J. Parallel Distrib. Comput. 124 (2019) 78–91

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Efficient parallel optimizations of a high-performance SIFT on GPUs

Zhihao Li [a,b], Haipeng Jia [a,*], Yunquan Zhang [a], Shice Liu [a,b], Shigang Li [a], Xiao Wang [a,b], Hao Zhang [c]

[a] State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[b] School of Computer and Control Engineering, University of Chinese Academy of Sciences, Beijing, China
[c] Shanghai Key Laboratory of Intelligent Information Processing, Fudan University, Shanghai, China

HIGHLIGHTS

- A high-performance SIFT implementation on GPUs.
- An FFT-based Gaussian convolution to accelerate the DoG pyramid construction stage.
- Rebalancing workloads and multigranularity parallelism to achieve load balancing.
- Two histogram algorithms to rapidly accumulate samples on different scales.
- The atomic-free multikey reduction to accelerate the trilinear interpolation.

ARTICLE INFO

Article history:
Received 9 November 2017
Received in revised form 12 September 2018
Accepted 23 October 2018
Available online 2 November 2018

Keywords:
HartSift
SIFT
GPU
High performance
Feature extraction

ABSTRACT

Stable local image feature detection is a fundamental problem in computer vision and is critical for obtaining the corresponding interest points among images. As a popular and robust feature extraction algorithm, the scale invariant feature transform (SIFT) is widely used in various domains, such as image stitching and remote sensing image registration. However, the computational complexity of SIFT is extremely high, which limits its application in real-time systems and large-scale data processing tasks. Thus, we propose several efficient optimizations to realize a high-performance SIFT (HartSift) by exploiting the computing resources of CPUs and GPUs in a heterogeneous machine. Our experimental results show that HartSift processes an image within 3.07∼7.71 ms, which is 55.88∼121.99 times, 5.17∼6.88 times, and 1.25∼1.79 times faster than OpenCV SIFT, SiftGPU, and CudaSift, respectively.

1. Introduction

Scale invariant feature transform [28,29] (SIFT) is acknowledged as a stable feature extraction algorithm. It extracts a large number of distinctive features that are well localized in the frequency and spatial domains. These features not only remain invariant to image rotation and scaling, but also offer robust matching across a substantial range of affine distortion, the addition of noise, changes in illumination, and three-dimensional (3D) viewpoints. Therefore, SIFT is widely applied to many fields related to computer vision, such as image stitching, remote sensing image registration, simultaneous localization and mapping (SLAM), structure from motion (SFM), and 3D reconstruction. Even though SIFT contains many competitive merits, its high computational complexity restricts its further application in large-scale data and real-time systems; therefore, many users have to consider other methods in their systems.

Other feature extraction algorithms with lower computational complexity are presented to accelerate the process of feature extraction, such as SURF [3], ORB [37], and other algorithms; however, these algorithms are less accurate compared to SIFT. Moreover, methods based on convolutional neural networks (CNN) [15,16,20,36] have attracted increasing attention recently, and have shown impressive performance when sufficient training data are provided. However, CNN-based methods are less competitive in the absence of training data or in an uncertain application environment, such as remote sensing image registration, SLAM, and SFM. Further, when the target images are gray-scale, CNN may be less effective than SIFT because the SIFT features are computed on gray-scale images without depending on color information. Hence, realizing a high-performance SIFT on GPUs is an important research topic.

Fortunately, the SIFT algorithm contains good parallelism; therefore, we can port SIFT to GPU platforms. A GPU is a substantial type of parallel processor, which provides tremendous computational power and high memory bandwidth [34]. More importantly, GPUs are widely used in the fields of computer vision. Therefore,

* Corresponding author.
E-mail address: jiahaipeng@ict.ac.cn (H. Jia).

https://doi.org/10.1016/j.jpdc.2018.10.012
0743-7315/

it is valuable for realizing high-performance SIFT by exploiting the computing power of heterogeneous systems equipped with GPUs.

The SIFT algorithm is composed of three stages: the construction of the difference-of-Gaussian (DoG) pyramid, the accurate keypoint localization, and the 128-dimensional descriptor generation. Because each stage contains its own characteristics, we adopt different optimizations and parallel granularities to achieve a high performance. Several efficient optimizations are applied to realize a high-performance SIFT, named HartSift. The overall performance of HartSift outperforms other renowned open-source SIFT implementations. Our experimental results demonstrate that HartSift can process images of different sizes within 3.07∼7.71 ms (129.7∼325.73 fps), which is 55.88∼121.99 times, 5.17∼6.88 times, and 1.25∼1.79 times faster than OpenCV SIFT (a CPU SIFT) [6], SiftGPU (a GPU SIFT) [41], and CudaSift (the fastest GPU SIFT) [4,5], respectively.

The key contributions and innovations of this study are summarized as follows:

- We developed a parallel SIFT implementation based on GPUs, satisfying the demanding requirements of high-performance computer vision applications.
- We implemented a fast Fourier transform (FFT)-based Gaussian convolution to accelerate the calculations of the Gaussian pyramid's large octaves on GPUs.
- We proposed two efficient optimizations (rebalancing workloads and multigranularity parallelism) to address load imbalance when porting SIFT to GPUs. Both optimizations are applicable for other load-imbalanced image processing algorithms.
- We introduced two high-performance histogram algorithms based on GPUs (the warp-based histogram algorithm and the atomic-free histogram algorithm) to rapidly accumulate samples information on different scales.

The remainder of this paper includes related works in Section 2, the description of the SIFT algorithm in Section 3, and the illustration of the optimization strategies in Section 4, followed by the experiments and discussions in Section 5. Finally, the conclusion of this paper will be presented in the last section.

2. Related works

The use of local image features in image matching can be traced back to Moravec [31]. Subsequently, Harris [18] improved the Moravec detector to use it in many image matching tasks; subsequently, various feature detection algorithms have emerged [14,17,27–30]. Each of these features remains invariant to scale, rotation, illumination, and affine distortion, which is important in image matching applications. SIFT has been proven to be superior to other features in invariance. However, many applications cannot tolerate SIFT's high computational complexity; thus, other low-precision algorithms are used to accelerate feature extraction, such as PCA-SIFT [23], SURF [3], ASIFT [32,43], and ORB [37].

To accelerate the SIFT algorithm, many have attempted to implement the SIFT algorithm on GPUs. Heymann [21] implemented a parallel SIFT and obtained 17.24 fps on the NVIDIA Quadro FX 3400 GPU for a 640 × 480 video. Heymann used texture memory and rearranged a gray image into a four-channel image to exploit the vector computing resources on GPUs. Warn [40] explored a parallel SIFT based on OpenMP and CUDA. However, only the DoG pyramid construction stage was placed on GPUs. Compared with their own serial SIFT version, the parallel implementation obtained a 1.9x acceleration on the NVIDIA Quadro FX 5800 GPU for a 4136 × 1424 image. Sinha's GPU-SIFT [38] obtained approximately 10x acceleration for a 640 × 480 image with 800 features compared with their own CPU SIFT on the NVIDIA GeForce 7800 GTX GPU. Sinha placed most stages of the SIFT on GPUs, except the 128D descriptor generation stage. SiftGPU [41] is a famous open-source project released by Wu. Inspired by Sinha's GPU-SIFT, Wu also utilized CUDA to implement a parallel SIFT. The CUDA version of SiftGPU-V400 obtained 27.22 fps on the NVIDIA GeForce 8800 GTX GPU for a 1280 × 768 image. Acharya proposed a GPU-based SIFT [1,2] that performed at approximately 13.36 fps for a 640 × 480 image on the NVIDIA Tesla S1070 GPU. Björkman's CudaSift [4,5] is the fastest SIFT on GPUs thus far. It processed a 1280 × 960 image (doubling the input image) in 13.2 ms (75.75 fps) on the NVIDIA GeForce GTX 580 GPU. Fassold [11] also realized a parallel SIFT to solve video tasks. This SIFT could process SD resolution images in approximately 40 ms. Our previous work [26] could process an image within 3.14∼10.57 ms (94.61∼318.47 fps). In this study, HartSift has been further improved based on the previous version.

Therefore, the primary purpose of this paper is to develop a parallel SIFT based on GPUs that can fulfill the demanding requirement of high-performance feature extraction.

3. Algorithm overview

This section briefly describes the SIFT algorithm based on Lowe's classic paper [29]. The overview of the SIFT algorithm is presented in Fig. 1.

3.1. Constructing difference-of-gaussian pyramid

To increase the number of keypoints, Lowe first doubles (upsamples) the input image as the lowest layer of the Gaussian pyramid, denoted as $I(x, y)$. Each scale, denoted as $L(x, y, \sigma)$, is produced by adopting a Gaussian convolution of a variable scale $G(x, y, \sigma)$:

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y) \tag{1}$$

where $*$ is the convolution operator of x and y; σ is the scale coordinate in the scale space.

The Gaussian pyramid, as shown in Fig. 1(a), is built sequentially using Eq. (1). The number of pyramid octaves O is determined by the size of the input image, and each octave contains $S + 3$ layers. Subsequently, the scale space contains $O \times (S+3)$ images. The first layer of each octave is produced by downsampling the Sth layer of the previous corresponding octave. For the same octave, the ith layer image is generated by applying the Gaussian convolution on the $(i-1)$th layer.

As shown in Fig. 1(b), when the Gaussian pyramid is established, the adjacent images will be subtracted to generate the DoG pyramid with $O \times (S+2)$ images:

$$D(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma) \tag{2}$$

where $k = 2^{(1/S)}$, in our case, $S = 3$.

The parameter σ of the Gaussian kernel in Eq. (2) is set to $2^{i-1}(k^{n-1}\sigma_0)$ and $n \in [0, O-1]$. The images in the first octave from the bottom up of the DoG pyramid contain these scales: σ_0, $k\sigma_0$, $k^2\sigma_0$, $k^3\sigma_0$, $k^4\sigma_0$, and the middle images with $k\sigma_0$, $k^2\sigma_0$, $k^3\sigma_0$ are used to detect the keypoint candidates. Similarly, the chosen scales of the next octave are $2k\sigma_0$, $2k^2\sigma_0$, $2k^3\sigma_0$. Because $k^3 = 2^{3/3}$, $2k = 2^{4/3}$, each chosen scale satisfies the consecutiveness of the scale space with the same scale variation $k = 2^{1/3}$.

3.2. Accurate keypoint localization

This stage contains three parts: detecting keypoint candidates, removing invalid keypoint candidates, and orientation assignment.

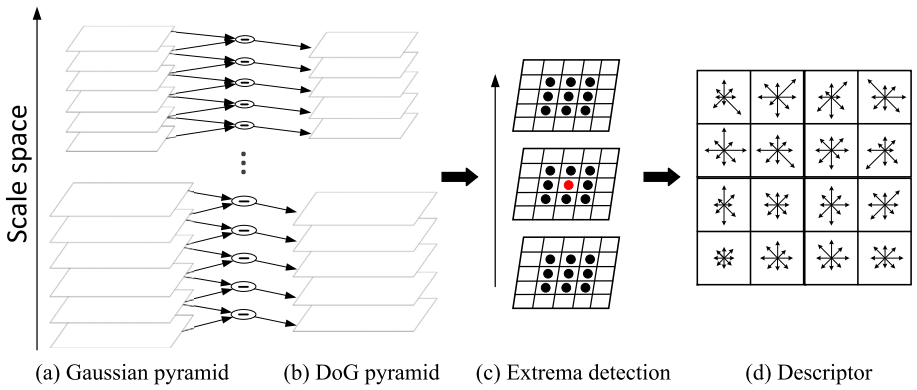

Fig. 1. The Overview of the SIFT algorithm.

3.2.1. Detecting keypoint candidates

This part compares each pixel of the DoG pyramid with its neighbor pixels. Eight pixels exist in the same scale, nine pixels in the upper scale, and nine pixels in the lower scale (see Fig. 1(c)). If the pixel is a local maximum or minimum, it will be regarded as a keypoint candidate.

3.2.2. Removing invalid keypoint candidates

To remove keypoint candidates with low contrast and unstable edge response, this step performs a subpixel interpolation by fitting a 3D quadratic function to the nearby data for their accurate location, scale, and the ratio of the principal curvatures.

3.2.3. Orientation assignment

To maintain invariance to image rotation, this part contains two steps: histogram generation and dominant orientation assignment. First, we need to accumulate the gradients of the neighbors around a keypoint to a 36-bin histogram. The radius of the neighbor region is calculated by σ_{octave} (the scale in the octave) of Eq. (3). Each neighbor is weighted by a Gaussian-weighted circular window with 1.5σ. Gradient values with the weight of the neighbors will be accumulated to a 36-bin orientation histogram. Second, we smooth the histogram and search for the peak bin. The peak bin is the dominant gradient orientation of the keypoint. Meanwhile, other local peak bins that are over 80% of the peak one are also regarded as keypoints with their own orientations.

$$radius = 3 \times 1.5 \times \sigma_{octave} \tag{3}$$

3.3. Generating descriptor

After completing the previous stage, keypoints are assigned x, y, scale, and orientation. This stage transforms the keypoints and their neighbors' information into distinctive descriptors to represent the local properties of the keypoints. The neighbor region of each keypoint is partitioned into 4×4 subregions. Each subregion contains an orientation histogram with eight bins. The neighbor region at this stage is much larger than that of the previous stage, as shown in Eq. (4) and Lowe set $d = 4$. A keypoint descriptor is generated by first computing the gradient magnitude and orientation of each neighbor in the region; subsequently, a Gaussian window is used to weigh the image gradients, and a trilinear interpolation is adopted to distribute the gradients into the orientation histogram, as shown in Fig. 1(d). Because the result index of the trilinear interpolation may be outside the boundaries of each dimension, a $4 \times 4 \times 8 = 128$-bin histogram is insufficient to implement the trilinear interpolation. Thus, we expand the boundaries in each dimension, and the $(4+2)\times(4+2)\times(8+2) = 360$-bin histogram is introduced. After completing the trilinear interpolation, each neighbor votes for 8 bins in the 360-bin histogram. The 360-bin histogram will be transformed into the $4 \times 4 \times 8$=128-dimensional descriptor, which will be further normalized to reduce the effects of illumination change.

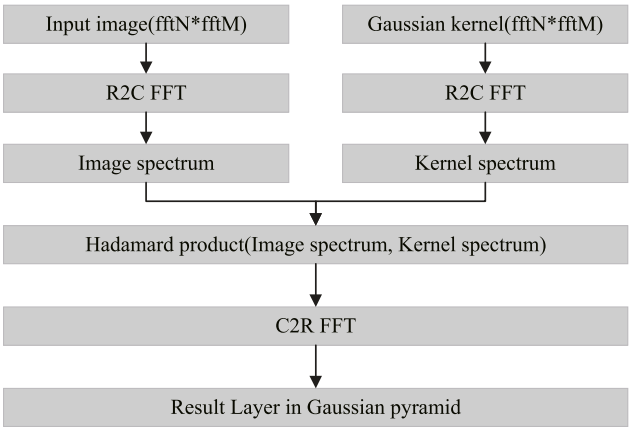

Fig. 2. The FFT-based Gaussian convolution.

$$radius = \frac{3\sigma_{octave} \times \sqrt{2} \times (d+1)}{2} \tag{4}$$

4. Parallel strategies and implementations

This section describes and analyzes the optimizations adopted in each stage of the HartSift in detail.

4.1. Constructing difference-of-gaussian pyramid

4.1.1. Parallelism analysis

As discussed in Section 3.1, the core operation at this stage is the Gaussian convolution kernel. The Gaussian convolution kernel will be called multiple times during the construction of the Gaussian pyramid. To accelerate this stage, we implement an FFT-based Gaussian convolution to accelerate the process of every single layer. Additionally, the Uber kernel programming model, heterogeneous parallelism, and the GPU-based resize function are adopted to further exploit the parallelisms of this stage.

4.1.2. The FFT-based Gaussian convolution

As a two-dimensional (2D) ($n \times n$) Gaussian kernel can be reduced to two consecutive one-dimensional (1D) Gaussian kernels, we used two 1D Gaussian convolutions to replace a 2D Gaussian direct convolution for an ($N \times M$) image. It is easier to realize coalesced global memory accesses and reduce redundant memory accesses for the separable 1D Gaussian convolution using shared

memory and vectorization methods. However, the sizes of Gaussian kernels used at this stage are large. The sizes are 11, 13, 17, 21, and 27. In this case, an FFT-based Gaussian convolution is introduced in large octaves and achieves higher performance than the separable one. The FFT-based Gaussian convolution is a frequency-domain-based algorithm, and the convolution operation between the image and the Gaussian kernel in the time domain is equivalent to their respective Hadamard product in the frequency domain after the FFT transformation.

Considering an FFT with a power-of-two length is faster than an FFT with the length of other prime radices; therefore, we allocate ($fftN \times fftM$) GPU memory for the input ($N \times M$) image, where $fftM$ is aligned with a power-of-two length. We used cudaMemcpy2D() to copy the input image to the global memory. Further, we need to expand the ($n \times n$) Gaussian kernel to ($fftN \times fftM$). We subsequently adopted a real-to-complex FFT (forward) in the expanded image and the Gaussian kernel, respectively. When the transformations are completed, we adopted the Hadamard product in their frequency domain data and obtained an intermediate matrix. A complex-to-real FFT (backward) is finally adopted in the intermediate matrix to yield the Gaussian convolution result, as shown in Fig. 2.

4.1.3. Uber kernel

Uber kernel [39] incorporates multiple different tasks into a single physical kernel, expressing task-level parallelism within one kernel without the overhead of switching between kernels. In the naïve implementation, the ith layer of the oth octave in the DoG pyramid is produced by the ith layer and the $(i+1)$th layer of the oth octave in the Gaussian pyramid. Packing the DoG pyramid kernel and the Gaussian convolution kernel into a single kernel on each octave can fully exploit task parallelism.

Another advantage of the Uber kernel is its ability to reduce global memory accesses, which is more fundamental to the overall performance than the reduction in kernel launchings. In the Uber kernel, threads no longer require re-reading the $(i+1)$th Gaussian layer. This is because as soon as the $(i+1)$th Gaussian layer is obtained, the result in registers (instead of in the global memory) will be reused to calculate the $(i+1)$th layer of the DoG pyramid. Benefiting from the Uber kernel, we reduced the global memory read of $O \times (S+2)$ images and $O \times (S+2)$ times kernel launchings. Therefore, the introduction of the Uber kernel reduces not only a large amount of global memory accesses but also the kernel launchings.

4.1.4. The heterogeneous parallelism

To utilize all the computing resources in a computing system, HartSift adopts a heterogeneous parallel approach to accelerate this stage. Here, CPUs and GPUs will operate in a parallel and cooperating manner to construct the DoG pyramid.

Compared with CPUs, GPUs contain few advantages in processing small images. It is a wise choice to construct large octaves on GPUs and small octaves on CPUs. According to the computing power of CPUs and GPUs used in this study, HartSift defines 256×256 as the separation point. Users can configure their own separation point to guarantee that workloads are balanced between CPUs and GPUs. As shown in Fig. 3, CPUs will process small octaves when a procedure arrives at the separation point, and GPUs continue to calculate the remaining layers of the current octave simultaneously.

HartSift adopts multiple CUDA streams to exploit parallelism. As CUDA operations in different streams run concurrently, HartSift breaks the order of irrelevant operations without losing any validity. The performance will be improved by overlapping GPU kernels with data transfers and CPU executions.

4.1.5. Resize function

The resize function is used in both upsampling and downsampling operations in the SIFT algorithm. For example, in the resize function of a linear interpolation, one pixel of the output image is based on a weighted sum of values of a small 2 x 2 matrix's pixels in the input image. While porting the resize function to the GPUs, we used adjacent threads in a warp to load continuous global memory data to shared memory in a coalesced manner, such that the cache line utilization will be improved. Considering pixels in the same row of the output image map to pixels of the same row in the input image, as well as the pixels in the same column, we subsequently vectorize each thread to accommodate more than one pixel and reduce the number of computations by reusing the typical intermediate results.

4.2. Accurate keypoint localization

4.2.1. Parallelism analysis

The number and locations of the keypoints are random, as they depend on the size and content of the input image. This phenomenon will lead to three serious problems when porting this stage to the GPUs: (a) Load imbalance. Detecting keypoint candidates will lead to load imbalance in two levels: interwarp load imbalance, and intrawarp load imbalance. (b) The accumulation of information of relatively small sample sets. The core operation of the orientation assignment is to construct histograms for the keypoints; therefore, a high-performance histogram algorithm will substantially improve the performance. (c) Low efficiency of the global memory access pattern. Because the locations of the keypoints are random, this stage contains irregular and redundant data accesses.

4.2.2. Optimization strategies

(1) Load balancing

A warp is the smallest executable unit of parallelism on NVIDIA GPUs and refers to a collection of 32 threads that are executed in a lockstep. Further, 32 threads of a warp execute the same instruction sequence on different data. If the workloads among the 32 threads are imbalanced, the execution time of the warp will depend on the last thread that completes its work. In addition, load imbalance among warps will decrease the utilization of multiprocessors within GPUs. Therefore, load balancing is critical to eliminate the performance bottleneck at this stage.

As shown in the upper part of Fig. 4, when porting this stage to the GPUs, it is natural to process these three parts of this stage using one single kernel. For a 1024×1024 image, we need to execute millions of threads to process all pixels in the DoG pyramid. As most pixels are not extrema, most threads that are mapped into these pixels will be idle after the keypoint candidates detection part. Based on the random distribution of the keypoint candidates, almost all warps contain a different number of idle threads. If these warps continue to process the following parts, the load imbalance of two levels will occur:

- Interwarp load imbalance. Because the number of active threads differs for different warps, different workloads will be dispatched to each warp.
- Intrawarp load imbalance. Different threads within the same warp contain different workloads.

In addition, during the removal of invalid keypoint candidates, the threads will perform a subpixel interpolation to obtain accurate information of the keypoint candidates such that keypoint candidates with low contrast or unstable edge response will be removed. In other words, some threads will exit earlier than others one more time. The load imbalance will become more severe.

To mitigate the performance bottleneck, HartSift introduces two optimizations: rebalancing workloads and multigranularity parallelism.

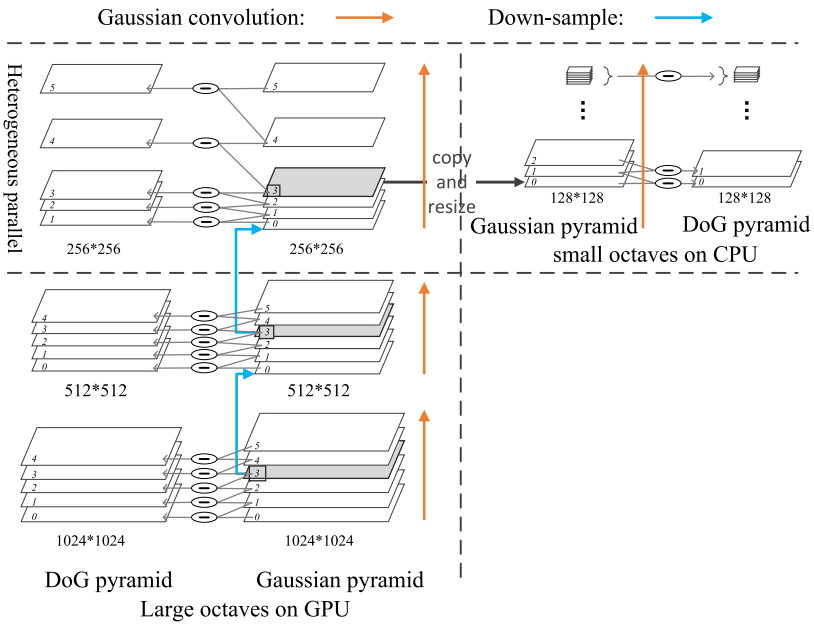

Fig. 3. The heterogeneous parallelism adopted in the pyramids construction.

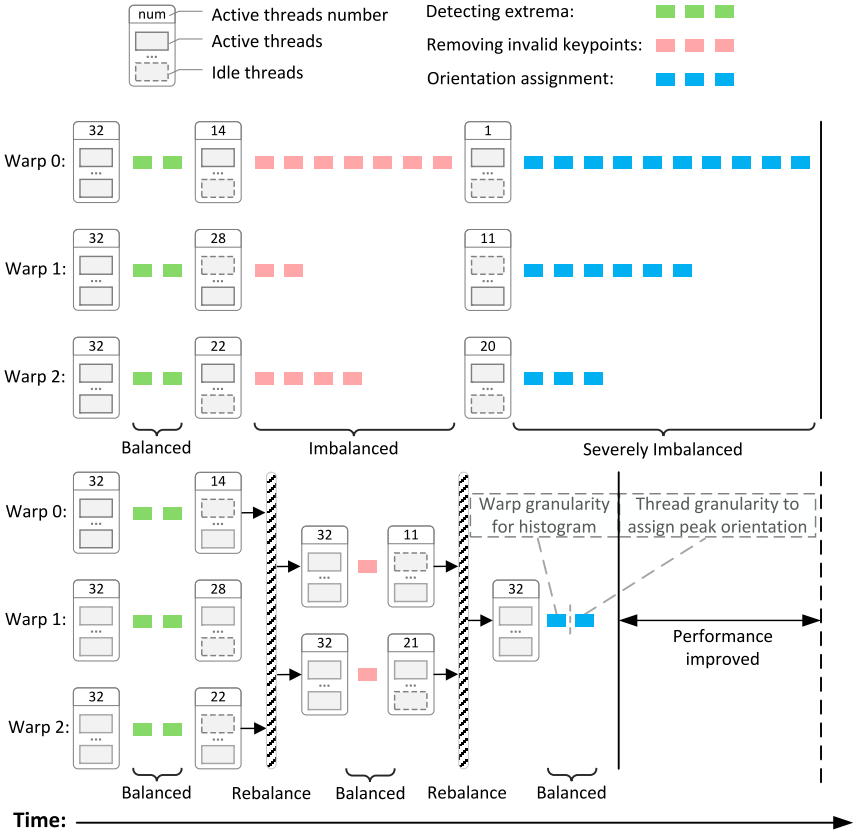

Fig. 4. The top of this figure is an imbalanced sample for the number of active threads, and the workloads of warps in the latter two parts are different. The bottom is a load-balancing sample using rebalancing workloads and multigranularity parallelism. Two new kernels with 32 active threads per warp will be launched to remove invalid keypoints and assign orientations after detecting the extrema, and after removing invalid keypoints, separately. Workloads are balanced in these three parts.

Rebalancing workloads. The upper part of Fig. 4 reveals the naïve implementation that only uses one kernel to detect the keypoint candidates, and removes the invalid keypoints and assigns orientations to the keypoints. The analysis above demonstrates that this method will lead to load imbalance. By rebalancing the workloads among the threads, the imbalanced execution path is shortened, as shown in the lower part of Fig. 4. To rebalance the workloads among the threads, HartSift defines three kernels with different granularities to process these three parts separately. When the load imbalance occurs, HartSift will rebalance the workloads by launching a new kernel with the appropriate granularity, and dispatch new warps with 32 active threads in each part.

In detecting the extrema, if a thread finds an extremum, it will subsequently use atomic operations to acquire the index of a temporary buffer. The thread will store the extremum's location information in the buffer; therefore, the extrema can be tracked by removing the invalid keypoint kernel. Similarly, if a thread of the removing invalid keypoint kernel is processing a valid keypoint, it will acquire an index of the keypoint structure array through atomics, and subsequently store the valid keypoint. Finally, the orientation assignment kernel updates each keypoint in the keypoint array with its own orientations.

We rebalance the workloads twice at this stage: after the keypoint candidates detection part, and after the invalid keypoints removal part. As shown in the lower part of Fig. 4, by rebalancing the workloads among the threads and warps, all threads and warps through this stage are fully loaded, and the workloads of the threads are almost balanced in every part. Furthermore, launching three kernels to process these three parts separately can not only rebalance the workloads for each part, but also provide the flexibility to choose the appropriate granularities according to the characteristics of different parts.

Multigranularity parallelism. Optimizing workload rebalance guarantees that the threads and warps in each kernel are completely loaded. Multigranularity parallelism optimization guarantees that workloads will be dispatched evenly to the threads and warps. Moreover, the selection of appropriate granularities for different kernels depends on the characteristics of the workloads. According to the corresponding characteristics of this stage, HartSift adopts thread granularity to the keypoint candidates detection part and the invalid keypoints removal part by mapping one thread to one or multiple pixels. However, because the neighbor-region radiuses of different keypoints are different in the orientation assignment part, they are determined by the parameter σ_{octave} of Eq. (3); therefore, the thread granularity will lead to load imbalance. Furthermore, thread granularity assigns excessive work to a single thread, thus limiting the utilization of hardware resources. Therefore, mix-granularity parallelism is applied in this part: adopting the warp granularity to construct histograms and thread granularity to search the dominant orientation and assigning it to the corresponding keypoint. After adopting the rebalancing workloads and multigranularity parallelism optimizations, load balancing is realized and the performance of this stage is substantially improved.

(2) The warp-based histogram algorithm

In the orientation assignment part, even though the distribution of keypoints is random and unpredictable, the neighbors of each keypoint in the region contain good data locality. Thus, we propose a high-performance histogram algorithm based on warp granularity and atomic operations for each keypoint to exploit neighbor locality. The cache line of the NVIDIA GPUs is 128 bytes; if 32 adjacent threads in a warp access 32 adjacent neighbors, then only one cache line is required to access 32 neighbors.

The naïve implementation uses one thread to construct a histogram for one keypoint, which contains two problems: (a) Load imbalance occurring among the threads. Because the neighbor-region radiuses of different keypoints are different, they are determined by σ_{octave} of Eq. (3); subsequently, each thread contains different workloads. (b) When threads within the same warp calculate the histograms for their own keypoints, adjacent threads read non-adjacent memory addresses to load the respective neighbors; therefore, the memory access pattern is non-coalesced.

Therefore, this stage uses one warp to construct a histogram for one keypoint. The warp granularity presents two primary advantages: (a) Eliminating load imbalance. Threads within a warp will handle almost the same amount of neighbors. (b) Realizing the coalesced global memory access pattern [12,35]. Threads within a warp will access neighbors in a coalesced manner, which will improve the efficiency of memory accesses.

Algorithm 1: The warp-based histogram algorithm.

Input: radius: the neighbor-region radius; kpts[]: the keypoints array;
img[]:the input layer.
Output: Hist[]: array for all keypoints' histogram in global memory.

```
1: lane ← threadIdx.x&31                          ▷ thread id within a warp, 0∼31
2: idx ← blockIdx.x × blockDim.x + threadIdx.x    ▷ the global thread id
3: g_wid ← idx >> 5                               ▷ the global warp id, a warp for one keypoint
4: len_div ← (radius × 2+1)^2 >> 5                ▷ iterations for 32 threads of a warp
5: pHist ← Hist+n × g_wid                         ▷ the histogram initial address of this keypoint
6: upperLeftPos(kpts[g_wid], radius, &x, &y)      ▷ calculating the initial address of neighbors' region of the keypoint
7: /*Note that the warp processes 32 neighbors in each iteration*/
8: for i ← 0 to len_div do
9:    calcPositon(&nx, &ny, x, y, lane, radius)   ▷ the (nx,ny) neighbor
10:   mag ← oriMagnitude(img, nx, ny)             ▷ the gradient magnitude
11:   w ← oriWeight(x, y, radius, lane)           ▷ the weights over neighbors
12:   ori ← oriBin(img, nx, ny)                   ▷ the bin for voting
13:   atomicAdd(pHist, ori, mag, w)               ▷ accumulate using the atomic operation
14:   lane ← lane+ 32                             ▷ traversing all the neighbors
15: end for
```

Because the neighbor-region radiuses are considerably small and only hundreds of neighbors occur per keypoint, as shown in Eq. (3), using atomic operations will not result in global memory pressure and performance reduction. However, the warp-based histogram algorithm based on atomic operations is not applicable in the descriptor generation stage, because the descriptor generation stage contains a much larger neighbor-region radius for each keypoint. Hence, an atomic-free histogram algorithm for large-scale neighbor samples will be introduced in the next stage.

The pseudo code of the warp-based histogram algorithm is presented in Alg. 1. This algorithm is employed in small-scale neighbor samples by adopting warp granularity and atomic operations. This stage always uses one warp to handle one keypoint. We configure 8 warps ($8 \times 32 = 256$ threads) per block in the kernel. During each iteration, 32 adjacent threads of a warp will load 32 adjacent neighbors in the coalesced way, and then calculate gradient orientations of neighbors and accumulate them in a 36-bin histogram. As threads within a warp likely contain the same bin to accumulate, atomicAdd() is used to guarantee the correctness. After traversing all the neighbors, each keypoint will obtain a primary orientation histogram.

(3) Efficient memory accesses

Eliminating redundant memory accesses and the SOA (Structure-Of-Arrays) memory layout are able to achieve efficient memory accesses.

Eliminating redundant memory accesses. In order to calculate the orientation of each keypoint, HartSift needs to calculate the orientation, gradient value, and weight of each keypoint's neighbors. The naïve implementation stores these results into a global memory buffer temporarily. These data will be read back from the buffer and accumulated to the histogram in the next step. Hence, this method produces many global memory accesses. After analyzing the dependency between the intermediate data calculation

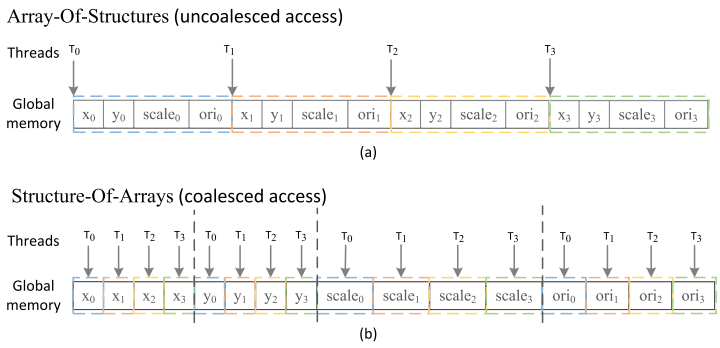

Fig. 5. This figure shows four threads read four extrema in two different manners. (a) The AOS pattern accesses extrema on global memory in a non-coalesced way. (b) The SOA pattern realizes coalesced global memory accesses.

and the histogram construction, temporary memory operations can be completely eliminated by merging the two steps into one. Once threads calculate the gradient, orientation, and weight of each neighbor, these results will be accumulated to the histogram without any temporary global memory access.

SOA memory layout. The structure of keypoint is composed of x coordinate, y coordinate, scale, and orientation. If HartSift manages keypoints by the AOS (Array-Of-Structures) pattern, as shown in Fig. 5(a), adjacent threads will access non-adjacent segments of adjacent keypoints, which is a non-coalesced access pattern. In order to achieve coalesced global memory accesses on GPUs, HartSift converts data structures from AOS to SOA representation, as shown in Fig. 5(b). In the SOA memory layout, warps are able to make full use of the 128-byte cache line and achieve coalesced memory accesses, which means 32 threads in a warp read adjacent segments simultaneously.

4.3. Generating descriptor

4.3.1. Parallelism analysis

This stage contains two challenges: (a) How to construct a 360-bin histogram with large-scale neighbor samples. (b) How to avoid memory address collisions among threads when doing the trilinear interpolation, since each image gradient votes for eight histogram bins.

For the first challenge, the neighbor-region radiuses of keypoints at this stage are much larger than the previous stage according to Eqs. (3) and (4). We need to accumulate thousands of neighbors' information in a 360-bin histogram for each keypoint. From analyses mentioned in Section 4.2.2, it is undoubtedly better to use warp granularity to construct the histogram for each keypoint. However, if we directly adopt the warp-based histogram algorithm with atomic operations as the previous stage, it will lead to different influences on the performance: (a) Warp granularity inherits its merits. Load balancing and coalesced global memory accesses are still able to accelerate the histogram generation but less prominent. (b) Many more neighbors produce more atomic operations on global memory, resulting in operation serialization and performance degradation.

For the second challenge, the accumulation operation at this stage is substantially different from the previous stage. In the previous stage, each image gradient only votes for one histogram bin. However, this stage adopts the trilinear interpolation to distribute gradients in the 360-bin histogram, meaning each gradient votes for 8 bins. In warp granularity, we use one warp to generate one 128D histogram for one keypoint. Then, 32 threads in a warp process 32 neighbors in one iteration. Because each gradient votes for eight bins, some of the 32 threads have a high probability of voting for the same 8 bins. If we adopt the warp-based histogram algorithm in this situation, the serialization of threads will be more serious. In an extreme case, 32 threads vote for the same 8 bins, then every 32 atomic operations on each bin will be serialized.

Algorithm 2: The atomic-free histogram algorithm.

Input: radius: the neighbor-region radius; kpts[]: keypoints; img[]:the input layer; hist[360×_NUM]: array in shared memory for _N warps per block.
Output: Hist[]: the 128D descriptor.

```
1:  lane ← threadIdx.x&31                          ▷ thread id in a warp, 0∼31
2:  upperLeftPos(kpts[idx>>5], radius, &x, &y)     ▷ the region's initial address
3:  b_wid ← threadIdx.x/32                         ▷ warp id in a block, _N warps/block
4:  phist ← hist+360×b_wid                         ▷ the 360-bin histogram of this keypoint
5:  len_div ← (radius×2+1)^2>>5                    ▷ iterations for 32 threads of a warp
6:  /*Note that the warp processes 32 neighbors in each iteration*/
7:  for i ← 0 to len_div do
8:    calcPositon(&nx, &ny, x, y, lane, radius)    ▷ the (nx,ny) neighbor
9:    ori ← descOrientation(img, nx, ny)           ▷ orientation
10:   pos ← calcIndex(nx, ny, lane, radius, ori)   ▷ the key for reduction, index of the nearest subregion's bin of this neighbor
11:   mag ← descMagnitude(img, nx, ny)             ▷ the gradient magnitude
12:   w ← descWeight(x, y, radius, lane)           ▷ the weight over the neighbors
13:   tri[] ← trilinearInterpolation(ori, mag, w)  ▷ eight gradients stored in registers, not memory. (tri[] is only used for convenience)
14:   atomicFree_multiKey_reduction(phist, tri, pos, lane)  ▷ votes for eight bins
15:   lane ← lane+ 32                              ▷ traversing all the neighbors
16: end for
17: transformHist2Desc(shist, Hist)                ▷ transform 360-bin shist to 128D Hist
```

To address these two challenges, we introduce the atomic-free histogram algorithm based on the warp-based histogram algorithm. The atomic-free histogram realizes an atomic-free multikey reduction using warp-level parallelism, to vote for eight bins without any atomic operations, synchronizations, and memory conflict.

4.3.2. Optimization strategies

(1) The atomic-free histogram algorithm The framework of the atomic-free histogram algorithm, as listed in Alg. 2, is similar to the warp-based histogram algorithm. First, we need to dispatch one keypoint to one warp and obtain the initial address of the neighbor region. Because 32 adjacent threads in a warp will process 32 adjacent neighbors, subsequently the warp can accumulate the 32 neighbors' information in the coalesced global memory access

lane	pos	peers			captain
		interation 1	iteration 2	iteration 3	
0	a	00101101	00101101	00101101	TRUE
1	b		10010010	10010010	TRUE
2	a	00101101	00101101	00101101	
3	a	00101101	00101101	00101101	
4	b		10010010	10010010	
5	a	00101101	00101101	00101101	
6	c			01000000	TRUE
7	b		10010010	10010010	

Fig. 6. Grouping threads with the same *pos* value. The 1st group is composed of thread 0/2/3/5; the 2nd group is composed of thread 1/4/7, and the 3rd group only contains thread 6. A group's captain is the thread with the smallest lane.

lane	pos	peers	captain	value		
				initial	iteration 1	iteration 2
0	a	00101101	TRUE	7	9	23
1	b	10010010	TRUE	1	7	15
2	a	00101101		2		
3	a	00101101		9	14	
4	b	10010010		6		
5	a	00101101		5		
6	c	01000000	TRUE	11	11	11
7	b	10010010		8	8	

Fig. 7. This figure shows a warp-level reduction with three keys. Values with the same key are accumulated in parallel in each iteration. Finally, each captain (thread 0/1/6) holds the sum of its group.

pattern in each iteration. After the warp traverses all the neighbors of the keypoint, the 360-bin histogram of the keypoint will be generated and further transformed into the 128D descriptor.

The difference between the atomic-free histogram algorithm and the warp-based histogram algorithm is how the gradients are accumulated in the histograms. The former adopts the trilinear interpolation and accumulates gradients in a 360-bin histogram. In line 14 of Alg. 2, the trilinearInterpolation() will return eight values that are used to update eight histogram bins. These eight values are stored in registers instead of memory. We use tri[] to replace eight temporary variables for convenience in the pseudo code. The gradients are not accumulated in the histogram yet in line 14, and the accumulations are completed in the atomicFree_multiKey_reduction(). The core operation of the atomic-free histogram algorithm is the atomic-free multikey reduction algorithm, which can complete the votes of each gradient for eight histogram bins rapidly. It is noteworthy that the hist[] array in Alg. 2 resides in shared memory. Because threads may have different and non-adjacent bins to update, it shows a non-coalesced memory access pattern. In this case, shared memory is a better choice than global memory. As we need to pre-allocate shared memory for each block, we recommend 2∼4 warps per block. If we configure four warps per block, then each block contains $4 \times 32 = 128$ threads. The details of the atomic-free multikey reduction algorithm are described in Alg. 3.

(2) The atomic-free multikey reduction algorithm

In each iteration in Alg. 2, 32 neighbors produce 32 *pos* variables. Typically, some of the 32 *pos* variables contain the same value. Considering that the 32 neighbors contribute to the same 360-bin histogram, how each gradient votes for eight histogram bins can be attributed to the multiple keys reduction problem. For gradients combined by key (*pos* variable) with different values, pre-combining gradients with the same key at the warp level leads to a significant acceleration.

The atomic-free multikey reduction algorithm, listed in Alg. 3, has three practical advantages. First, threads in a warp efficiently communicate with each other using shuffle/vote intrinsic instructions. This communication is significantly faster than communications through shared/global memory and necessary synchronizations. Second, because warp is the smallest parallel unit and

Algorithm 3: The atomic-free multikey reduction algorithm.

Input: pos: index of the nearest subregion's bin of this neighbor; lane: thread id within warps, 0∼31; tri[]: the weighted eight gradients.

Output: phist[]: the 360-bin histogram of the warp in shared memory.

```
1: peers ← 0; available ← 0xffffffff; ▷ 32 bits represent 32 threads in a warp
2: while is_peer≠0 do
3:     peers ← __ballot((pos = __shfl(pos, __ffs(available) - 1))) ▷ if threads have the same pos value (matching), their peers variables are equal
4:     available ←available∧peers ▷ use xor mask to remove matched threads
5: end while
6: captain ← (__ffs(available)-1=lane) ▷ captain:is the smallest id thread?
7: relPos ← __popc(peers<<(32-lane)) ▷ lane's relative position
8: peers ← peers&(0xfffffffe<<lane) ▷ clear bits up to this lane
9: /*32 threads may have different pos variables. The captains use warp-level instruction to accumulate other threads' tri[] with the same pos*/
10: while __any(peers) do
11:     next ← __ffs(peers) ▷ search which thread with the same pos to add
12:     _tri[] ← __shfl(tri[], next-1) ▷ Be careful: the __shfl() only obtain one variable. We use _tri[] and tri[] to replace 8 __shfl() just for convenience.
13:     if next then
14:         tri[] ← tri[]+_tri[] ▷ Be careful: use _tri[] and tri[] for convenience.
15:     end if
16:     peers ← peers&__ballot((rel_pos&1)≠ 0) ▷ clear peers that were just used
17:     rel_pos ← __ffs(peers) ▷ shift position to control iteration
18: end while
19: if captain = TRUE then
20:     voteForBins(phist, tri, pos) ▷ each captain updates 8 bins, no conflict here
21: end if
```

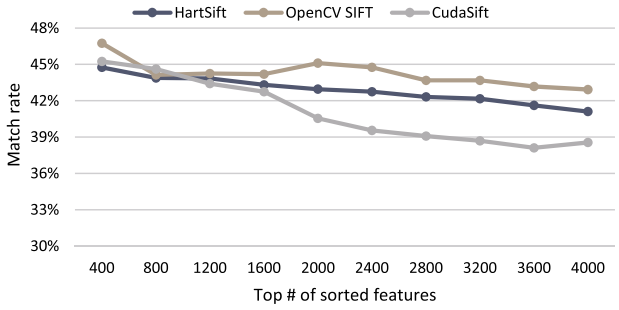

Fig. 8. The matching rates of HartSift, OpenCV SIFT and CudaSift. The abscissa axis is the top number of sorted features.

threads in a warp work simultaneously, all synchronizations or atomic operations will be completely eliminated. The final advantage is that the algorithm addresses memory address collisions among threads.

The atomic-free multikey reduction algorithm is composed of two steps. The first step is to find out which threads (peers) share the same key within each warp. In our case, the key is the *pos* variable and the values are gradients. Lines 1∼6 in Alg. 3 illustrate

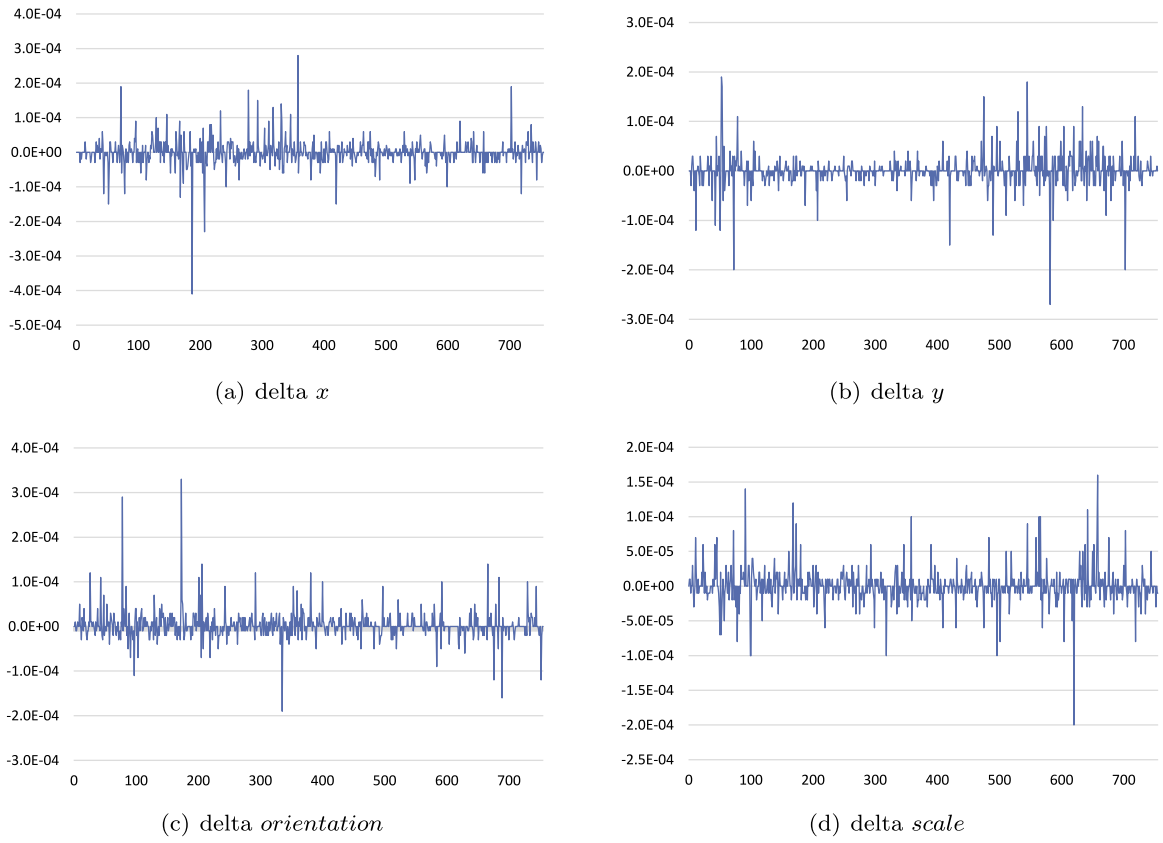

Fig. 9. Comparisons of four components of keypoints between HartSift and OpenCV SIFT. The abscissa axis is the number of keypoints. (a) The difference of x-coordinate, (b) The difference of y-coordinate, (c) The difference of orientation, and (d) The difference of scale.

the process of the first step. For simplicity, Fig. 6 assumes each group contains 8 threads (instead of 32). In the first iteration, because threads 0/2/3/5 contain the same *pos*, these threads belong to the same group and acquire the same *peers*. In this example, there are three groups after we finish lines 1∼5 in Alg. 3. Line 6 will appoint the thread with the smallest lane as the captain of the group. The next step is to accumulate all peers' eight gradients according to the bit-pattern *peers*, and the captain of each group finally votes for eight bins with the sum of peers' gradients. Fig. 7 gives a simplified example of this processing. For simplicity, Fig. 7 only accumulate one integer variable. In a real situation, this step needs to accumulate eight gradients (line 14 in Alg. 3). As shown in Fig. 7, threads use parallel tree-like reductions to add up gradients according to the bit-pattern *peers* variable. The *peers* will deactivate the used threads with higher lanes in each iteration. Finally, only captains hold the eight sums of every eight bins of their groups and have the right to vote for eight histogram bins. Because the captains are able to accumulate their eight gradients into different sets of eight bins without conflicts, the problem of serializing parallel threads is completely addressed.

5. Experiments

Considering OpenCV SIFT, SiftGPU, and CudaSift are presently the most popular open-source SIFT implementations, this section will evaluate the reliability and performance of HartSift through comparisons with these other implementations. We will double the input image in the DoG pyramid construction stage and use "-O3" in compile phase in all experiments. Table 1 presents the experimental environment.

Table 1
Experimental environment.

CPU	Intel Xeon E5-2670 v3	2.3 GHz	–
GPU	TITAN X (Pascal)	1.4 GHz	11 TFlops
Library	OpenCV 3.3.0 Release Version with IPP, TBB, SSE, AVX; CudaSift; SiftGPU-V400's CUDA version; CUDA 8.0.		

5.1. Stability comparison

Different application scenarios contain different requirements on the number and quality of features. Some applications benefit from a few high-accuracy features, while other applications require as many features as possible. Considering more distinct features with higher DoG responses tend to be of higher quality, we perform a lateral comparison of the matching rate among HartSift, OpenCV SIFT, and CudaSift. The matching rate is the proportion of good matches under the condition of generating the same number of features. In this experiment, we use three SIFT implementations with their own recommended settings to generate features. The two 1280 × 960 matching images are from CudaSift's dataset [4]. These features are sorted in descending order by their corresponding DoG responses.

Fig. 8 displays the matching rates of HartSift, OpenCV SIFT, and CudaSift. Because features with higher responses in the differential of Gaussian are of higher quality and easier to be matched, OpenCV SIFT has the highest matching rate, and HartSift's matching rate is close to OpenCV SIFT. CudaSift is more sensitive to changes of the DoG response than HartSift and OpenCV SIFT. Furthermore, as the top number of sorted features increases, CudaSift's matching rate decreases faster than HartSift and OpenCV SIFT. Therefore, if applications only depend on a small number of high-accuracy

Fig. 10. (a) OpenCV SIFT vs HartSift. The image shows the keypoints matching between OpenCV SIFT (white solid signs) and HartSift (green hollow signs). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

features, then all three implementations meet the requirements. However, HartSift and OpenCV SIFT are the better choices for applications that require a large number of features, and HartSift is much faster than OpenCV SIFT.

5.2. Similarity between HartSift and OpenCV SIFT

HartSift's features are similar to OpenCV SIFT's. We compare the differences of the x coordinate, y coordinate, gradient orientation, and scale coordinate of keypoints produced by HartSift and OpenCV SIFT (except for a few outliers). As shown in Figs. 9(a) and 9(b), the differences of x are less than 0.0005 and y's are less than 0.0004. The x and y coordinates are accurate for the resolutions of x coordinate and y coordinate are 1 (one pixel). As HartSift adopts a 36-bin orientation histogram, the resolution of gradient orientation is $2\pi/36$. Because the differences of the orientation shown in Fig. 9(c) are less than 0.0004, which is much smaller than the resolution, the orientation is also accurate. In addition, Fig. 9(d) shows that the differences of scale are less than 0.0003.

Fig. 10 indicates the matching of the keypoints between HartSift (marked by the green hollow signs) and OpenCV SIFT (marked by the white solid signs). If a keypoint of HartSift contains a matching keypoint of OpenCV SIFT, then the green sign will perfectly overlap the white sign. Fig. 10 suggests that the matching rate is very high, and less than 0.01% outliers exist.

From the comparisons of keypoint structure's components in Fig. 9 and the matching of keypoints in Fig. 10, the features extracted by HartSift are highly similar to OpenCV SIFT. Because OpenCV library is widely used (but does not contain a GPU-based SIFT implementation), we suggest users to simply substitute HartSift for OpenCV SIFT to accelerate applications.

5.3. Overall performance comparisons

Fig. 11 suggests that the overall performance of HartSift outperforms OpenCV SIFT's, SiftGPU's, and CudaSift's. HartSift processes different-sized images within 3.07∼7.71 ms (129.7∼325.73 fps). OpenCV SIFT costs 187.26∼771.71 ms (1.30∼5.34 fps), SiftGPU costs 20.63∼39.87 ms (25.08∼48.47 fps), and CudaSift costs 4.19∼11.21 ms (89.21∼238.66 fps). Fig. 12 shows HartSift is 55.88∼121.99 times, 5.17∼6.88 times, and 1.25∼1.79 times faster than OpenCV SIFT, SiftGPU, and CudaSift respectively.

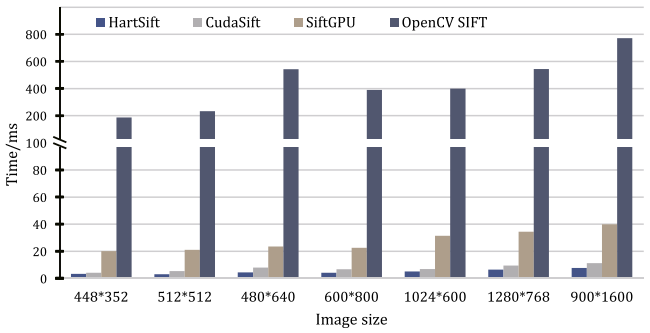

Fig. 11. Performance comparisons among HartSift, CudaSift, SiftGPU, and OpenCV SIFT.

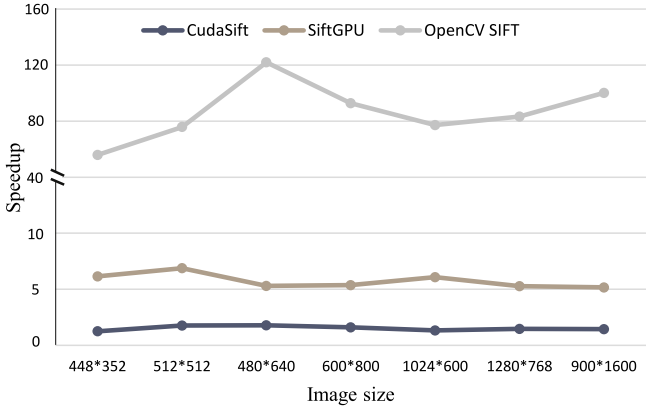

Fig. 12. HartSift's speedups by comparing with CudaSift, SiftGPU, and OpenCV SIFT.

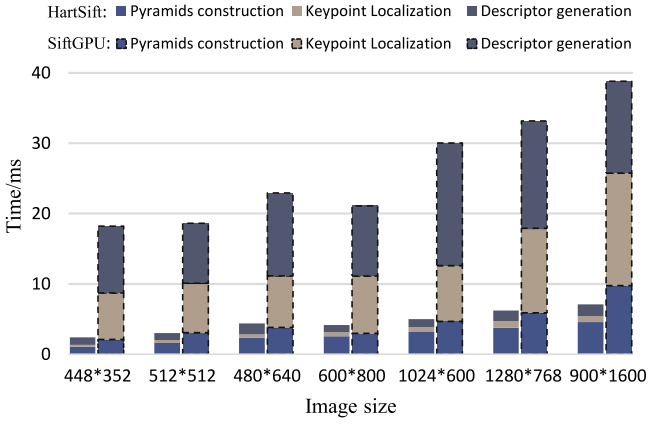

Fig. 13. Performance of different stages between HartSift and SiftGPU.

5.4. Performance tendency analysis of each stage

Fig. 13 shows the performance of each stage between HartSift and SiftGPU:

(a) The three stages of HartSift outperform the corresponding stages of SiftGPU.

(b) Each stage contains its own key factors on the performance. The cost of the pyramids construction stage will increase with the growth of the size of the input image. The accurate keypoint localization stage is related to both the size and the content of the image. For the descriptor generation stage, it only depends on the content of the input image.

(c) The performance bottleneck of SiftGPU is generated from the accurate keypoint localization stage and the descriptor generation

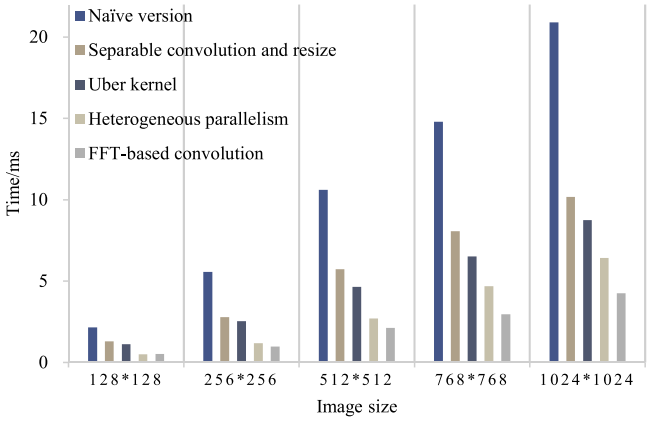

Fig. 14. Optimizations in the pyramids construction stage.

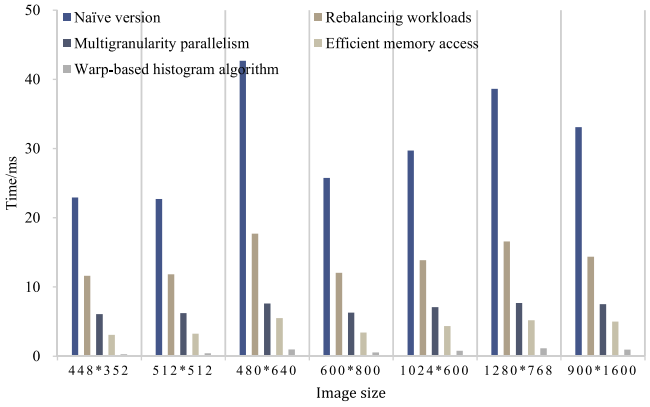

Fig. 15. Optimizations in the accurate keypoint localization stage.

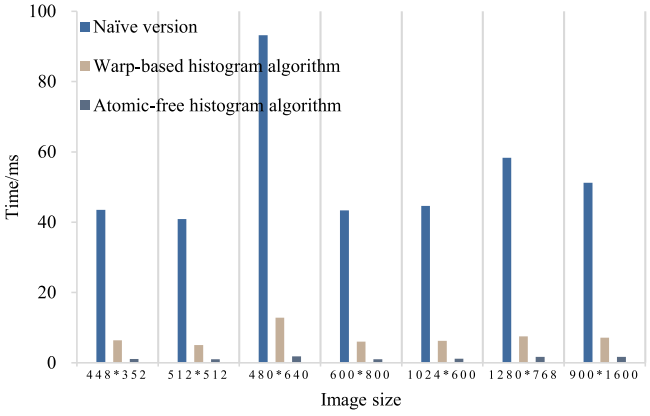

Fig. 16. Optimizations in the descriptor generation stage.

stage. After optimizing the latter two stages in HartSift, the performance bottleneck will be eliminated.

5.5. Optimizations on the pyramids construction stage

HartSift adopts several optimizations on the naïve implementation of the DoG pyramid construction, which is based on an optimized 2D Gaussian convolution. Fig. 14 records the improved performance of each optimization:

(a) HartSift initially uses the 1D separable Gaussian convolution with lower computation complexity to replace the 2D Gaussian convolution. In addition, the GPU-based resize function is introduced.

(b) Uber kernel packs the DoG pyramid kernel and the Gaussian pyramid kernel into one single kernel. By adopting Uber kernel, HartSift saves a significant amount of global memory accesses and reduces the number of kernel launchings.

(c) HartSift takes full advantage of heterogeneous computing resources by overlapping GPU kernels with data transfers and CPU programs. This strategy keeps CPUs and GPUs as busy as possible.

(d) The FFT-based convolution further accelerates the construction of the DoG pyramid, especially in large images.

5.6. Optimizations on the accurate keypoint localization stage

The naïve version only launches a thread-granularity kernel to process the whole stage. To achieve load balancing, HartSift introduces rebalancing workloads and multigranularity parallelism optimizations to provide active warps and multiple granularities. Furthermore, both optimizations lead to an efficient memory access pattern. As the locations and number of keypoints are random, HartSift adopts the warp-based histogram algorithm to efficiently accumulate small-scale neighbors. As Fig. 15 shows:

(a) Load imbalance of SIFT is a substantial detractor from the performance of GPUs. As illustrated in Section 4.2, kernels with inactive warps will result in performance degradation. Therefore, HartSift rebalances workloads to ensure each kernel is fully loaded. In addition, this optimization provides flexibility for each kernel to choose the most appropriate granularity and process their corresponding workloads.

(b) Benefiting from the flexibility provided by the rebalancing workloads optimization, at this stage HartSift is able to adopt different granularities in different kernels. Based on the characteristics of these three parts of this stage, HartSift adopts thread granularity to the keypoint candidates detection and invalid keypoints removal parts. For the orientation assignment part, HartSift launches a mix-granularity kernel. The rebalancing workloads and multigranularity parallelism optimizations address load imbalance in the entire stage. Therefore, the performance is significantly improved.

(c) Two optimizations are introduced to achieve efficient memory accesses: eliminating redundant memory accesses, and adopting the SOA memory layout. These improve performance by removing dependency among each part and taking advantage of the cache line.

(d) In the warp-based histogram algorithm, we adopt the warp-level parallelism to accumulate information of neighbors to the histogram for every keypoint. HartSift then launches a thread-level parallelism to search the dominant orientation. Mapping one warp to one keypoint's neighbors not only eliminates load imbalance among threads and all synchronizations, but also makes full use of neighbor locality.

5.7. Optimizations on the descriptor generation stage

The core operation of this stage is to construct a high-performance histogram for a much larger scale of neighbors of every keypoint than the previous stage's. As Fig. 16 demonstrates, an atomic-free histogram algorithm is much faster than the atomic one with large-scale sample sets:

(a) Benefiting from the warp-based histogram algorithm introduced in the accurate keypoint localization stage, warp granularity is still able to improve the performance of this stage. However, numerous atomic operations result in memory address collisions, serializing threads that can execute in parallel.

(b) To eliminate all atomic operations, the atomic-free histogram algorithm adopts the warp-level shuffle/vote instructions

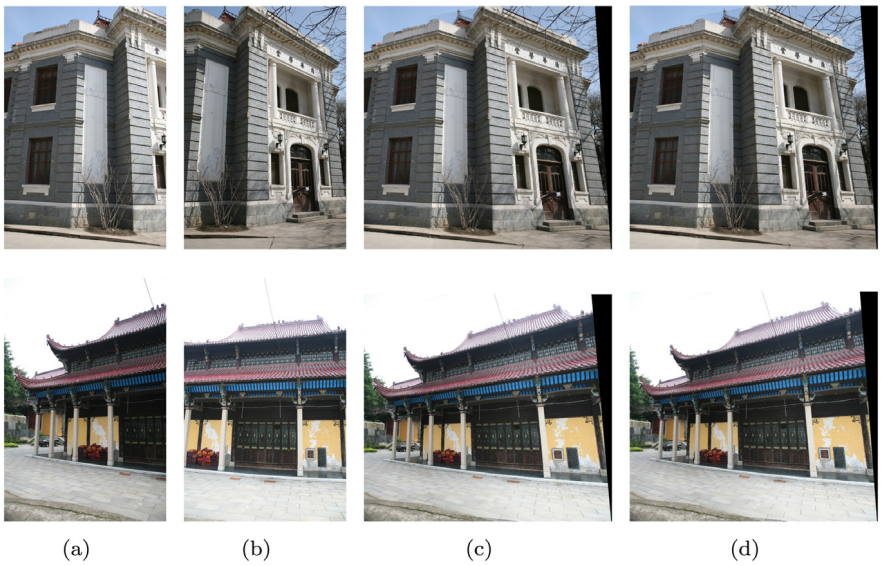

Fig. 17. (a) The first input image. (b) The second input image. (c) The stitching result of OpenCV SIFT. (d) The stitching result of HartSift.

instead of using shared memory or global memory to communicate among threads. Hence, HartSift further improves performance.

5.8. Application to image stitching

Image stitching is an important computer vision problem, which obtains a panoramic image from several related images. It is widely utilized in panoramic cameras, medical image processing, and other applications. Various stitching methods have emerged, such as the feature-based methods [13,25,33], the appearance-based methods [9,10,42] and the learning-based methods [7,8,24]. Their common aim is vital for achieving image stitching, which is to estimate the homography matrix between two images. The feature-based methods utilize a variety of feature detectors to calculate the homography matrix by minimizing the re-projection error. Thus, the feature-based methods are simple and time efficient. However, they are inefficient in non-texture areas. The appearance-based methods directly estimate the homography matrix by optimizing the photometric error. The appearance-based methods perform better in non-texture areas, but are very time consuming. The learning-based methods use machine-learning models to generate the homography matrix, such as convolution neural networks. These methods require a considerable number of labeled datasets and require further study. Considering the convenience and efficiency of stitching, the feature-based methods are recommended.

In this application, we extract SIFT features from two related images, and match them to acquire their corresponding keypoints. We then adopt the normalized Direct Linear Transform (DLT) algorithm [19] to estimate the homography matrix $H_{matrix} \in \mathbb{R}^{3\times3}$. In addition, we warp the second image to stitch the first image according to Eq. (5); Every point $p_2 = [x_2, y_2, 1]^T$ in the second image contains a corresponding position $p_1{}' = [x_1', y_1', 1]^T$ in the first image. Finally, the first image merges with the second warped image. Thus, we obtain the panoramic image.

$$p_1{}' = H_{matrix} \cdot p_2 \tag{5}$$

We used 3D reconstruction datasets [22] to evaluate the stitching effect of HartSift and OpenCV SIFT. In the image-stitching procedure, we used HartSift and OpenCV SIFT to extract their features; the results are shown in Fig. 17. The results indicate that both HartSift and OpenCV SIFT can achieve image-stitching. To quantitatively analyze HartSift and OpenCV SIFT, we used the mean pixel position bias (MPPB) to compare the difference in the position $p_1{}'$ acquired by HartSift and OpenCV SIFT. We estimated the homography matrix H_{hart} by HartSift, and H_{opencv} by OpenCV SIFT, and calculated the corresponding positions $p_{1i}{}'$ and $q_{1i}{}'$ depending on Eq. (5). We finally obtained the MPPB as the following:

$$MPPB = \frac{1}{N}\sum_{i=1}^{N} \|p_{1i}{}' - q_{1i}{}'\|_2 \tag{6}$$

where N denotes the number of pixels in the second image, $p_{1i}{}' = H_{hart} \cdot p_2$ and $q_{1i}{}' = H_{opencv} \cdot q_2$.

Fig. 18 shows the MPPBs between HartSift and OpenCV SIFT. The average pixel bias is 1.7, which is depicted by a broken red line. The biases are all lower than five pixels and most of them are lower than two pixels, which are much less than the size of the input image. In addition, Table 2 offers the number of good matches and matching rates of OpenCV SIFT and HartSift. In the image-stitching application, the matching rates of HartSift are equal to those of OpenCV SIFT. Because the two images of the first stitching have less co-vision, the numbers of good matches and matching rates are lower than the those of the second stitching. Even though the input images are different, the matching effect is stable. Hence, HartSift can perform equally to OpenCV SIFT.

6. Conclusions

We herein proposed a parallel SIFT on GPUs, named HartSift. It used CPUs and GPUs within a single machine to achieve a higher performance than other open-source implementations. HartSift boosted the performance by adopting different parallel strategies based on different characteristics of each stage, such as the FFT-based Gaussian convolution, rebalancing workloads, multigranularity parallelism, warp-based histogram algorithm, atomic-free histogram algorithm, and other optimizations. HartSift extracted the features of images within 3.07∼7.71 ms (129.7∼325.73 fps), thus satisfying the demanding requirements of real time processing. In addition, compared with OpenCV SIFT, SiftGPU, and CudaSift, HartSift acquired 55.88∼121.99 times, 5.17∼6.88 times, and 1.25∼1.79 times acceleration, respectively. In the future, we plan to develop a high-performance feature extraction library that includes other feature extraction algorithms.

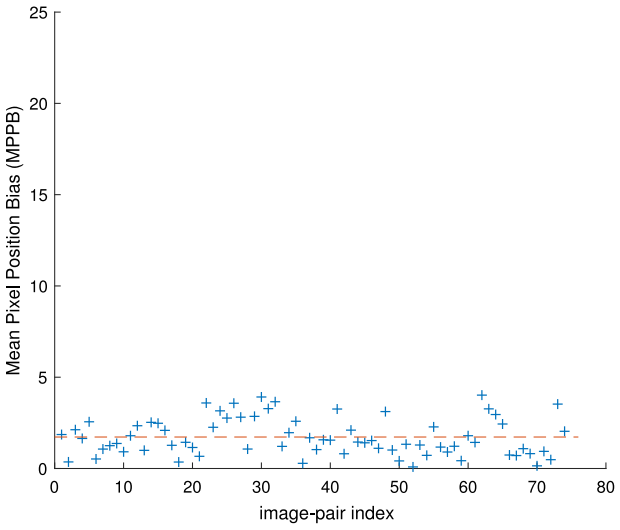

Fig. 18. MPPBs between HartSift and OpenCV SIFT on 77 pairs of images. This quantitative result shows the average bias is 1.7 (red broken line) and all biases are lower than 5 pixels, which can be neglected for the image size is much larger.

Table 2
The good matches and matching rate of the stitching application.

	The 1st stitching		The 2nd stitching	
	OpenCV SIFT	HartSift	OpenCV SIFT	HartSift
Good matches	1138	988	1605	1391
Matching rate	37%	35%	43%	42%

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant Nos. 61602443, 61432018, 61521092 and 61502450; the National Key Research and Development Program of China under Grant Nos. 2107YFB0202105, 2016YFE0100300 and 2017YFB0202302; the Key Technology Research and Development Programs of Guangdong Province under Grant No. 2015B010108006.

References

[1] K. Acharya, R. Babu, Speeding up SIFT using GPU, in: Computer Vision, Pattern Recognition, Image Processing and Graphics (NCVPRIPG), 2013 Fourth National Conference on, IEEE, 2013, pp. 1–4.
[2] K. Acharya, R. Babu, S.S. Vadhiyar, A real-time implementation of sift using gpu, J. Real-time Image Process. (2014) 1–11.
[3] H. Bay, T. Tuytelaars, L. Van Gool, Surf: speeded up robust features, in: European Conference on Computer Vision, Springer, 2006, pp. 404–417.
[4] M. Björkman, CudaSift: The fourth version of a sift implementation using cuda for gpus, 2017. https://github.com/Celebrandil/CudaSift.
[5] M. Björkman, N. Bergstrom, D. Kragic, Detecting, segmenting and tracking unknown objects using multi-label mrf inference, Comput. Vis. Image Underst. 118 (2014) 111–127.
[6] G. Bradski, The Open Computer Vision Library v3.3.0, 2017. http://opencv.org/opencv-3-3.html.
[7] A. Byravan, D. Fox, Se3-nets: learning rigid body motion using deep neural networks, in: Robotics and Automation (ICRA), 2017 IEEE International Conference on, IEEE, 2017, pp. 173–180.
[8] D. DeTone, T. Malisiewicz, A. Rabinovich, Deep image homography estimation, arXiv preprint arXiv:1606.03798.
[9] J. Engel, T. Schöps, D. Cremers, LSD-SLAM: large-scale direct monocular slam, in: European Conference on Computer Vision, Springer, 2014, pp. 834–849.
[10] J. Engel, J. Sturm, D. Cremers, Semi-dense visual odometry for a monocular camera, in: Proceedings of the IEEE international conference on computer vision, 2013, pp. 1449–1456.
[11] H. Fassold, J. Rosner, A real-time gpu implementation of the sift algorithm for large-scale video analysis tasks, in: Real-Time Image and Video Processing 2015, vol. 9400, International Society for Optics and Photonics, 2015, p. 940007.
[12] N. Fauzia, L.-N. Pouchet, P. Sadayappan, Characterizing and enhancing global memory data coalescing on gpus, in: Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, IEEE Computer Society, 2015, pp. 12–22.
[13] C. Forster, M. Pizzoli, D. Scaramuzza, SVO: fast semi-direct monocular visual odometry, in: Robotics and Automation (ICRA), 2014 IEEE International Conference on, IEEE, 2014, pp. 15–22.
[14] W.T. Freeman, E.H. Adelson, The design and use of steerable filters, IEEE Trans. Pattern Anal. Mach. Intell. 13 (9) (1991) 891–906.
[15] R. Girshick, Fast r-cnn, in: Proceedings of the IEEE International Conference on Computer Vision, 2015, pp. 1440–1448.
[16] R. Girshick, J. Donahue, T. Darrell, J. Malik, Rich feature hierarchies for accurate object detection and semantic segmentation, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2014, pp. 580–587.
[17] C. Harris, Geometry from visual motion, in: Active Vision, MIT press, 1993, pp. 263–284.
[18] C. Harris, M. Stephens, A combined corner and edge detector, in: Alvey Vision Conference, vol. 15, Citeseer, 1988, p. 50.
[19] R. Hartley, A. Zisserman, Multiple View Geometry in Computer Vision, Cambridge university press, 2003.
[20] K. He, X. Zhang, S. Ren, J. Sun, Spatial pyramid pooling in deep convolutional networks for visual recognition, in: European Conference on Computer Vision, Springer, 2014, pp. 346–361.
[21] S. Heymann, B. Fröhlich, et al., SIFT implementation and optimization for general-purpose gpu, in: The 15th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, Citeseer, 2007, pp. 317–322.
[22] Institute of Automation, Chinese Academy of Sciences, Image dataset- s for 3d reconstruction, 2011. http://vision.ia.ac.cn/zh/data/index.html.
[23] Y. Ke, R. Sukthankar, PCA-SIFT: A more distinctive representation for local image descriptors, in: Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on, vol. 2, IEEE, 2004, pp. II–506.
[24] A. Kendall, M. Grimes, R. Cipolla, Posenet: a convolutional network for real-time 6-dof camera relocalization, in: Proceedings of the IEEE international conference on computer vision, 2015, pp. 2938–2946.
[25] G. Klein, D. Murray, Parallel tracking and mapping for small ar workspaces, in: Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on, IEEE, 2007, pp. 225–234.
[26] Z. Li, H. Jia, Y. Zhang, HartSift: a high-accuracy and real-time sift based on gpu, in: Parallel and Distributed Systems (ICPADS), 2017 IEEE 23rd International Conference on, IEEE, 2017, pp. 135–142.
[27] T. Lindeberg, Feature detection with automatic scale selection, Intl. J. Comput. Vis. 30 (2) (1998) 79–116.
[28] D.G. Lowe, Object recognition from local scale-invariant features, in: Computer Vision, 1999. The proceedings of the Seventh IEEE International Conference On, vol. 2, IEEE, 1999, pp. 1150–1157.
[29] D.G. Lowe, Distinctive image features from scale-invariant keypoints, Intl. J. Comput. Vis. 60 (2) (2004) 91–110.
[30] K. Mikolajczyk, C. Schmid, An affine invariant interest point detector, in: Computer VisionECCV 2002, Springer, 2002, pp. 128–142.
[31] H.P. Moravec, Rover visual obstacle avoidance, in: Proceedings of the 7th International Joint Conference on Artificial Intelligence, vol. 2, Morgan Kaufmann Publishers Inc., 1981, pp. 785–790.
[32] J.-M. Morel, G. Yu, ASIFT: A new framework for fully affine invariant image comparison, SIAM J. Imaging Sci. 2 (2) (2009) 438–469.
[33] R. Mur-Artal, J.M.M. Montiel, J.D. Tardos, ORB-SLAM: a versatile and accurate monocular slam system, IEEE Trans. Robot. 31 (5) (2015) 1147–1163.
[34] NVIDIA, CUDA C Programming Guide v9.2, 2018. http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf.
[35] NVIDIA, CUDA C Best Practices Guide v9.1, 2018. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.
[36] S. Ren, K. He, R. Girshick, J. Sun, Faster r-cnn: towards real-time object detection with region proposal networks, IEEE Trans. Pattern Anal. Mach. Intell. 39 (6) (2017) 1137–1149.
[37] E. Rublee, V. Rabaud, K. Konolige, G. Bradski, ORB: an efficient alternative to sift or surf, in: Computer Vision (ICCV), 2011 IEEE International Conference On, IEEE, 2011, pp. 2564–2571.
[38] S.N. Sinha, J.-M. Frahm, M. Pollefeys, Y. Genc, Feature tracking and matching in video using programmable graphics hardware, Mach. Vis. Appl. 22 (1) (2011) 207–217.
[39] S. Tzeng, A. Patney, J.D. Owens, Task management for irregular-parallel workloads on the gpu, in: Proceedings of the Conference on High Performance Graphics, Eurographics Association, 2010, pp. 29–37.
[40] S. Warn, W. Emeneker, J. Cothren, A. Apon, Accelerating sift on parallel architectures, in: 2009 IEEE International Conference on Cluster Computing and Workshops, IEEE, 2009, pp. 1–4.

[41] C. Wu, Siftgpu-v400: An mature open-source gpu implementation of sift, 2013. http://cs.unc.edu/~ccwu.
[42] S. Yang, S. Scherer, Direct monocular odometry using points and lines, arXiv preprint arXiv:1703.06380.
[43] G. Yu, J.-M. Morel, ASIFT: an algorithm for fully affine invariant comparison, Image Process. On Line 1 (2011) 11–38.

Zhihao Li received the B.Eng. degree from Guangdong University of Technology, Guangdong, China, in 2015. He is currently working toward the Ph.D. degree at the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His research interests include heterogeneous computing, large-scale parallel computing and optimizing fast Fourier transform (FFT) library.

Haipeng Jia received the B.Eng. and Ph.D. degrees from Ocean University of China, Qingdao, China, in 2007 and 2012, respectively. He was a Visiting Ph.D. student with the Institute of Software, Chinese Academy of Sciences, Beijing, China, from 2010 to 2012. He is currently an Assistant Professor with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences. His research interests include heterogeneous computing, many-core parallel programming method and computer version algorithms on multi-/many-core processors

Yunquan Zhang (Member, IEEE) received the Ph.D. degree in computer software and theory from the Chinese Academy of Sciences, Beijing, China, in 2000.

He is a Full Professor of computer science with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His research interests are in the areas of high performance parallel computing, with particular emphasis on large scale parallel computation and programming models, high-performance parallel numerical algorithms, and performance modeling and evaluation for parallel programs. He has published over 100 papers in international journals and conferences proceedings.

Shice Liu received the B.S. degree from the College of Computer and Control Engineering, Nankai University, China, in 2016, and pursuing a master degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. He was an intern with the Institute of Robotics and Automatic Information System, Nankai University, China, in 2015. He is currently a Master student with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His research interests include computer vision, deep learning and simultaneous location and mapping.

Shigang Li received the Ph.D. degree from the Department of Computer Science and Technology, University of Science and Technology, Beijing, China, in 2014. He was a Visiting Ph.D. student with the University of Illinois at Urbana-Champaign from 2011 to 2013. His research interests include large-scale parallel computing based on MPI, heterogeneous computing, and scalable deep learning algorithms on multi-/many-core clusters.

Xiao Wang received B.S. degree from the School of Information Science and Engineering, Yunnan University, China, in 2017. He is currently a Master degree candidate in the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His research interests include program optimization and parallel computing.

Hao Zhang received the B.S. degree in applied mathematics from Guangdong university of technology in 2011. He is currently pursuing the Ph.D. degree in computer science at Fudan university. His research interests are mainly in the fields of data mining, causal inference and image processing.