling is the preprocessing that is required for the SIFT algorithm. In the literature, octaves are processed sequentially such that the first scale of any octave is not created unless the preceding octave is completely finished. Contrarily, in our implementation, the original image itself is subsampled multiple times at the beginning of execution to simultaneously create the first scale of all octaves. Thereafter, all scales of all octaves are constructed in parallel by convolving the already generated first scale in each octave.

5.1.2 Data preprocessing for SURF algorithm

Computing the integral image is the preprocessing that is required for the SURF algorithm. We propose two new parallel algorithms to quickly calculate the integral image. Each algorithm targets a certain GPU compute capability version. One algorithm is for compute capability 1.×, whereas the other is for compute capability 2.× and higher. In the following paragraphs, the former is named the two-way algorithm, while the latter is named the one-way algorithm.

The two-way algorithm is listed in Algorithm 1. It actually implements the steps that are shown in Fig. 8. Our two-way algorithm enhances the execution time of the second phase of the work-efficient parallel scan, which is discussed in Sect. 3.2. Accordingly, phase 1 is similar to the up-sweep phase of the work-efficient parallel scan and an ordinary binomial summation is carried out. Nevertheless, as shown in Fig. 8, our implementation of phase 2 starts by dividing the first row into two partitions. The value of the last cell in the left partition is added to that of the middle cell in the right one. At each subsequent level, the partition size is decreased to the half and the process is repeated. Each level in our implementation requires only sum operations. Therefore, it takes less time than the wok-efficient parallel scan, which needs sum and swap operations in each level.

Algorithm 1 Proposed two-way algorithm for prefix sum computation of n cells

```
1.  i= thread idx;            //Each thread gets its index
2.  For j=1:log(n)            //Phase 1
3.    x=n/(2×j)               //Calculate the number of working threads (x) at this level
4.    If i<x
5.      Y=i×(2^j)             // Calculate the index of the destination cell ( y ) for this
                              //thread
6.      [y]<=[y] +[y-2^(j-1)] //Add values and save the result in the destination cell
7.   end
8.  For j=1: log (n)-1        //Phase 2
9.    z=2^j-1                 //Calculate the number of working threads ( z ) at this level
10.   If (i<z)
11.     m= n/(2×i )           //Calculate the index of the source cell at this level ( m )
12.     [m + (n/4×j)]<= [m] + [m + (n/4×j)]  //Add the two values and save the result
                              //in the destination
13. end
```

GPUs with compute capability 2.× and higher have the capability to broadcast and multicast memory addresses. They could serve multiple read requests by different threads within the same warp to a certain memory address without suffering from a shared memory bank conflict. Our one-way algorithm benefits from this capability. It is listed in Algorithm 2, and its implementation onto 8 threads is shown in Fig. 9. It is named a one-way algorithm because it only requires one phase to compute the prefix sum. Our one-way algorithm mixes between the naïve parallel scan and the work-efficient parallel scan, which are discussed in Sect. 3.2. As shown in Fig. 9, the algorithm starts by dividing the first row into partitions of two cells. The partition size doubles at each subsequent level. In each partition of P cells, the value of cell $P/2$ is added to that of all successor cells within its own partition. Needless to say, by saving the time of the second phase, our one-way algorithm outperforms both the work-efficient parallel scan and our proposed two-way algorithm.

(4.m, 21.n/16). It is worth emphasizing that our unified grid could be used with any image resolution, i.e., any values of m and n, without exceeding the grid size limitations in CUDA.

Block-level parallelism, which is discussed in Sect. 3.2, is used in this stage. Accordingly, an image is partitioned into rectangular segments. Each block has the responsibility to convolve pixels within one segment. Each pixel has one thread to serve. Each thread identifies which octave to serve by its block index in the y-direction and which scale to serve by its block index in the x-direction. In order to carry out the convolution process, for a Gaussian filter of size f, each block needs pixels of its designated segment of the image plus $f/2$ pixels in each side. Nevertheless, to speed up the execution, all threads actually collaborate in loading these pixels and coalesced memory access, which is discussed in Sect. 4, is used. Due to the limitations on the number of registers for compute capability 1.3, block dimensions of 16 × 8 threads could maximally be used. For compute capability 2.× and higher, each block consists of 16 × 16 threads, in order to maximize the SM occupancy.

Algorithm 2 Proposed one-way algorithm for prefix sum computation of n cells

```
1.  i= thread idx;              //Each thread gets its index
2.  For j=1: log (n)
3.    p=2^j                     //Calculate the partition size( p ) at layer j
4.    z=ceil (2×i/p)            //Each thread detects which partition it serves
5.    s= (2×z-1)×p/2            //Calculate the index ( s ) of the cell, whose value is added
      //to that of the following cells
6.    d=i-(z-1)×p/2             //Calculate the destination cell index ( d ) for each thread
7.    [d]=[s] + [d]             //Add the two values and save the result in the destination
8.  end
```

5.2 Scale space construction

5.2.1 Scale space construction for SIFT algorithm

As mentioned in Sect. 4, in our implementation, we try to reach a midway between significant load imbalance and excessive kernel invocations. Therefore, we carry out all the computations of the scale space construction using only one kernel. This results in a unified grid that simultaneously manipulates all scales of all octaves. Figure 10 shows an example of our unified grid for 3 octaves, of 4 scales each. For an $m \times n$ image, the first scale of the first octave needs $m \times n$ threads to process all pixels in parallel. This results in a total of $4 \times m \times n$ threads for the first octave. Due to subsampling in each subsequent octave, the required number of threads shrinks to one quarter of that in the previous octave. Therefore, for 3 octaves of 4 scales each, the final dimensions of our unified grid would be

After finishing the convolution process, the kernel calculates the DOG using a similar unified grid approach. As shown in Fig. 11, each block processes the results of the Gaussian convolution for its scale and the scale after it. Therefore, collaborative loading and coalesced memory access are again used in loading scales onto the shared memory. Thereafter, all blocks calculate the difference between their two designated scales to construct all DOG layers in parallel. Finally, DOG results are saved into the global memory.

5.2.2 Scale space construction for SURF algorithm

Similar to SIFT, we use our unified grid approach in constructing the scale space of the SURF algorithm. All

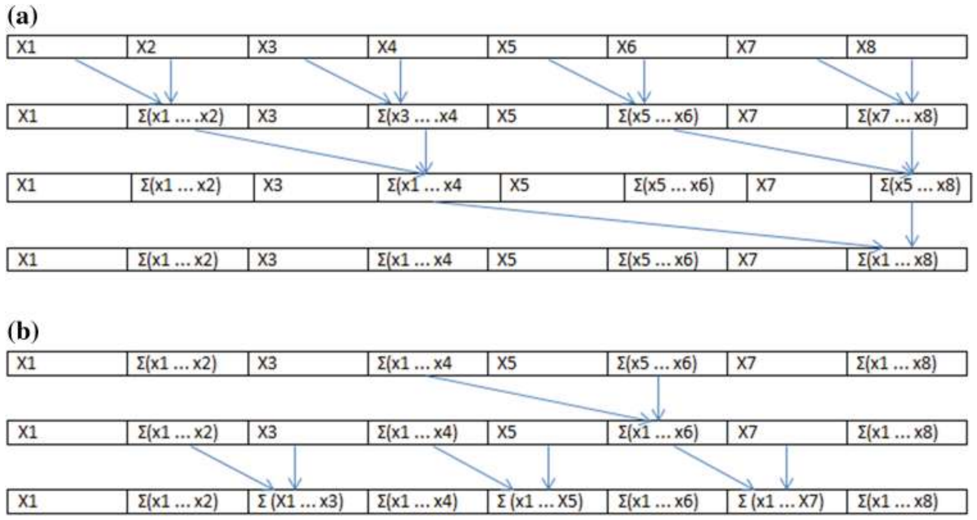

(a)
X1 X2 X3 X4 X5 X6 X7 X8
X1 Σ(x1 … .x2) X3 Σ(x3 … .x4 X5 Σ(x5 … x6) X7 Σ(x7 … x8)
X1 Σ(x1 … x2) X3 Σ(x1 … x4 X5 Σ(x5 … x6) X7 Σ(x5 … x8)
X1 Σ(x1 … x2) X3 Σ(x1 … x4 X5 Σ(x5 … x6) X7 Σ(x1 … x8)
(b)
X1 Σ(x1 … x2) X3 Σ(x1 … x4 X5 Σ(x5 … x6) X7 Σ(x1 … x8)
X1 Σ(x1 … x2) X3 Σ(x1 … x4) X5 Σ(x1 … x6) X7 Σ(x1 … x8)
X1 Σ(x1 … x2) Σ (X1 … x3) Σ(x1 … x4) Σ (x1 … X5) Σ(x1 … x6) Σ (x1 … X7) Σ(x1 … x8)

Fig. 8 Explanation of the two-way algorithm for prefix sum computation: **a** phase 1, **b** phase 2

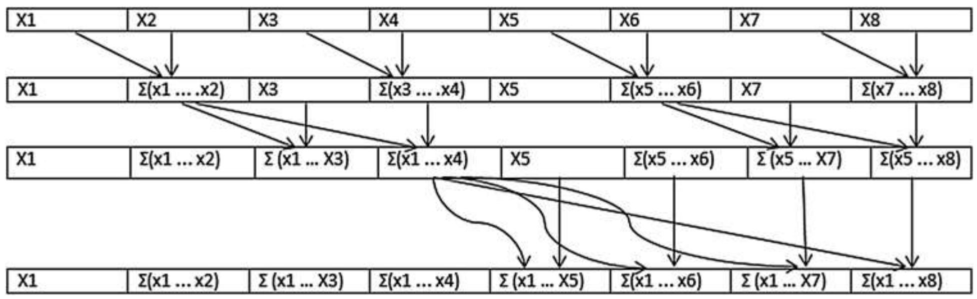

X1 X2 X3 X4 X5 X6 X7 X8
X1 Σ(x1 … .x2) X3 Σ(x3 … .x4) X5 Σ(x5 … x6) X7 Σ(x7 … x8)
X1 Σ(x1 … x2) Σ (x1 … X3) Σ(x1 … x4) X5 Σ(x5 … x6) Σ (x5 … X7) Σ(x5 … x8)
X1 Σ(x1 … x2) Σ (x1 … X3) Σ(x1 … x4) Σ (x1 … X5) Σ(x1 … x6) Σ (x1 … X7) Σ(x1 … x8)

Fig. 9 Explanation of the one-way algorithm for prefix sum computation

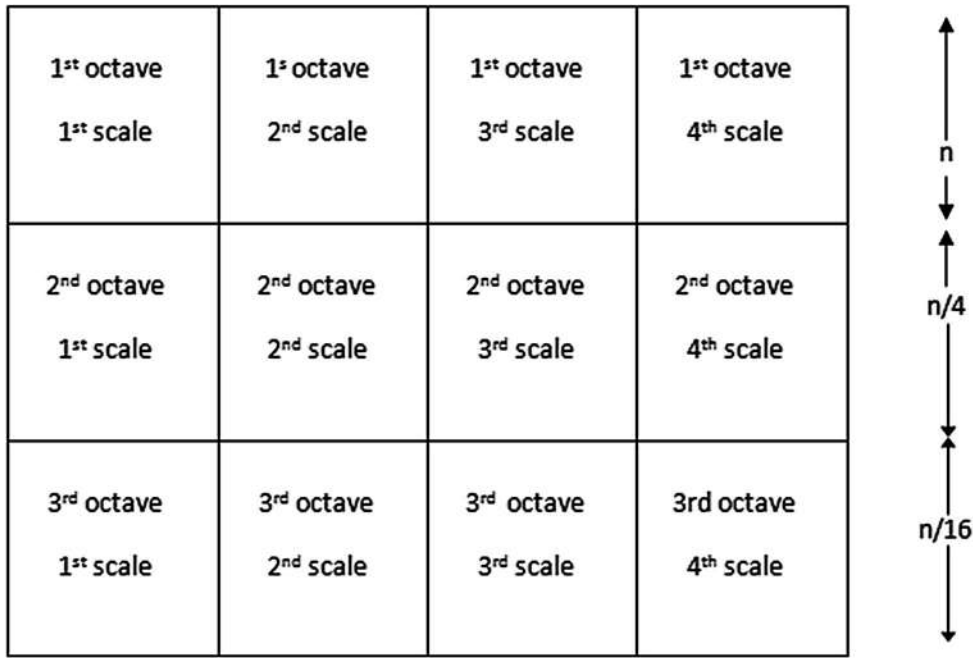


Fig. 10 Proposed unified grid for SIFT scale space construction

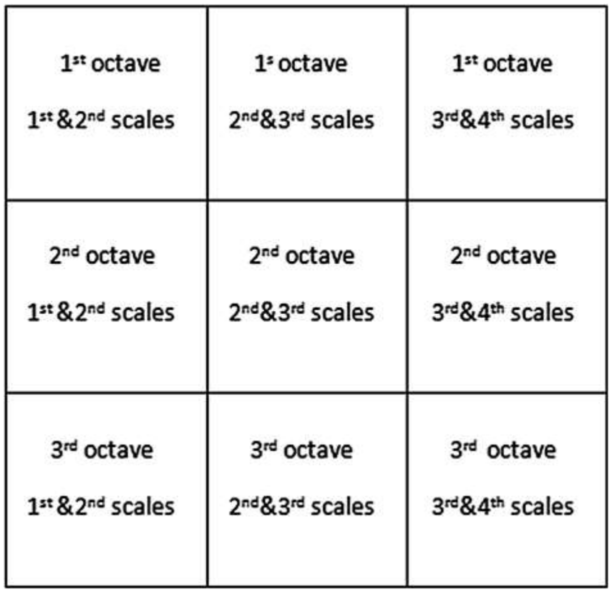


Fig. 11 Unified grid for DOG calculation

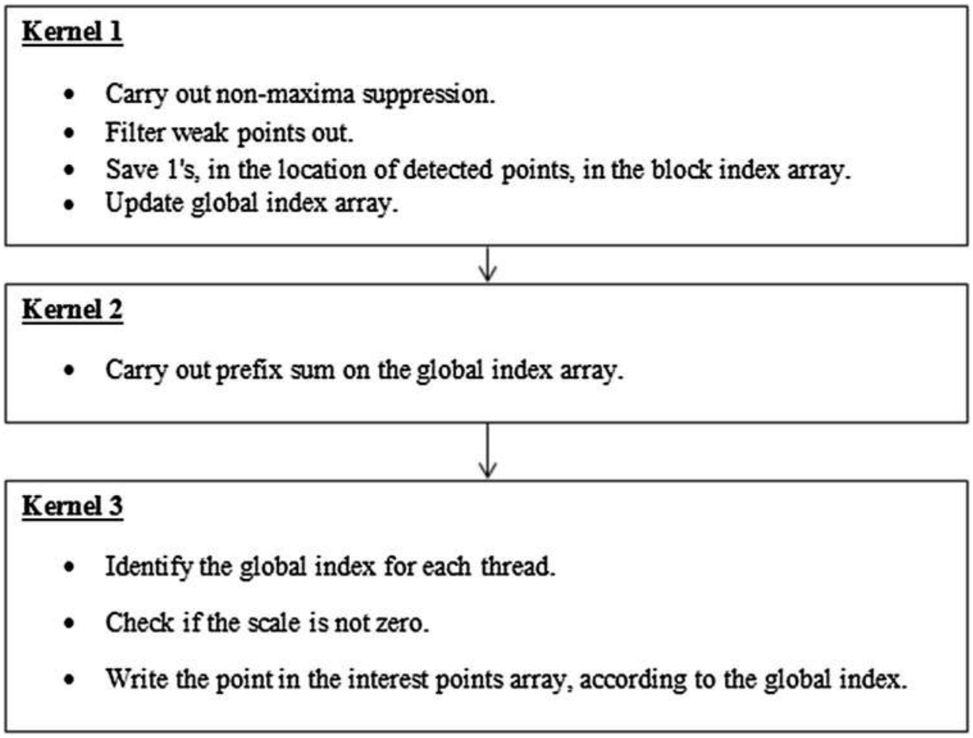
calculations are done in parallel by directly processing the original image. This is realized by increasing the filter size, according to the scale. However, our implementation of the SURF has two differences from that of the SIFT algorithm. First, each thread now calculates the Hessian matrix for one point. Second, the grid size of SURF is different from that of SIFT. No subsampling is carried out in the SURF algorithm. Accordingly, starting from the second octave, the number of required threads does not decrease. In other words, for an $m \times n$ image, each scale needs a block of $m \times n$ threads to serve it. Therefore, for 3 octaves of 4 scales each, the final dimensions of our unified grid would be $(4.m, 3.n)$.

5.3 Interest points detection

For both SIFT and SURF algorithms, the detection stage could be divided into two main steps. In the first step, DOG extrema are located and then filtered by deleting low-contrast points. In our implementation of the two algorithms, we used our unified grid approach, as discussed in Sect. 5.2, to locate all interest points in parallel. An image is again partitioned into rectangular segments that are processed by blocks of our unified grid in parallel. Each pixel is assigned a thread to serve. As explained in Sect. 2.1, the thread should compare that pixel to its 26 neighbors, in order to decide whether it is a local maximum or not. Therefore, in a coalesced manner, threads within a block collaboratively load pixels of that block plus one pixel in each direction to the shared memory. For example, for a block of 16 $\times$ 16 threads, a total of 18 $\times$ 18 pixels are loaded. Consequently, comparisons are simultaneously made by all threads and candidate interest points are generated. Finally, these candidate points are filtered using sub-pixel interpolation to eliminate low-contrast points. Strong points, which survive the filtering process, constitute the interest points of the image.

In the second step of the interest points detection stage, detected points are stored into a global interest points array. In previous parallel implementations of the two algorithms, all threads share a global index, which they use to store detected points into that global interest points array. As previously discussed in Sect. 4, these implementations further use atomic add instructions to increment this index. In other words, a thread could not increase the global index and store its detected point, if any, unless its preceding thread is completely finished. Indeed, this serializes the execution and a potential speedup is therefore lost. Nevertheless, in our implementation of the two algorithms, we completely eliminate the use of these atomic add instructions. Instead, we propose a methodology that allows each thread to directly know its designated index in the global interest points array. Using our approach, all threads use their unique index to simultaneously store the detected interest points.

Our implementation of the interest points detection stage is shown in Fig. 12 and detailed in Algorithm 3. We propose a new 3-kernel methodology to enable threads to efficiently detect interest points and save them, without using atomic instructions. Kernel 1 detects candidate interest points and filters weak ones out. Kernels 2 and 3 are responsible for saving the detected points into the global interest points array. Figure 12 lists the tasks performed by each kernel in detail. In our methodology, each block has two local arrays, which are shared by its threads. The dimensions of these arrays are those of the block. The first one is the block index array, which holds 1s or 0s to detect whether the pixel in any location is an interest point or not, respectively. The second one is the block scale array, which holds the scale of detected interest points. Moreover, besides the global interest points array, our methodology uses one global index array. At the end of our methodology, this global index array would have a unique index for each thread. The unique index would be used by this thread to write into the global interest points array. The dimensions of this global index array are those of the DOG middle layer.



Kernel 1

- Carry out non-maxima suppression.
- Filter weak points out.
- Save 1's, in the location of detected points, in the block index array.
- Update global index array.

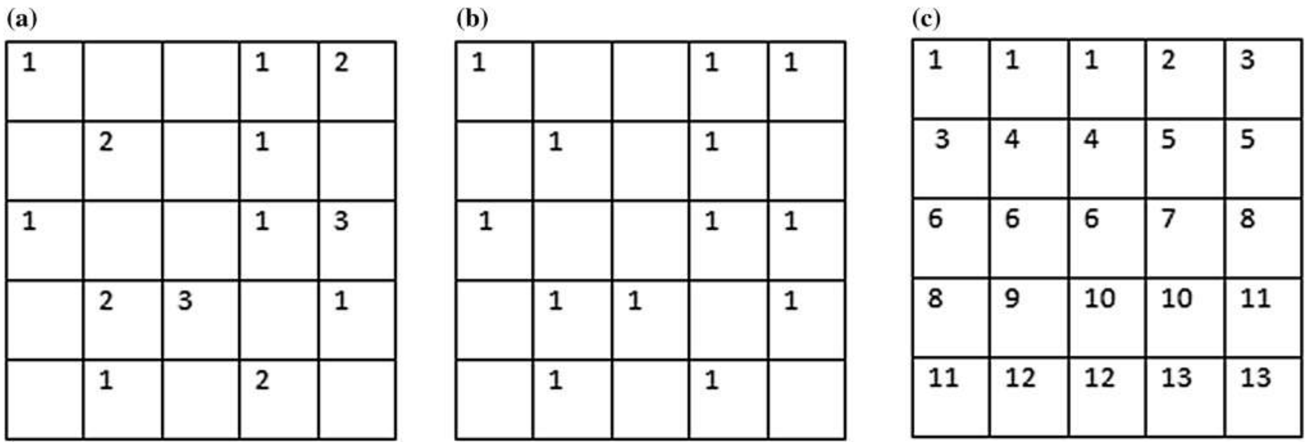
Kernel 2

- Carry out prefix sum on the global index array.

Kernel 3

- Identify the global index for each thread.
- Check if the scale is not zero.
- Write the point in the interest points array, according to the global index.

Fig. 12 Kernels used in the interest points detection stage



(a)

1			1	2
	2		1	
1			1	3
	2	3		1
	1		2	

(b)

1			1	1
	1		1	
1			1	1
	1	1		1
	1		1	

(c)

1	1	1	2	3
3	4	4	5	5
6	6	6	7	8
8	9	10	10	11
11	12	12	13	13

Fig. 13 An example on our 3-kernel methodology to generate a unique global index for each interest point. **a** Block scale array after applying kernel 1. **b** Block index array after applying kernel 1. **c** Global index array after completely applying our methodology

Algorithm 3 explains our methodology in detail. Kernel 1 is corresponding to lines 1–17. In the first three lines, the kernel loads pixels into the shared memory. In lines 4–8, each pixel is compared to its 26 neighbors to identify whether it is a candidate interest point or not. In lines 9–10, sub-pixel interpolation is carried out to eliminate weak points. In lines 11–16, for each detected interest point, a value of 1 is written in its corresponding location of the block index array. The scale is also saved into its corresponding location of the block scale array. In line 17, results of all threads are finally saved into the global index array. As an example of arbitrary outputs resulted from kernel 1, Fig. 13a and b shows 5×5 portions of the block scale array and the block index array after applying kernel 1, respectively.

ithm 3 Proposed implementation of the interest points detection stage.

```
ead and block indexes to identify which points in the DOG pyramid to serve
xels to the shared memory

            //v is a boolean variable that indicates if the pixel is a local maximum
ghbor=1:26
ghbor > point
 0

ut the sub-pixel interpolation
ected interest points
 in the block index array
he scale in the block sale array

ues to the global index array

refix
refix
```

Kernel 2 is corresp nding
In line 18, a prefix su is
index array. As a re ult.
have the number of te
prefix sum is only a
index array. By do g
have the total numb r o
and all preceding row
detected interest p ints,
ending cell at the ast ro

Kernel 3 is correspond
Each block in this kernel

scale is found, the thread saves its corresponding point into the global interest points array, using its unique index from the global index array. Finally, for the same 5 $\times$ 5 arbitrary outputs resulted from kernel 1, Fig. 13c shows its corresponding 5 $\times$ 5 portion of the global index array, after applying kernel 3. As shown in the figure, each interest point, which has a scale in Fig. 13a, gets a unique global index in Fig. 13c. This unique index is directly used to save the interest point into the global interest points array, without any need for atomic add instructions.

At the end, one point should be emphasized. Although our methodology carries out extra work than previous parallel implementations, we managed to completely avoid using atomic add instructions. Therefore, our methodology results in a lower overall execution time.

5.4 Orientation assignment and descriptor construction

In our implementation of the orientation assignment stage for both SIFT and SURF algorithms, each block calculates the orientation of many interest points. In detail, every 32 threads within a block are assigned a single interest point to serve. Each thread is only responsible for calculating one or two directions and saving them in the orientation histogram. The dominant direction of the point is assigned as its orientation.

In our implementation of the descriptor construction stage for both SIFT and SURF algorithms, each block consists of 16 $\times$ 8 threads. We assign 16 threads for each interest point to construct its descriptor vector. These threads are corresponding to the 16 regions of the algorithm, and each of them calculates the gradients histogram for a subregion in the descriptor window.

6 Experimental results

This section presents our experimental results and validates the efficiency of our parallel implementations. Throughout the section, we refer to our parallel implementation of SIFT and SURF algorithms by PR-SIFT and PR-SURF, respectively. In our experiments, we use an Intel Xeon, E5-2667, CPU with 16 GB RAM. The Intel Xeon is a 130 W CPU that has 6 cores with 15 MB cache and works at 2.9 GHz. To verify the powerfulness of our implementations, four GPUs with a different number of cores, i.e., SPs, are used. These GPUs are NVIDIA GTX-275, 240 cores and 1.4 GHz, NVIDIA GTX-480, 480 cores and 1.24 GHz, NVIDIA GTX-750, 512 cores and 1.02 GHZ, and NVIDIA GTX-960, 1024 cores and 1.17 GHz. The four GPUs have a compute capability of 1.3, 2.0, 5.0, and 5.2, respectively. It is worth mentioning that compute capabilities of 1.$\times$ and 2.$\times$ are not supported starting CUDA 7 and CUDA 9, respectively [32]. However, we consider them to show the efficiency of our implementations, in case a resource-limited GPU is used. Therefore, results of GTX-275 and GTX-480 are only included when we evaluate the performance of our implementations against changes in GPU features and capabilities. Otherwise, results of the two other GPUs are only presented. Images used in our experiments are taken from Barandiaran [33] and Mikolajczyk [34] datasets. The two datasets are widely used in comparing local features extraction techniques. The following subsections discuss our results in terms of speedup, accuracy, and SM occupancy. All timing results presented in this section include the processing time and the data transfer time from main memory to the GPU.

6.1 Speedup

In this subsection, we aim at evaluating how speedy are our implementations with respect to previous ones. This evaluation is done for images of different qualities. Accordingly, for both SIFT and SURF, we start by presenting the speedup of our implementation over the sequential one. Thereafter, the performance of our implementation in real time is evaluated by calculating the achieved frames per second. Our implementations are then compared to previous parallel counterparts of the two algorithms. Finally, a statistical analysis using the standard deviation and the coefficient of variation is performed to evaluate how consistent our implementations are in realizing the presented speedup. Throughout this subsection, Barandiaran dataset is used [33]. For any image quality, in order to increase the credibility of our results, ten arbitrary images of that same quality are selected from the dataset. The average speedup and frames per second are then calculated over these ten images. For different image qualities, these averages are the ones used in presenting our results in all figures within the subsection.

6.1.1 SIFT results

For images of different qualities, we first evaluate our parallel implementation with respect to the sequential implementation, SIFTPP [18]. Figure 14 shows the speedup we achieved using the four employed GPUs for images of different resolutions. The figure clarifies that the speedup increases when either the number of cores within the GPU or the image resolution increases. Accordingly, for an image of 1280 $\times$ 1024 resolution, PR-SIFT, running on GTX-960, achieves a maximum speedup of 28$\times$ with respect to the sequential algorithm that runs on the Xeon, E5-2667, CPU.

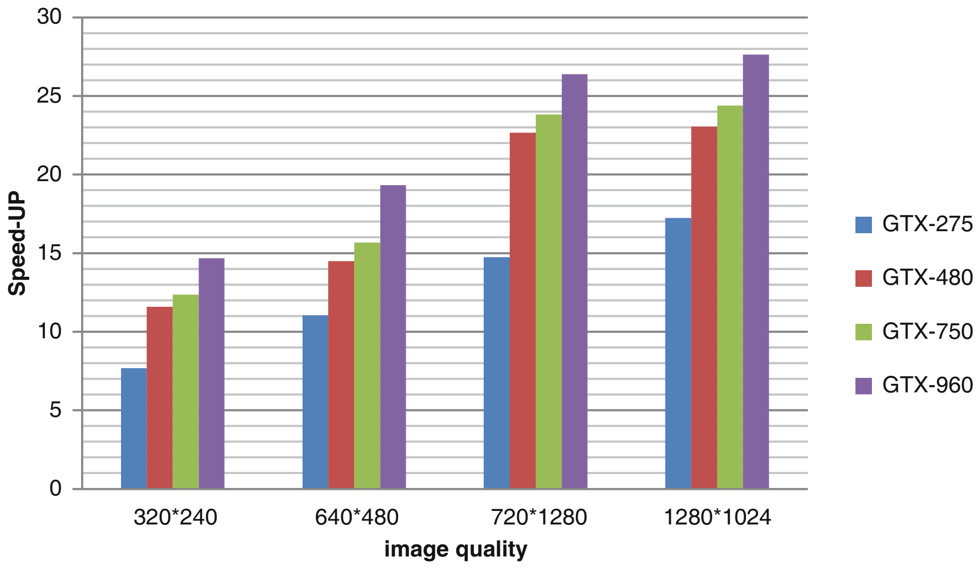


Fig. 14 Speedup achieved by PR-SIFT over SIFTPP for different GPUs and image quality

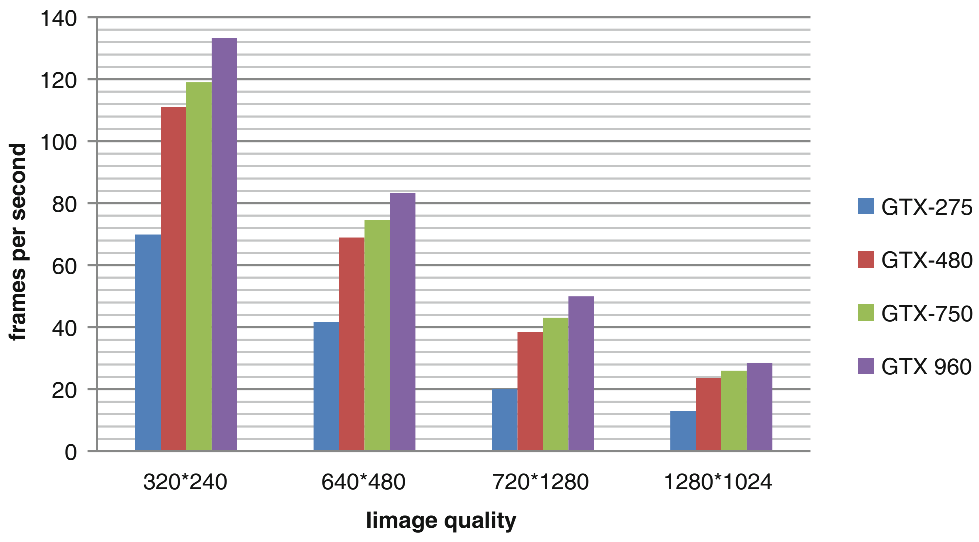


Fig. 15 Frame rate resulted from PR-SIFT for different GPUs and image quality

To evaluate the efficiency of our implementation in real-time applications, we calculate the number of frames that could be processed by it per second. Figure 15 shows this frame rate for the same GPUs and image quality. The figure first shows that the achieved frame rate decreases as the number of cores within the GPU decreases or the image resolution increases. Accordingly, PR-SIFT manages to achieve a maximum frame rate of 133 frames per second for an image resolution of 320 × 240. Furthermore, for different image qualities, the figure clarifies that our implementation could efficiently be used in real-time applications.

We then compare the frame rate resulted from our implementation to that of previous parallel implementations, which are reviewed in Sect. 3.1. SIFT-GPU [21], CUDA-SIFT [19], and two versions of the platform-independent implementation, OpenCL SIFT [6], are considered for comparison. One version runs on GPUs, whereas the second version is executed on our 6-core Xeon CPU. It is worth mentioning that we do not actually implement CUDA-SIFT. As it is compared to SIFT-GPU in [17], we rather employ normalized results for it with respect to SIFT-GPU. For the GTX-750 GPU, Fig. 16 shows the frame rate resulted from the five aforementioned implementations. For images with resolutions of 640 × 480 and higher, all GPU-based implementations significantly outperform the CPU-based one. Moreover, our own implementation accomplishes a higher frame rate than all other parallel implementations. In general, the percentage enhancement in the frame rate of our implementation with respect to that of previous implementations grows up as the image quality increases. For example, for

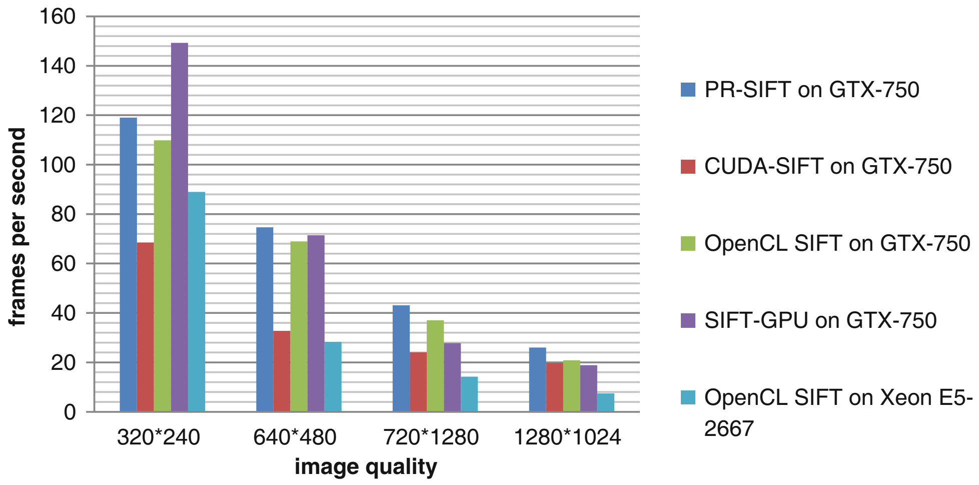


Fig. 16 Frame rate resulted from different parallel implementations of the SIFT algorithm

an image resolution of 1280 × 1024, our implementation significantly outperforms SIFT-GPU and OpenCL SIFT, which runs on the Xeon CPU. Furthermore, it achieves 1.42 and 1.24 higher frame rates with respect to CUDA-SIFT and OpenCL SIFT, which runs on the GPU, respectively.

We then compare the frame rate resulted from our implementation to that of previous parallel implementations, which are reviewed in Sect. 3.1. SIFT-GPU [21], CUDA-SIFT [19], and two versions of the platform-independent implementation, OpenCL SIFT [6], are considered for comparison. One version runs on GPUs, whereas the second version is executed on our 6-core Xeon CPU. It is worth mentioning that we do not actually implement CUDA-SIFT. As it is compared to SIFT-GPU in [17], we rather employ normalized results for it with respect to SIFT-GPU. For the GTX-750 GPU, Fig. 16 shows the frame rate resulted from the five aforementioned implementations. For images with resolutions of 640 × 480 and higher, all GPU-based implementations significantly outperform the CPU-based one. Moreover, our own implementation accomplishes a higher frame rate than all other parallel implementations. In general, the percentage enhancement in the frame rate of our implementation with respect to that of previous implementations grows up as the image quality increases. For example, for an image resolution of 1280 × 1024, our implementation significantly outperforms SIFT-GPU and OpenCL SIFT, which runs on the Xeon CPU. Furthermore, it achieves 1.42 and 1.24 higher frame rates with respect to CUDA-SIFT and OpenCL SIFT, which runs on the GPU, respectively.

For the resolution of 320 × 240, SIFT-GPU managed to realize a frame rate higher than our PR-SIFT. This could be explained that SIFT-GPU ignores some steps of the algorithm, like filtering out weak detected points [21]. This allows it to outperform all parallel implementations for low-quality images. However, as the resolution increases and without filtering weak points out, the number of detected points increases. SIFT-GPU uses atomic add instructions to sequentially store these points into the global interest points array. This makes SIFT-GPU to take considerable time to save all detected points. As a result, it could not outperform our PR-SIFT for higher resolutions. Furthermore, its performance even deteriorates as the image quality increases.

Although our implementation outperforms previous ones, in terms of average speedup and frame rate, we herein aim at statistically evaluating the spread of our ten runs around these average values. Therefore, we calculate the standard deviation and the coefficient of variation for all implementations and image quality. These two statistical metrics reflect how consistent an implementation is in achieving the aforementioned speedup and frame rate. The lower the value of the coefficient of variation, the more consistent the implementation is. Table 1 shows these two metrics for the frame rate results in Fig. 16. For any image quality, S and C represent the standard deviation and the percentage coefficient of variation, respectively. The table clarifies that, for any image quality, our PR-SIFT has the lowest coefficient of variation and hence is the most consistent one. It also signifies that our implementation keeps its consistency as the image quality increases. The percentage coefficient of variation of our implementation only increases from 3.08 to 3.3% as the image quality changes from 320 × 240 up to 1280 × 1024. Nevertheless, the percentage coefficient of variation of previous parallel implementations might quadruple, as for OpenCL SIFT, for the same change in image quality.

6.1.2 SURF results

Similar to the SIFT algorithm, we first verify the efficiency of our parallel SURF implementation with respect to the sequential algorithm. For different image resolutions,

Table 1 Standard deviation and coefficient of variation of different parallel SIFT implementations and image quality

	320 × 240		640 × 480		720 × 1280		1280 × 1024	
	S	C	S	C	S	C	S	C
PR-SIFT	3.50	3.08	2.21	3.09	1.39	3.45	0.81	3.30
CUDA-SIFT	2.77	4.04	2.08	6.34	1.53	6.35	1.32	6.60
OpenCL SIFT	4.04	3.68	3.16	4.58	2.62	7.08	1.56	6.50
SIFT-GPU	7.32	4.90	5.61	7.85	2.89	10.40	1.70	9.00
OpenCL SIFT	3.80	4.27	2.65	9.36	1.54	10.85	1.23	16.40

S is the standard deviation, whereas *C* is the % coefficient of variation

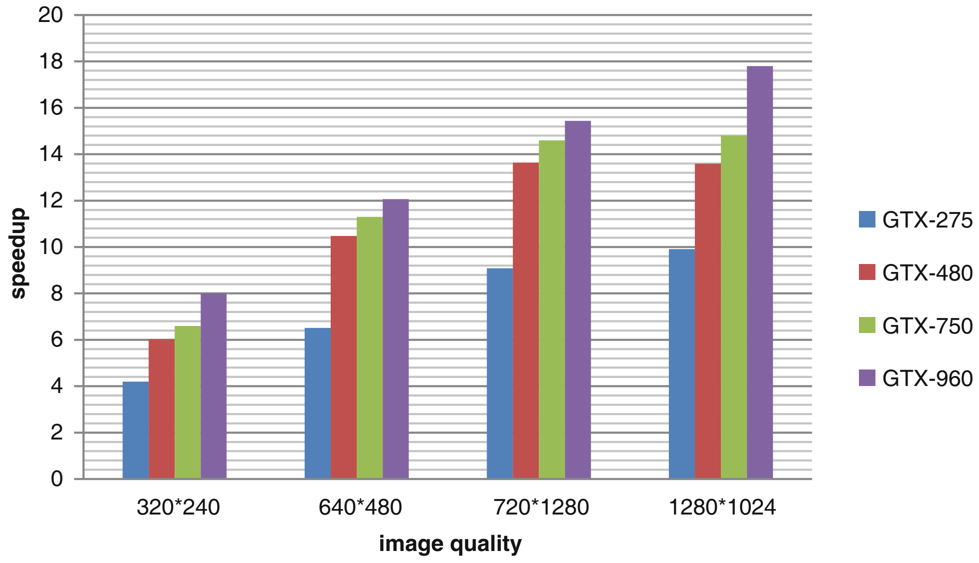


Fig. 17 Speedup achieved by PR-SURF over the sequential SURF for different GPUs and image quality

Fig. 17 shows the speedup achieved by our PR-SURF over the sequential SURF algorithm, for the same previously mentioned GPUs. The figure clarifies that the speedup continuously grows up by increasing either the number of cores within the GPU or the image resolution. Accordingly, for an image of 1280 × 1024 resolution, PR-SURF, running on GTX-960, achieves a maximum speedup of 18 × with respect to sequential SURF that runs on the Xeon CPU.

In order to evaluate the efficiency of our SURF parallel implementation in real-time applications, Fig. 18 shows the realized frame rate, for the same GPUs and image quality. The figure first clarifies that our PR-SURF results in a high frame rate for all GPUs and image resolutions. For example, for GTX-960, our implementation achieves a frame rate that ranges from 230 fps, for the lowest resolution, to 49 fps, for the highest resolution. This proves that our implementation could efficiently be used in real-time applications. Moreover, the figure shows that the achieved frame rate decreases as the number of cores within the GPU decreases or the image resolution increases.

Finally, we compare the frame rate resulted from PR-SURF to that of previous parallel implementations, which are reviewed in Sect. 3.2. We consider two GPU-based implementations, OpenCL SURF [8] and CSURF [7], and one CPU-based implementation, AD-PIPE, for comparison. For the GTX-750 GPU, Fig. 19 shows the frame rate resulted from the four implementations. The figure first clarifies that all GPU-based implementations significantly outperform the CPU-based one. Moreover, our own implementation accomplishes a higher frame rate than all other parallel implementations. The percentage enhancement in the frame rate of our implementation with respect to that of previous implementations grows up as the image quality increases. For example, for an image resolution of 1280 × 1024, our PR-SURF achieves 1.1 and 1.14 higher frame rates with respect to CSURF and OpenCL SURF, respectively.

Similar to the SIFT algorithm, we finally evaluate the spread of the performed ten runs around the average values and the consistency of our implementation in achieving the aforementioned speedup and frame rate. Consequently, Table 2 represents the standard deviation (*S*) and the percentage coefficient of variation (*C*) for the frame rate results in Fig. 19. The table clearly shows that our PR-SURF has the lowest coefficient of variation. This again

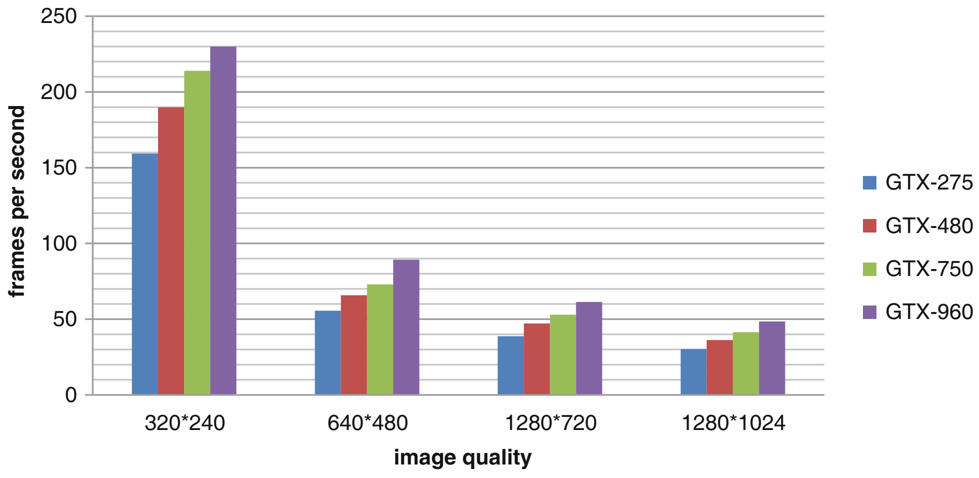


Fig. 18 Frame rate resulted from PR-SURF for different GPUs and image quality

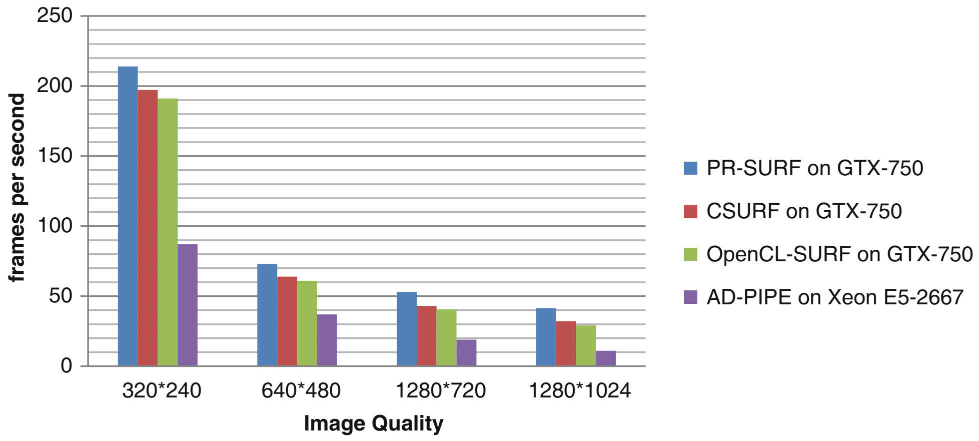


Fig. 19 Frame rate resulted from different parallel implementations of the SURF algorithm

Table 2 Standard deviation and coefficient of variation of different parallel SURF implementations and image quality

	320 × 240		640 × 480		720 × 1280		1280 × 1024	
	S	C	S	C	S	C	S	C
PR-SURF	3.96	1.86	2.44	3.48	2.01	4.03	1.92	4.87
CSURF	4.12	2.08	3.93	6.15	3.27	7.61	2.96	9.19
OpenCL SURF	3.79	1.98	3.28	5.37	2.84	7.00	2.43	8.33
AD-PIPE	1.94	2.23	1.78	4.81	1.24	6.53	1.21	11.03

S is the standard deviation, whereas *C* is the % coefficient of variation

emphasizes that our implementations are the most consistent in achieving the presented speedup and frame rate. The table also shows that this consistency becomes more significant as the image quality increases. Accordingly, for an image quality of 1280 × 1024, the coefficient of variation of PR-SURF is 1.89, 1.71, and 2.26 lower than that of CSURF, OpenCL SURF, and AD-PIPE, respectively.

6.2 Accuracy

In this subsection, we evaluate the accuracy of our SIFT and SURF implementations in detecting interest points. Before presenting our results, we quickly introduce the Mikolajczyk dataset [34], which is used throughout this subsection. Mikolajczyk dataset is a transformation-type one that contains eight image sets. Each set consists of six images. These are the original image and five transformed variants from this original image. Consequently, the five transformed images are obtained by introducing changes of different degrees into the blur radius, the lightening, the view angle, the rotation, the zoom, and the compression ratio. For seven image sets, these changes are done physically by adjusting the camera focus, brightness, position, angle, and zoom. For the eighth image set, the changes are introduced synthetically by increasing the compression ratio. Nevertheless, for the blur radius, the lightening, the zoom, the rotation, and the compression ratio, authors do not explicitly specify the magnitude of changes they introduce in each transformed image. For the view angle, changes are introduced by moving the camera from the view center position by 20°, 30°, 40°, 50°, and 60°. Throughout this subsection, we present results of the eight image sets. These are the Leuven image set, in which the lightening decreases, the Bikes and Trees image sets, in which the blur radius increases, the Graffiti and Wall image sets, in which the view angle increases in a step of 10°, the Bark and Boat image sets, in which both the rotation and zoom increases, and the UBC image set, in which the compression ratio increases.

6.2.1 For SIFT

In order to evaluate the accuracy of our implementation of SIFT algorithm, we first calculate the total number of detected interest points resulted from all parallel implementations. Over all images within the eight aforementioned image sets, Table 3 presents the average number of detected interest points from all implementations.

As the total number of detected interest points might be misleading, we further test our detection accuracy using true-positive points only. These are the points which exist at the same locations of interest points resulted from the sequential algorithms, or displaced from these locations by a maximum of one pixel. Thereafter, we validate our implementations using two quantitative metrics, the precision and the recall [35]. For each of the five transformed images within an image set, the two metrics represent the ratio of the number of true-positive points resulted from applying an implementation onto that transformed image to the total number of detected points. While precision considers total points resulted from applying the implementation itself, recall considers the sequential algorithm instead. Over all images within the eight considered image sets, Figs. 20 and 21 represent the resulted average precision and recall of different SIFT implementations. The two figures clearly signify the accuracy of our implementation. Accordingly, the least calculated precision and recall of our implementation are 77 and 70% for the Wall image set, respectively. Furthermore, the figures show that our PR-SIFT has the highest accuracy over all other parallel SIFT implementations, except for OpenCL SIFT on the Boat image set. Finally, the figures clarifies that SIFT-GPU is the least accurate parallel SIFT implementation. Its precision and recall go down to 69 and 58% for the Wall image set, respectively.

Once more, using true-positive points might not be the most accurate measure of accuracy. Therefore, for each transformed image within an image set, we then evaluate our accuracy by calculating the percentage matching accuracy [34]. Percentage matching accuracy represents the ratio of the correct matched points, resulted from applying an implementation on a transformed image, to the total number of detected points, resulted from applying the same implementation on the original image. Figure 22 represents the calculated percentage matching accuracy of different SIFT implementations. The five sub-figures represent an image set in which the lightening decreases, Leuven, an image set in which the blur radius increases, Bikes, an image set in which the view angle increases, Graffiti, an image set in which the compression ratio increases, UBC, and an image set in which both the rotation and zoom increases, Boat, respectively. For each sub-figure, the five ticks within the horizontal axis represent the five transformed images within the image set. The figure first shows that SIFT-GPU is the least accurate implementation. Over almost all images with image sets, the figure further emphasizes that our PR-SIFT has the highest percentage matching accuracy over all other implementations.

6.2.2 For SURF

Similar to the SIFT algorithm, the number of detected interest points resulted from all SURF implementations are

Table 3 Total number of detected interest points of different SIFT implementations (presented numbers are averages over all images within the image set)

Image set	Implementation				
	SIFT	PR-SIFT	CUDA-SIFT	SIFT-GPU	OpenCL SIFT
Bark	1812	1760	1725	1691	1657
Boat	1903	1822	1849	1553	1709
Leuven	984	933	961	838	863
Bikes	946	910	832	755	921
Trees	2289	2271	2362	2333	2216
UBC	1749	1714	1774	1488	1423
Graffiti	2619	2580	2422	2219	2453
Wall	2306	2244	2177	1981	2377

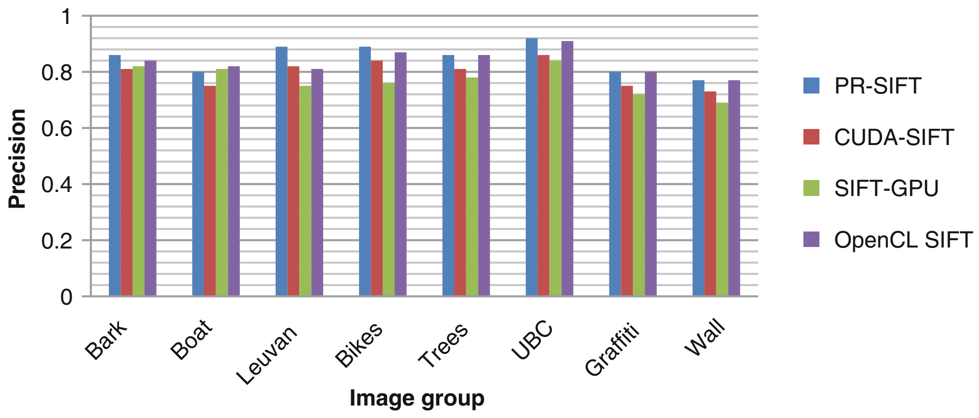


Fig. 20 Average precision of different SIFT parallel implementations (average over all images within the image set)

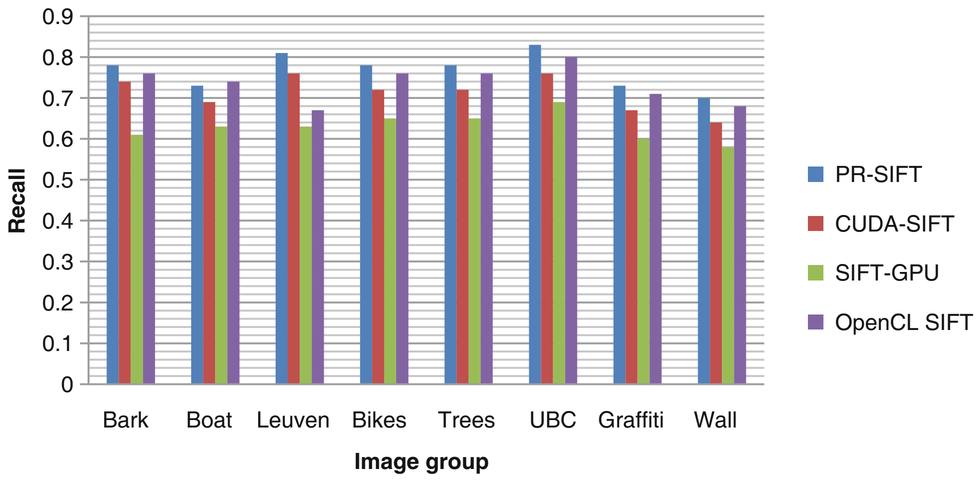


Fig. 21 Average recall of different SIFT parallel implementations (average over all images within the image set)

found. Over all images within an image set, Table 4 shows the average number of detected interest points of SURF, PR-SURF, OpenCL SURF, and CSURF. Thereafter, the average precision and recall of all implementations are calculated and are shown in Figs. 23 and 24, respectively. The two figures again emphasize the accuracy of our implementations. The least recorded precision and recall are found to be 79.5 and 68% for the Wall image set, respectively. The two figures also clarify that our implementation outperforms other parallel SURF implementations with respect to both precision and recall, except for OpenCL SURF on Bark and Trees image sets.

Similar to Fig. 22, the percentage matching accuracy is calculated for different lightening, blur radii, view angles, compression ratios, and both rotations and zooms. Figure 25 shows the resulted percentage accuracy of different SURF implementations. The figure clarifies that CSURF is the least accurate implementation. Moreover, it again shows that our PR-SURF significantly outperforms the two other parallel SURF implementations.

6.2.3 Example images

In this subsection, we aim at visually showing the accuracy of our implementation in detecting interest points, under aforementioned changes. As a proof of concept, we suffice with one transformed image from three image sets, Leuven, Bikes, and Graffiti. Accordingly, Fig. 26 presents sample results of our SIFT and SURF implementations. Sub-figures (a), (b), and (c) show interest points detected by our PR-SIFT, whereas sub-figures (d), (e), and (f) show these points for our PR-SURF. Each sub-figure has two images. The image to the left is the original image within the corresponding image set, whereas that to the right is the fourth transformed variant, in which the blur radius, the lightening, or the view angle is changed. Yellow lines connect between correctly matched interest points of the two images. Consequently, more yellow lines indicate better matching accuracy. In detail, sub-figures 26a and d connect between matched points of original and Blur 5 images of the Leuven image set. The percentage matching accuracies corresponding to these sub-figures are 94 and 90%, respectively. This is reflected by a large number of yellow lines. Similarly, Fig. 26b and e connects between

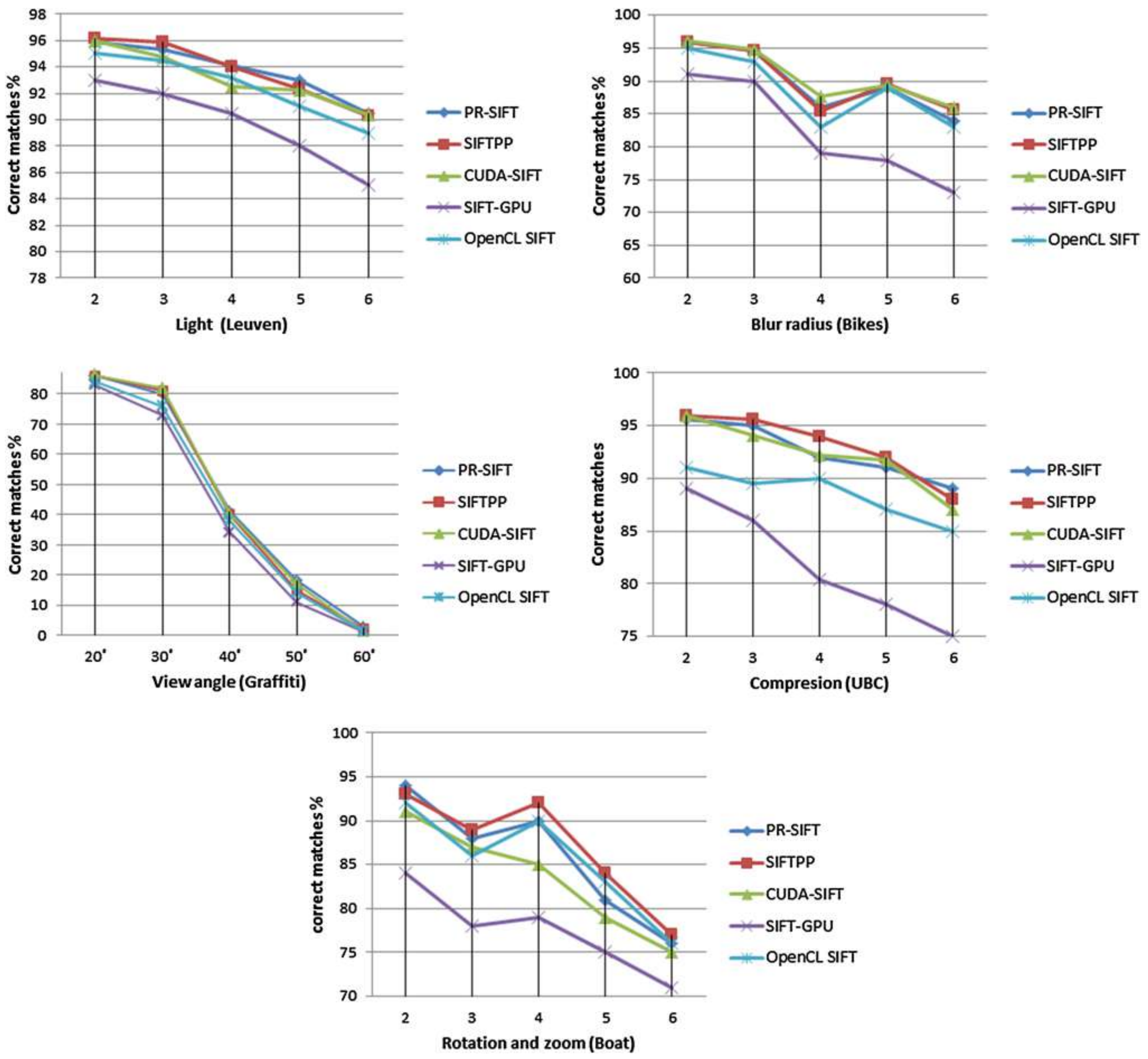


Fig. 22 Percentage matching accuracy of different SIFT implementations for different lightening, blur radii, view angles, compression ratios, and rotations/zooms

matched points of original and Light 5 images of the Bikes image set. The percentage matching accuracies corresponding to these sub-figures are 87 and 83%, respectively. Therefore, the number of yellow line is slightly less than that of the Leuven image set. Finally, Fig. 26c and f connects between matched points of original and 40° transformed images of the Graffiti image set. The percentage matching accuracies corresponding to these sub-figures are 43 and 61%, respectively. This is reflected by a lower number of yellow lines in the two sub-figures. In a nutshell, Fig. 26 visually clarifies that our implementations have good detection accuracies when changes are introduced into an image. Therefore, at the end of this subsection, we could conclude that our implementations manage to speed up both SIFT and SURF algorithms with minimal effect on their detection accuracy.

6.3 SM occupancy

In this subsection, we aim at evaluating the SM occupancy of our implementations with respect to that of previous CUDA-based ones. SM occupancy depends mainly on the logical organization of each implementation. Throughout the execution of SIFT and SURF algorithms, this organization changes from one stage to another. Consequently, for all CUDA-based implementations, we find the SM

Table 4 Total number of detected interest points of different SURF implementations

Image set	Implementation			
	SURF	PR-SURF	OpenCL SURF	CSURF
Bark	905	926	798	824
Boat	930	1028	828	947
Leuven	482	476	507	553
Bikes	473	478	477	430
Trees	1389	1356	1229	1445
UBC	903	921	937	881
Graffiti	1515	1475	1506	1538
Wall	1145	1127	1099	1087

Presented numbers are averages over all images within the image set

occupancy in each stage of the two algorithms. First, in Sect. 6.3.1, we calculate the theoretical SM occupancy according to the logical organization of each implementation. This theoretical SM occupancy represents a ceiling that could not be exceeded by the implementation. However, it is not guaranteed during runtime due to other factors, which might affect the occupancy, like the load imbalance. Therefore, in Sect. 6.3.2, we present the runtime SM occupancy of different implementations in different stages. This runtime SM occupancy is extracted from NVIDIA profiler toolkit. NVIDIA programming guide names the runtime SM occupancy the achieved occupancy. Accordingly, for the rest of this subsection, we adopt NVIDIA name for that term. Moreover, for simplicity and as being logically implied, we drop the SM letters from both occupancies. Finally, obtained results show that our

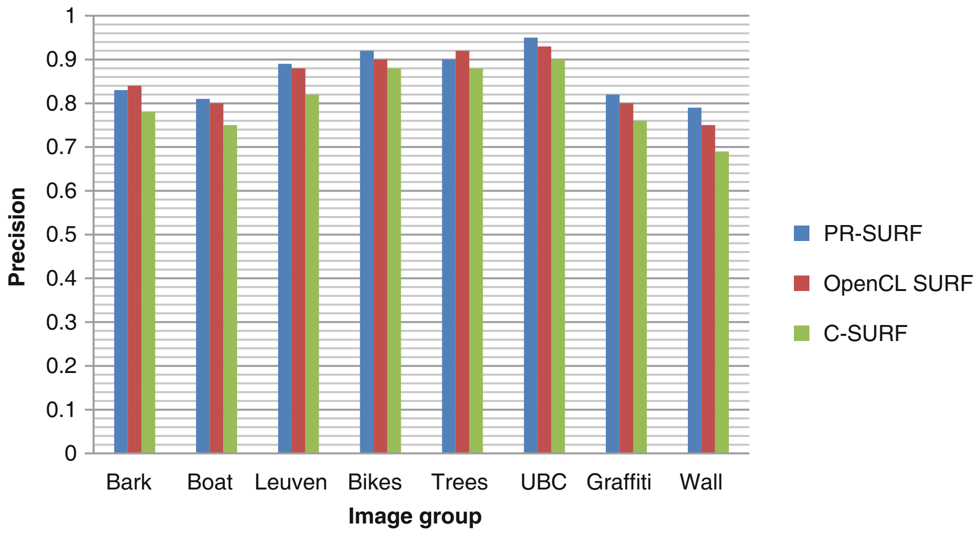


Fig. 23 Average precision of different SURF parallel implementations (average over all images within the image set)

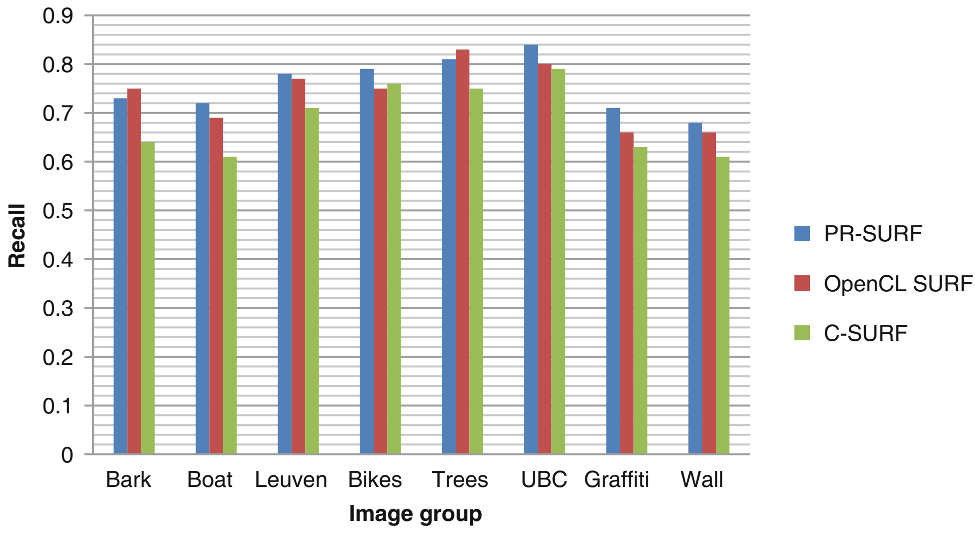


Fig. 24 Average recall of different SURF parallel implementations (average over all images within the image set)

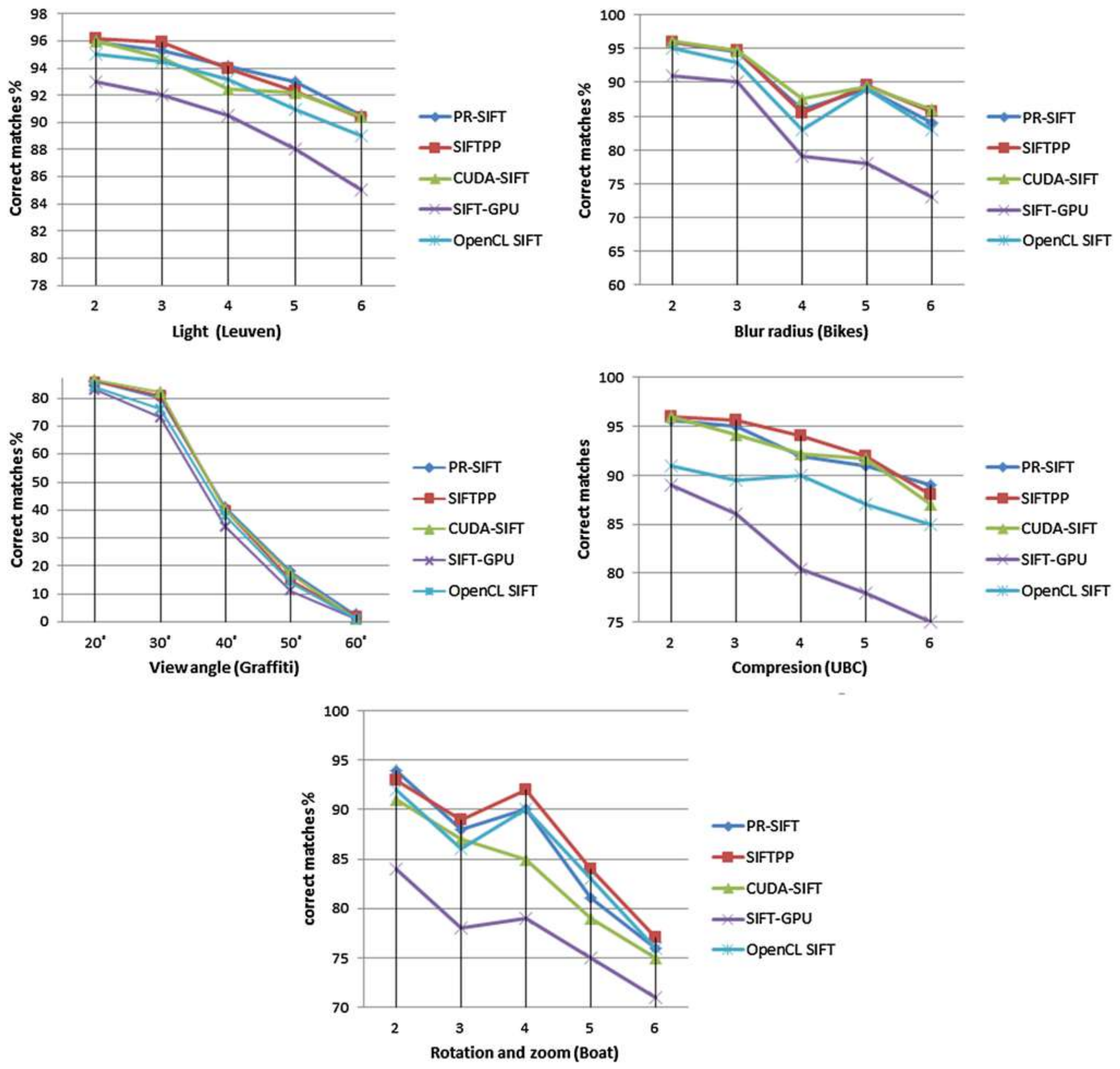


Fig. 25 Percentage matching accuracy of different SURF implementations for different lightening, blur radii, view angles, compression ratios, and rotations/zooms

implementations outperform other CUDA-based ones with respect to both theoretical and achieved occupancies.

6.3.1 Theoretical occupancy

As explained in Sect. 4, theoretical occupancy is the ratio of the number of warps generated by an implementation to the maximum number of warps that could run on this SM. SM occupancy enhances if more threads, and hence, warps, are generated by the presented implementation. The maximum number of threads, blocks, warps, and registers per SM is decided according to the compute capability version of the GPU. For example, for GTX-480 with compute capability 2.0, a maximum of 8 blocks, 48 warps, and 32 k registers are allowed per SM. Each warp consists of 32 threads. In our implementation of the descriptor construction stage of both SIFT and SURF algorithms, we use a block organization of 8 $\times$ 16 threads. This results in a total of 128 threads, which are equivalent to 4 warps per each block. On the other hand, the number of employed blocks is limited by the maximum number of registers. Our kernel requires 29 registers for each thread. Therefore, a total of 3712 registers, i.e., 128 $\times$ 29, are needed for each block. Dividing the maximum allowable number of registers,

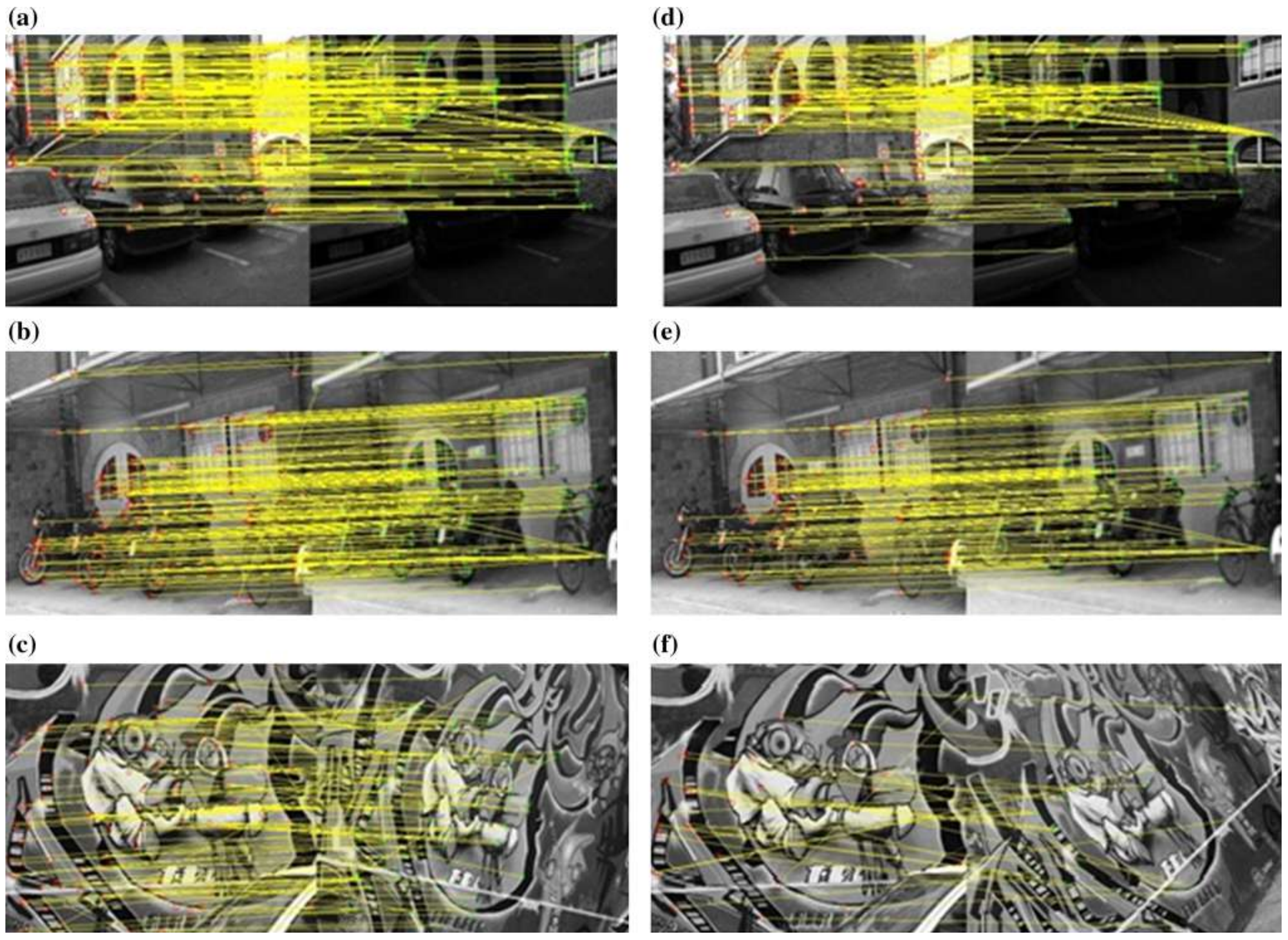


Fig. 26 Detection accuracy of PR-SIFT and PR-SURF for different blur radii, lightening, and view angles. Original image is at the left, and the transformed one is at the right. Yellow lines connect between matched points of the two images. **a** Matched points between original and Blur 5 images of the Leuven image set using PR-SIFT. **b** Matched points between original and Light 5 images of the Bikes image set using PR-SIFT. **c** Matched points between original and 40° transformed images of the Graffiti image set using PR-SIFT. **d** Matched points between original and Blur 5 images of the Leuven image set using PR-SURF. **e** Matched points between original and Light 5 images of the Bikes image set using PR-SURF. **f** Matched points between original and 40° transformed images of the Graffiti image set using PR-SURF

32 k, by 3712 gives 8.8. Therefore, 8 blocks are used in our kernel. The total number of warps for our kernel is accordingly 32 warps, i.e., 8 blocks × 4 warps per block. As the maximum number of possible warps is 48, our theoretical occupancy is 66.6%.

In a way similar to that described in the previous paragraph, the theoretical occupancy of PR-SIFT, CUDA-SIFT, SIFT-GPU, PR-SURF, and CSURF is calculated for all stages of the two algorithms. It is worth mentioning that, for GTX-275, a maximum of 8 blocks, 32 warps, and 16 k registers are allowed per SM. For GTX-750 and GTX-960, a maximum of 32 blocks, 64 warps, and 64 k registers are allowed per SM. Accordingly, for the two resource-rich GPUs with compute capability 5.×, all algorithms achieve a theoretical occupancy of 100% in all stages. These two GPUs put no restrictions on the logical organization of any implementation. On the other hand, for the other two resource-limited GPUs, our implementations outperform their counterparts with respect to the theoretical occupancy. For GTX-275 and GTX-480, Tables 5, 6, 7, and 8 summarize the resultant theoretical occupancies for the scale space construction, interest points detection, orientation assignment, and descriptor construction stages,

Table 5 Theoretical occupancy of different parallel implementations of SIFT and SURF algorithms, for the scale space construction stage

	Compute capability 1.×	Compute capability 2.×
PR-SIFT	62%	83%
CUDA-SIFT	62%	83%
SIFT-GPU	50%	67%
PR-SURF	62%	83%
CSURF	On CPU	On CPU

Table 6 Theoretical occupancy of different parallel implementations of SIFT and SURF algorithms, for the interest points detection stage

	Compute capability 1.× (%)	Compute capability 2.× (%)
PR-SIFT	62	83
CUDA-SIFT	56	79
SIFT-GPU	62	83
PR-SURF	62	83
CSURF	50	83

Table 7 Theoretical occupancy of different parallel implementations of SIFT and SURF algorithms, for the orientation assignment stage

	Compute capability 1.× (%)	Compute capability 2.× (%)
PR-SIFT	50	67
CUDA-SIFT	50	67
SIFT-GPU	28	56
PR-SURF	62	83
CSURF	62	83

Table 8 Theoretical occupancy of different parallel implementations of SIFT and SURF algorithms, for the descriptor construction stage

	Compute capability 1.× (%)	Compute capability 2.× (%)
PR-SIFT	50.00	66.6
CUDA-SIFT	33.00	58.3
SIFT-GPU	33.00	56.4
PR-SURF	62.50	66.6
CSURF	56.25	58.0

respectively. The tables clarify that our implementations achieve higher or at least the same theoretical occupancy over all other CUDA-based implementations. In other words, our implementations outperform other CUDA-based implementations with respect to theoretical occupancy in at least two out of the four stages of the algorithms. Furthermore, for the descriptor construction stage, our implementations outperform all other CUDA-based implementations. For example, for compute capability 2.×, our PR-SIFT has 8.3 and 10.2% higher theoretical occupancy than CUDA-SIFT and SIFT-GPU, respectively. PR-SURF also achieves an 8.6% higher theoretical occupancy than CSURF.

6.3.2 Achieved occupancy

As mentioned at the beginning of Sect. 6.3, other factors, like the load imbalance, might result in an achieved occupancy lower than the theoretical one. In this subsection, we aim at evaluating the actual achieved occupancy of our implementations with respect to previous parallel ones. Therefore, we use the NVIDIA profiler toolkit to extract the achieved occupancy of all CUDA-based implementations. For GTX-275 and GTX-480, our implementations result in higher achieved occupancy. However, these results are excluded as the theoretical occupancy already reflected the higher performance of our implementations for these two GPUs. Nevertheless, for both GTX-750 and GTX-960, Fig. 27 shows the achieved occupancy resulted from all implementations in each stage of SIFT and SURF algorithms. The figure shows that our implementations realize higher achieved occupancy for almost all stages. The only exception happens in the orientation assignment stage, in which SIFT-GPU realizes 1.9% more achieved occupancy than ours. However, our PR-SIFT results in 7, 26, and 6.3% higher achieved occupancy than SIFT-GPU for scale space construction, interest points detection, and descriptor construction stages, respectively. Similarly, the achieved occupancy of PR-SIFT exceeds that of CUDA-SIFT by 14, 22, 3.6, and 2.9% for the scale space construction, interest points detection, orientation assignment, and descriptor construction stages, respectively. Finally, our PR-SURF outperforms CSURF by 26, 3, and 3.5% for the same last three stages, respectively.

7 Conclusion and future work

This work presents new CUDA-based parallel implementations of two spatiotemporal algorithms for image features extraction: SIFT and SURF. The proposed implementations decrease the processing time of the two algorithms by solving problems in previous parallel implementations, like load imbalance, low SM occupancy, and the use of atomic

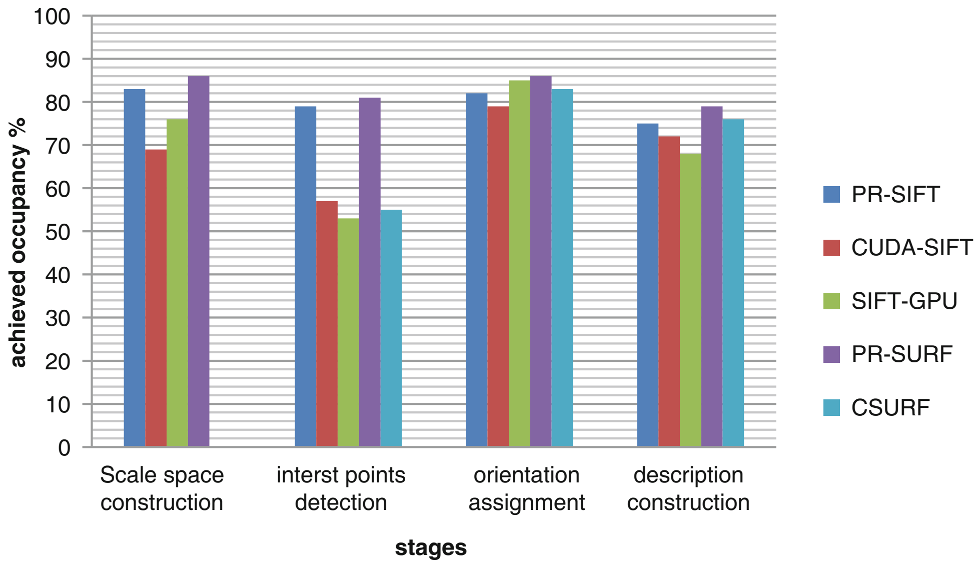


Fig. 27 Achieved occupancy of different parallel implementations of SIFT and SURF algorithms, in different stages of the two algorithms for GPUs with compute capability 5.×

operations. The frame rate achieved by our implementations allows them to be efficiently used with real-time applications. Our implementations also have a minimal impact on the accuracy. Many experiments are conducted using an Intel Xeon CPU, E5-2667, and 4 different GPUs to verify the efficiency of our implementations. Results show that our implementations achieve higher speedup, higher frame rate, higher accuracy, and higher SM occupancy than previous parallel implementations of the two algorithms.

In future, we will try to extend the work presented in this paper in two directions. First, we would adjust our implementations to run on multiple connected GPUs. This would allow us to process images of very high resolution in real time. Second, as our implementations are currently customized to NVIDIA GPUs, we would present platform-independent ones that could be used with any other GPUs.

References

1. Lowe, D.: Distinctive image features from scale-invariant keypoints. Int. J. Comput. Vis. **60**(2), 91–110 (2004)
2. Laptev, I., Lindeberg, T.: Local descriptors for spatio-temporal recognition. Lect. Notes Comput. Sci. **3667**, 91–103 (2006)
3. Bay, H., Ess, A., Tuytelaars, T., Van Gool, L.: Speeded-up robust features (SURF). Comput. Vis. Image Underst. **110**(3), 346–359 (2008)
4. Lee, C., Rhee, C.E., Lee, H.-J.: Complexity reduction by modified scale-space construction in sift generation optimized for a mobile GPU. IEEE Trans. Circuits Syst. Video Technol. **27**(10), 2246–2259 (2017)
5. Zhang, Q., Chen, Y., Zhang, Y., Xu, Y.: SIFT implementation and optimization for multi-core systems. In: 2008. IPDPS 2008. IEEE International Symposium on Parallel and Distributed Processing, pp. 1–8. IEEE (2008)
6. Moren, K., Göhringer, D.: A framework for accelerating local feature extraction with OpenCL on multi-core CPUs and co-processors. J. Real-Time Image Process. **10**(1007), 1–18 (2016)
7. Zhu, F., Chen, P., Yang, D., Zhang, W., Chen, H., Zang, B.: A GPU-based high-throughput image retrieval algorithm. In: Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units. ACM30-37, (2012)
8. Yan, W., Shi, X., Yan, X., Wang, L.: Computing OpenSURF on OpenCL and general purpose GPU. Int. J. Adv. Robot. Syst. **10**(10), 375 (2013)
9. Lu, Y., Li, Y., Song, B., Zhang, W., Chen, H., Peng, L.: Parallelizing image feature extraction algorithms on multi-core platforms. J. Parallel Distrib. Comput. **92**, 1–14 (2016)
10. Luebke, D.: CUDA: scalable parallel programming for high-performance scientific computing. In: The 2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro (ISBI 2008). IEEE836-838, (2008)
11. Hwu, W.-M.W.: GPU Computing Gems Emerald Edition. Elsevier, Amsterdam (2011)
12. Brown, M., Lowe, D. G.: Invariant features from interest point groups. In: Proceedings of the British Machine Vision Conference 2002, BMVC, pp. 253–262. (2002)
13. Antonini, M., Barlaud, M., Mathieu, P., Daubechies, I.: Image coding using wavelet transform. IEEE Trans. Image Process. **1**(2), 205–220 (1992)
14. Heymann, S., Muller, K., Smolic, A., Frohlich, B., Wiegand, F.: SIFT implementation and optimization for general-purpose GPU. In: Proceedings of the International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, (2007)
15. Sinha, S. N., Frahm, J.-M., Pollefeys, M., Genc, Y.: GPU-based video feature tracking and matching. In: EDGE, Workshop on Edge Computing Using New Commodity Architectures, vol. 278, p. 4321. (2006)
16. Sinha, S., Frahm, J.-M., Pollefeys, M., Genc, Y.: Feature tracking and matching in video using programmable graphics hardware. Mach. Vis. Appl. **22**(1), 207–217 (2007)
17. Wu, C.: SiftGPU: a GPU implementation of scale invariant feature transform, https://github.com/pitzer/SiftGPU (2012)
18. Vedaldi, A.: An open implementation of the SIFT detector and descriptor. UCLA CSD, http://vision.ucla.edu/~vedaldi/code/sift.html (2007)

19. Yonglong, Z., Kuizhi, M., Xiang, J., Peixiang, D.: Parallelization and optimization of sift on GPU using CUDA. In: 2013 IEEE 10th International Conference on High Performance Computing and Communications, The 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), IEEE1351-1358, (2013)
20. Mohammadi, M., Rezaeian, M.: Towards affordable computing: SiftCU a simple but elegant GPU-based implementation of SIFT. Int. J. Comput. Appl. **90**(7), 30–37 (2014)
21. Acharya, K., Babu, R. V., Vadhiyar, S. S: A real-time implementation of SIFT using GPU. J. Real-Time Image Process. 1–11 (2014). https://doi.org/10.1007/s11554-014-0446-6
22. Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with CUDA. GPU Gems **3**(39), 851–876 (2007)
23. Terriberry, T., French, L., Helmsen, J.: GPU accelerating speeded-up robust features. In: Proceedings of 3DPVT. p. 355–362. (2008)
24. Blelloch, G.: Prefix sums and their applications. In: J.H. Reif (ed). Synthesis of Parallel Algorithms, Morgan Kaufmann Publishers Inc. San Francisco, CA, USA (1993)
25. Bilgic, B., Horn, B. K., Masaki, I.: Efficient integral image computation on the GPU. In: Intelligent Vehicles Symposium (IV), 2010 IEEE, IEEE528-533, (2010)
26. Fang, Z., Yang, D., Zhang, W., Chen, H., Zang, B.: A comprehensive analysis and parallelization of an image retrieval algorithm. In: 2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE154-164, (2011)
27. Schulz, A., Jung, F., Hartte, S.: CUDA SURF: a real-time implementation for SURF. https://www.d2.mpi-inf.mpg.de/surf (2011)
28. Cheon, S., Eom, I.K., Ha, S.W., Moon, Y.H.: An enhanced SURF algorithm based on new interest point detection procedure and fast computation technique. J. Real-Time Image Process (2016). https://doi.org/10.1007/s11554-016-0614-y
29. Hong, S., Kim, H.: An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In: ACM SIGARCH Computer Architecture News, ACM.37, 3, pp. 152–163. (2009)
30. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Elsevier, Amsterdam (2011)
31. Nvidia: NVIDIA Tesla P100: the most advanced datacenter accelerator ever built, featuring pascal GP100, the world's fastest GPU, In: whitepaper. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf
32. C. Nvidia: C Programming Guide v9. 1. Nvidia Corporation, Santa Clara (2017)
33. Barandiaran, I., Cortes, C., Nieto, M., Grana, M., Ruiz, O. E.: A new evaluation framework and image dataset for keypoint extraction and feature descriptor matching. In: Proceedings of the International Conference on Computer Vision Theory and Applications (VISAPP). vol 1, pp. 252–257. (2013)
34. Mikolajczyk, K., Schmid, C.: A performance evaluation of local descriptors. IEEE Trans. Pattern Anal. Mach. Intell. **27**(10), 1615–1630 (2005)
35. Van Rijsbergen, C.: Information Retrieval. vol 14, Department of Computer Science, University of glasgow. citeseer.ist.psu.edu/vanrijsbergen79information.html (1979)

Ahmed Mehrez obtained his bachelor's degree from Faculty of Engineering at Shoubra, Benha University. From 2012 to date, he works as a Teaching Assistant at Benha University.

Ahmed A. Morgan received the Ph.D. degree from the University of Victoria, Victoria, BC, Canada, in 2011, and the B.Sc. degree (first class honors) and the M.Sc. degree from the Faculty of Engineering at Shoubra, Benha University, Egypt, in 2000 and 2005, respectively. He is an Assistant Professor in the Department of Computer Engineering, Cairo University, Egypt. He is currently in a leave at the college of Computers and Information Systems, Umm Al-Qura University, Makkah, Saudi Arabia. His research interests include parallel architectures, multicore systems, digital VLSI design, wireless sensor networks, and network-on-chip (NoC) modeling, optimization, and performance evaluation. He has about 25 publications that span journals, conferences, book chapters, and technical reports.

Dr. Hemayed is currently working as a Professor at Cairo University and at Zewail City for Science and Technology. He got his Ph.D. from University of Louisville, KY, in 1999. He got the Graduate Dean's Citation Award and the John M. Houchens Prize in recognition of excellent doctoral dissertation. He got his M.Sc. and his B.Sc. from Cairo University in 1992 and 1989. From 2007 to 2014, he worked as an Assistant Professor and then an Associate Professor at Cairo University. From 2004 to 2007, he worked as an Assistant Professor at the UAE University. From 1999 to 2005, he was working at Trendium, Florida. His last position was VP of Technical Services and Solution Development. He was awarded the year 2005 Trendium Pioneer. From 1989 to 1994, he worked as a Teaching Assistant at Cairo University. He is a senior member of IEEE and a member of ACM and has been a regular reviewer of international conferences and journals. He has published over 70 papers. His research interest includes computer vision, machine learning, and big data analytic.