J Real-Time Image Proc (2018) 14:267–277
https://doi.org/10.1007/s11554-014-0446-6

ORIGINAL RESEARCH PAPER

A real-time implementation of SIFT using GPU

K. Aniruddha Acharya · R. Venkatesh Babu · Sathish S. Vadhiyar

Received: 2 September 2013 / Accepted: 1 August 2014 / Published online: 20 August 2014

Abstract Scale-Invariant Feature Transform (SIFT) is one of the widely used interest point features. It has been successfully applied in various computer vision algorithms like object detection, object tracking, robotic mapping and large-scale image retrieval. Although SIFT descriptors are highly robust towards scale and rotation variations, the high computational complexity of the SIFT algorithm inhibits its use in applications demanding real-time response, and in algorithms dealing with very large-scale databases. This paper presents a parallel implementation of SIFT on a GPU, where we obtain a speed of around 55 fps for a 640 $\times$ 480 image. One of the main contributions of our work is the novel combined kernel optimization that has led to a significant improvement of 12.2 % in the execution speed. We compare our results with the existing SIFT implementations in the literature, and find that our implementation has better speedup than most of them.

Keywords SIFT · GPU · CUDA · Combined kernel

1 Introduction

Many computer vision algorithms involve the process of image feature extraction. Image features are interest points or regions of an image, which are invariant to translation, rotation, or scale changes of objects in the image. The need for image features is twofold. One is to convert the high-dimensional raw images to the better manageable lower dimensional signals, and the other is to achieve signals that can be relied upon for tasks that follow in the image processing pipeline. There are a huge number and a variety of image feature descriptors in the literature. Some of the well-known interest point detectors are: Harris corners [5], Shi Tomasi corners [11], SIFT [7] and SURF [2]. There are many other image features extracted from image structures such as edges, gradients and regional properties such as color. Of the many feature extraction algorithms, Scale-Invariant Feature Transform (SIFT) [7] is one of the most widely used point features that are scale, rotation and to some extent illumination invariant.

Although SIFT is known to be a good image feature, its computational complexity limits its use in real-time applications as well as in applications that process large-scale image and video databases. For a video-based human–computer interaction application, it is necessary that the video analysis happens at least at the frame rate of the input video, which is typically more than 15 fps. And the feature extraction step being only a part of the video analysis, must be faster than 15 fps. Large-scale image and video processing algorithms have to deal with huge databases. Hence a slow feature extraction step is not suitable, as it may take several days or weeks of computation time. Sequential implementations of SIFT are known to have high execution times. The open source sequential implementation SIFT++ [13] takes around 3.3 s on a 2.4 GHz processor for a 640 $\times$ 480 image. This can allow a maximum frame rate of around 0.31 fps, which is much less than the minimum frame rate expected.

Graphics processing units (GPUs) are processors that were traditionally meant for image rendering in graphics

K. A. Acharya · R. Venkatesh Babu (✉)
Video Analytics Laboratory, Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, India
e-mail: venky@serc.iisc.ernet.in; venky@serc.iisc.in

K. A. Acharya
e-mail: aniruddha753@gmail.com

S. S. Vadhiyar
Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, India

applications. As compared to the typical CPUs which have 4–16 cores, GPUs have a huge number of processor cores, 160–400, while each core has computational capability lower than that of the CPU cores. The motivation for building such a system was to maximize the throughput of the system, i.e., the number of operations or tasks completed per unit time rather than minimizing the latency of each task, which is the aim of the traditional CPUs. In the recent years, GPUs are also being used for general-purpose computing (GPGPUs). The huge parallel processing capability of the GPUs is being harnessed for accelerating computation for various non-graphics algorithms too [4]. Image processing, being the inverse process of computer graphics, is also suitable for execution on GPUs as can be seen in [3]. This paper presents a parallel implementation of SIFT on GPU and describes some of the techniques that have led to the speedup of its execution.

The aim of our work is to accelerate the execution of the SIFT feature extraction and feature description algorithms. We also aim at scalability of our implementation with respect to the image size. We have used GPU in this work to parallelize because of its potential of obtaining huge speedups as compared to the traditional multi-core parallel strategies. One of the main contributions of our work is the novel combined kernel optimization explained in Sect. 5. Our results show significant improvement in the SIFT execution time for all the image resolutions considered. A preliminary version of this work can be found in [1].

The paper is organized as follows. Section 2 briefly describes the SIFT algorithm. Section 3 describes other GPU implementations of SIFT. Section 4 gives an overview of CUDA and GPU. Section 5 describes the methodology adopted in our implementation. Section 6 describes the experiments conducted. Section 7 discusses the results obtained and Sect. 8 concludes the paper.

2 SIFT

This section will briefly describe the various steps of the SIFT [7] algorithm.

2.1 Scale space construction

This stage involves (1) construction of multiple Gaussian convolved images, and (2) computing the difference of Gaussian images. The input image is repeatedly convolved with Gaussian kernels of different variances to form a set of $O \times (S+3)$ Gaussian blurred images, where O and S are the number of octaves and the number of levels in each octave, respectively, in the output scale space. Each octave has an image dimension of half that of the previous octave. For creating the base image of the initial octave (-1st octave), the input image is super-sampled by a factor of two using linear interpolation. The difference of successive Gaussian convolved images is computed to produce $O \times (S+2)$ difference of Gaussian (DoG) images, forming the scale space.

2.2 Keypoint detection and localization

This stage involves finding the scale space extrema. For the central S images in each of the octaves, and for each of the pixels in them, values of 26 neighbors (8 in the same level, 9 in the next higher level and 9 in the next lower level) are inspected to detect local minima or maxima. Of the extrema points, only the high contrast pixels are retained. This is done by imposing a minimum threshold on the absolute value of the DoG image pixels. Among the retained pixels, those having a high edge response are eliminated, and the remaining ones form the set of SIFT keypoints. The Gaussian standard deviation corresponding to the scale in which the keypoint was obtained is regarded as the scale of the keypoint. For obtaining accurate positions of the keypoints, quadratics are fit using the samples of DoG scale space near the keypoints and the locations of the local maxima or minima of the quadratics are used as the locations of the keypoints.

2.3 Keypoint orientation computation

Each SIFT keypoint has an orientation associated with it. This step requires gradients of images near the keypoints. For each of the SIFT keypoints, a patch of 16×16 pixels around the keypoint is considered from the gradient image with a scale closest to the scale of the keypoint. A histogram of gradient orientations of pixels in the patch is constructed with 36 equally spaced orientation bins. The orientation with highest histogram value is considered as the orientation of the keypoint.

2.4 Keypoint description

SIFT keypoints are associated with 128 dimensional SIFT descriptors. A 16×16 patch is considered around the keypoint and rotated to point along the orientation of the keypoint to make the descriptor rotation invariant. The patch is divided into four 4×4 sub-patches, and individual 8-bin histograms are computed for each of the sub-patches. All the 16 8-bin histograms are concatenated to form the 128-dimensional keypoint descriptor.

3 Related work

The problem of speeding up the SIFT algorithm has been worked on by researchers in the past and few have implemented SIFT on GPU [12]. They have ported the scale space construction, difference of Gaussian, keypoint detection and orientation assignment steps of SIFT to GPU, but have retained the step of descriptor creation to the CPU. Sinha et al. [12] have used OpenGL/CG for their implementation on NVIDIA Geforce 7900 GTX card and reported a speed of 10 fps for a 640 $\times$ 480 video.

Heymann et al. [6] have presented an implementation of SIFT on GPUs that have the capabilities of dynamic branching and multiple render targets (MRTs) in the fragment processor. All the steps of SIFT that have been implemented were executed on GPU. Dynamic branching has been used for detecting feature points from the difference of Gaussian images, and MRTs have been used for gradient calculation. The authors have used NVIDIA QuadroFX 3400 GPU and reported an execution speed of 17.24 fps for a 640 $\times$ 480 video.

Zhang et al. [16] use multi-core CPU systems for speeding up the execution of SIFT. They use data level parallelism and task level parallelism on the major time consuming steps of SIFT. The sequence of the steps in the original SIFT algorithm has been changed to decrease load imbalance among the cores. The orientation assignment and feature description steps for all the scales of an octave are merged and done only once at the end of keypoint detection of all scales in an octave. Implementation has been done using OpenMP on a dual-socket quad-core system with total 8 cores, each core running at 2.33 GHz. The execution time reported is 45 fps for a 640 $\times$ 480 image.

Wu [15] has implemented two versions of SIFT on GPU, one using GLSL and the other using CUDA. These implementations execute at 23.0 and 27.1 fps, respectively, for a 640 $\times$ 480 image on the NVIDIA 8800 GTX GPU.

Warn et al. [14] explored and compared acceleration of SIFT using OpenMP and GPU. They have focused on very large images obtained from aerial and satellite photography to reduce the time taken to extract information from large image databases. In the GPU version, only the convolution step of SIFT was offloaded to GPU. System used was NVIDIA FX 5800 and the implementation was done using CUDA. A speedup of 1.9$\times$ is reported for a 4,136 $\times$ 1,424 image as compared to execution on a 2.33 GHz core.

4 CUDA programming model for GPU

Compute Unified Device Architecture (CUDA) is a programming model for modern NVIDIA GPUs and provides a simplified programmers view of the GPU for general purpose programming. From the programmers point of view, a GPU consists of a number of thread blocks which in turn contain multiple threads capable of running in parallel. Threads within the same thread block have better capabilities of synchronization and cooperation among each other than threads from different blocks. Computations can be offloaded to GPU by means of invoking a CUDA kernel. During every kernel invocation, the programmer must specify the number of blocks (thread blocks) and the number of threads per block to be used, and a function that must be run on all the threads. The maximum number of thread blocks and threads is much more than the actual number of independent parallel cores in the system. In fact, there is large scope for performance improvement if the number of blocks and threads used is more than the number of cores. A scheduler does context switching among different blocks to optimize the use of the processing elements.

5 Methodology

This section describes the method and techniques used by the proposed implementation of SIFT. We have used CUDA for programming the GPU. Our implementation offloads all the steps of SIFT to GPU.

5.1 Scale space construction

2D Gaussian kernels are known to be separable kernels. That is, a 2D convolution by a Gaussian kernel is equivalent to two consecutive 1D convolutions, one across the rows and one across the columns. It has been found that the use of this separable property results in large improvements in performance for convolutions in GPUs [10]. We have adapted the NVIDIA kernel for 2D separable convolution [10] for parallelizing construction of each of the scale space Gaussian convolved images.

Combined kernel optimization The original sequential SIFT algorithm proceeds by computing the Gaussian convolved images from the base octave till the highest octave, while sequentially computing all of the scales in each octave. Thus, the natural sequence of kernel calls during Gaussian scale space construction would be as shown in Algorithm 1.

Algorithm 1

```
super-sample using GPU;
for all octaves from $-1$ to $O-2$ do
    for all scales from $-1$ to $S+1$ do
        convolve using GPU;
    end for
    sub-sample using GPU;
end for
```

This sequence is maintained in all of the GPU implementations of SIFT in the literature known to us. We achieve further parallelism here by altering this order.

In case, all the sub-sampling steps are done just after the super-sampling, the base level Gaussian convolved images of all octaves are available before the start of the outer loop in Algorithm 1, and we are only left with a sequence of convolutions that are independent of each other, but which deal with different-sized images. We replace all of these convolution kernels by a single combined kernel. This is illustrated in Algorithm 2.

Algorithm 2

```
super-sample using GPU;
for all octaves from $-1$ to $O-2$ do
    sub-sample using GPU;
end for
convolve using GPU with combined kernel;
```

The top portion of Fig. 1 depicts the individual kernel grids for the convolutions, if Algorithm 1 is used. Initially, all the levels of octave -1 are convolved one by one using GPU. For each of these convolutions, a unique kernel call has to be made, with the right number of blocks to cover the entire image of the particular scale. When each of the kernels is invoked, the code corresponding to the kernel function is transferred to GPU. The requested number of thread blocks are initialized, and the convolution of the particular level of octave -1 gets initiated. Each of the thread blocks has access to a unique variable called *blockIdx* that identifies the block. The input to the kernel is the base image of the octave that is considered, and the output of the kernel is the convolved image corresponding to the particular scale under consideration. The parallel convolution algorithm uses *blockIdx* and the input image and produces the output. After completion of the kernel execution, the next level is considered. The three largest grids of Fig. 1 represents the kernel grids of octave -1 that are sequentially executed on the GPU. This process is then sequentially repeated for higher octaves.

In the above-explained process, parallelism is exploited only within the convolution of each scale. On the other hand, Algorithm 2 exploits parallelism among the scales as well using the combined kernel. One of the main challenges in the design of such a combined kernel is fixing of the grid size in such a way that all scale convolutions take place with the right input and output. Identification of the virtual individual grids within the combined grid is crucial, and must be accomplished without much code overhead.

All the thread blocks of convolution in Algorithm 1, irrespective of the octave and level of the scale space they populate, execute the same piece of convolution code, but with different inputs, outputs and parameters. We use this property to assimilate the thread blocks of all octaves and levels in a single combined kernel. We also observe that by keeping the number of threads per block a constant, there is a simple pattern in the number of blocks required for different scales:

If the number of thread blocks required per convolution kernel of each level of the -1st octave is (W, H), i.e. $N_{-1} = (W, H)$ (N stands for Number of Blocks), then $N_0 = (W/2, H/2)$, for the 0th octave, and $N_1 = (W/4, H/4)$, for the 1st octave, since we maintain the number of threads per block (16×16) across all scales.

The bottom part of Fig. 1 depicts the process of merging individual thread blocks. The total number of blocks N_{COMBINED} of new combined kernel depends on the way the individual kernels are laid out. It can be seen from Fig. 1 that there are multiple ways of placing the thread blocks on the grid. The bottom left layout uses $4.5WH$ thread blocks and is better than the bottom right layout, which uses $6WH$ thread blocks. The best placement strategy is the one that minimizes the area of the rectangle that is needed to enclose all the thread blocks, as CUDA kernels can be invoked only using a rectangular grid.

Maintaining the structure and shape of the individual kernel grid inside the combined kernel is important. This is because the kernel code must be able to identify the correct original kernel it belonged to, so that it can identify the correct IO (input, output) parameters, define and use a modified version of the *blockIdx* variable to compensate for the offset arise due to combining of kernels. For doing the initial selections of IO parameters, the combined kernel function must have some additional code in the beginning. And during the kernel call, IO parameters required by all of the original kernels must be passed.

Ideally, the combined kernel must complete execution within the time required for executing one instance of the -1st octave kernel. This is so because, ideally all the thread blocks in a grid execute in parallel, and the time required by the levels of other octaves is lesser than that required by the base octave, due to the reduced image dimensions in the higher octaves. Hence, ideally this optimization results in huge speedup of computations.

However, practically it is not true that all the blocks execute completely in parallel. The number of processing elements is usually lesser than the number of threads asked for by the kernel call. The thread blocks are kept in a queue and are scheduled by a scheduler for execution on the processors, in a time shared manner. Due to this practical limitation, the huge speedup expected in the ideal case is not possible. But, we obtain considerable performance boost due to this optimization, the details are shown in the

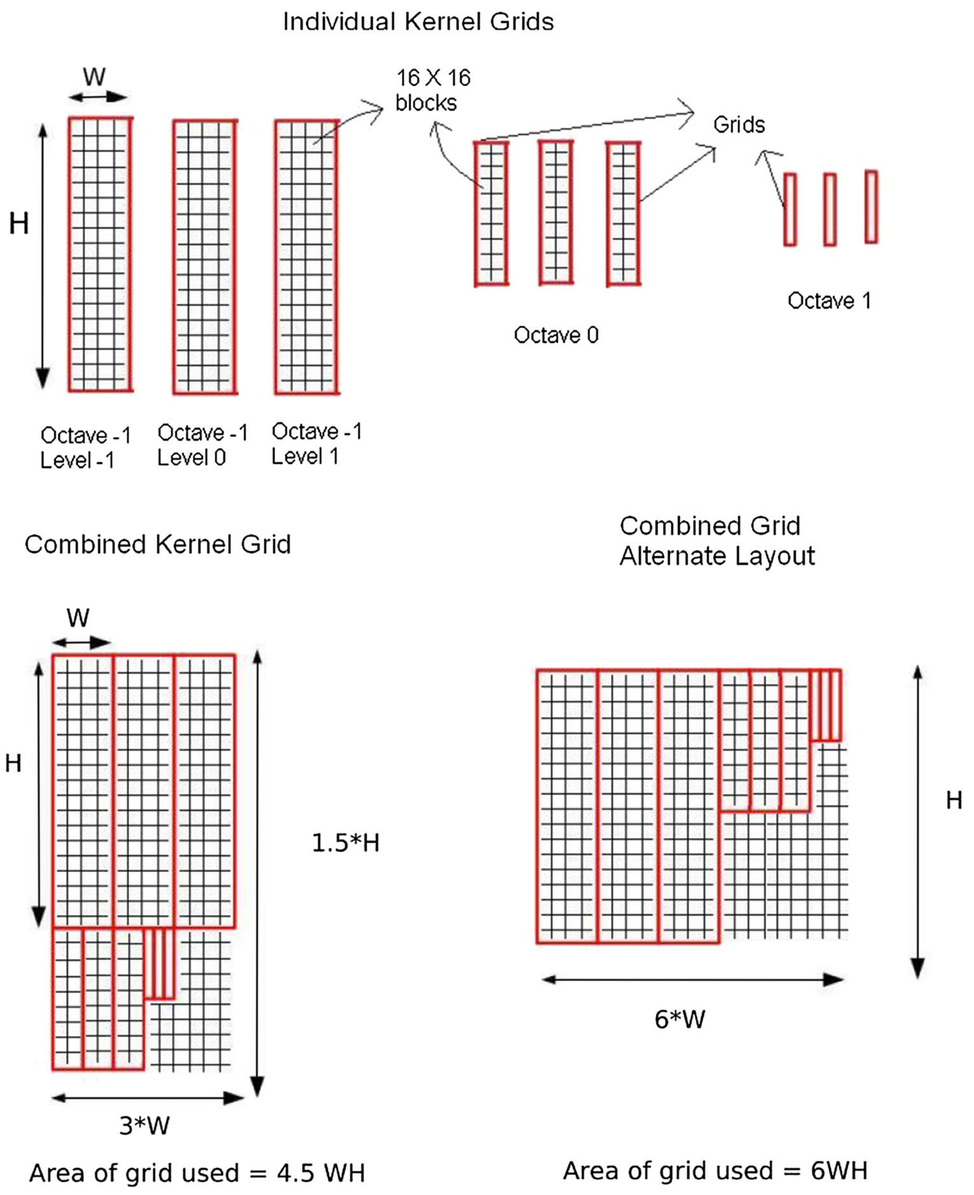

Fig. 1 *Top* individual kernel grids for convolution, *bottom left* combined kernel grid, *bottom right* alternate placement strategy

experiments section. By invocating thread blocks of all the kernels at once, we impart a larger flexibility for the scheduler to optimize the utilization of the processors. This is one of the reasons for the increased performance.

There is another cause for the success of this optimization. The number of kernel calls is decreased from $O \times (S + 2)$ to 1. Hence we save on the kernel invocation time, in which the kernel code is copied to the device memory and, SMs and SPs (hardware elements) are initialized. In addition, there is only a small quantity of code overhead in the combined kernel function.

DoG computation involves subtracting images and is straightforward to implement in CUDA. Fine-grained parallelism is exploited by assigning one thread per DoG pixel. It can be observed that the subtraction kernels for each of the scales in the scale space are independent of each other. The combined kernel optimization is used for this step also, and we reduce $O \times (S + 2)$ kernels to a single kernel.

5.2 Keypoint detection

In this step, local extrema of the central S DoG images of all of the octaves are to be detected. A total of $O \times S$ kernels are used, in which each thread checks for extremum at a single pixel in the DoG image. The challenge involved in this step is the method for storing the detected keypoints. The number of keypoints generated by the kernel and the order in which points are generated are uncertain before the execution of the kernels. A global array in the device memory to store the keypoints and a global index variable to keep track of the number of keypoints found is the usual method in most of the sequential implementations. But this method cannot be directly applied on GPU because of the synchronization problem. There is no defined order in which keypoints get detected. Hence, we use atomic add instructions to enforce serialization of keypoint storage among all the thread blocks.

Due to the use of atomic instructions, increasing the number of parallel elements increases the contention for accessing the shared variable. Using a combined kernel for this stage increases the number of thread blocks that can potentially simultaneously contend for the shared index variable. In fact, we have observed that using combined kernel here has increased the execution time of this stage, due to the increased serialization.

Another method for storing the keypoints is to have a 2D array of the same size as the input image and make each thread write into the array at the location that the thread is responsible for, if a keypoint is detected. But this would not result in an array that has keypoints in consecutive locations. And the kernels that are executed after this stage require the keypoints to be sequentially placed. Hence, we chose not to use this data structure.

5.3 Keypoint orientation and description

A single kernel function is used for both orientation assignment and descriptor construction. This is done because both the steps require the same gradient information around the keypoint. Unlike the sequential SIFT, we do not pre-compute gradient image for the entire scale space. The gradient of only a 16×16 patch around the keypoint is computed. The number of thread blocks used for the kernel is the number of keypoints detected in the previous stage. Each thread block is responsible for one keypoint.

Each thread block has 17×17 threads. Initially, threads in a block load the 17×17 patch around the keypoint into the shared memory. This is done as a caching mechanism to decrease the latency of memory accesses. A synchronization point is used to ensure all threads have completed loading. Then the gradient is computed for 16×16 pixels.

Histogram creation is not trivial to implement on GPU. This is because all the threads update the shared histogram simultaneously, and hence we have the synchronization problem. Use of atomic instructions significantly reduces the performance in this case, because of the small number of bins and large contention among threads. There is a technique for parallel histogram creation in the literature that uses multiple histogram copies which are updated by different parallel elements and merges the different partial histograms to form the final histogram [9]. We have adapted this technique for computing histogram of orientations. To find the maximum of histogram, we adopt the well-known GPU reduce algorithm, with the MAX operator.

6 Experiments

All the experiments are done using a single GPU of the NVIDIA S1070 device. The GPU is connected to a 2.4 GHz AMD Opteron 8378 processor. The GPU operates at 1.296 GHz and has 4 GB RAM and a total of 240 cores.

Pinned memory usage: *cudaMemcpy* function works much faster with pinned CPU memory blocks than the usual memory blocks [8]. Pinned memory is characterized by the property that it is never swapped out of main memory by the operating system. This avoids the process of duplicating the source memory block on CPU memory before memory transfer. Our implementation uses pinned memory to store the image on CPU. Although pinned memory is attractive for a fast implementation, it cannot be used in abundance, as it is a scarce resource. We tackle this problem by adopting tiling of the input image.

If the input image has dimensions larger than a threshold t, it is split into multiple smaller image tiles, and each of the tiles is processed sequentially. This also lets our code to be executed on GPUs with much lesser RAM capacity, as the amount of GPU memory allocated only depends on the size of the tile, and not on the size of the input image.

Use of CPU: While the GPU works on one tile of the image, the CPU copies the next tile of the image to the pinned CPU memory. Our experiments have shown that the time required for this transfer is only slightly less than the time required for the GPU to process one tile. Hence the GPU waiting time (idle time) is almost zero. We have used the pthreads library for the parallel execution on CPU.

We have executed our SIFT code for images of different dimensions ranging from 320×240 (CIF) to $2{,}048 \times 1{,}536$ (3M). The number of octaves is fixed to 6, and the number of levels within each octave is 3. We compare the execution time of our implementation with that of the open source CPU implementation by [13]. We compare [13] code with two versions of our GPU implementation, namely with and without the combined kernel optimization. Figure 2 shows the speedup of the proposed method with and without the combined kernel. Speedup us defined as the ratio of $\text{time}_{\text{vedaldi}}$ and time_{GPU}. It is seen from the figure that the speedup factor is quite high in general for all image sizes. It can be observed that the version with combined kernel has enhanced speedup for most of the image sizes.

One characteristic feature of both the plots in Fig. 2 is that the speedup shoots up for 640×480 image size, then

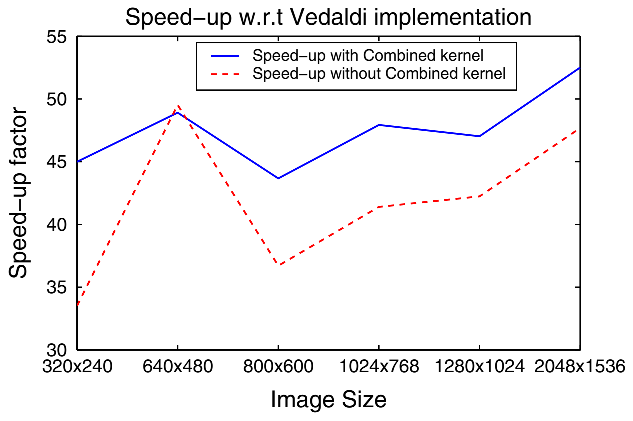

Fig. 2 Speedup of our code w.r.t Vedali [13] vs. image size. Speedup $= \frac{\text{time}_{\text{Vedaldi}}}{\text{time}_{\text{GPU}}}$

Table 1 Effect of combined kernel optimization

	Execution time without combined kernel (ms)	Execution time with combined kernel (ms)	Percentage decrease in time (%)
320 × 240	7.197	5.361	25.51
640 × 480	17.638	17.861	−1.26
800 × 600	34.975	29.403	15.93
1,024 × 768	51.074	44.116	13.62
1,280 × 1,024	83.432	74.932	10.19
2,048 × 1,536	178.565	162.149	9.19

drops down, and then shows a monotonic increasing behavior. This happens because, for images larger than 640 × 480, we introduce tiling of images. It may seem from the graph that the speedup would be much higher without tiling, and it is a better option to avoid tiling. But tiling is inevitable for a scalable implementation that should be able to run on GPUs with low device memory. We store the entire scale space, DoG scale space, the detected keypoints along with their descriptors in the device memory.

One other observation that can be made from Fig. 2 is that the code without combined kernel preforms slightly better than the one with the combined kernel for the 640 × 480 case. It was mentioned in the Sect. 5.1 that with increase in the number of thread blocks in the combined kernel, the scheduler gets larger flexibility to optimize the GPU resource usage. But there is an upper limit on the number of blocks that can be easily handled by the scheduler, which depends on various factors like memory access patterns of the kernel code, number of synchronization points, atomic operations and so on. For the 640 × 480 case, the number of blocks reached the saturation level, because of which it was not possible to gain speed using the combined kernel. However, we have observed in our experiments that combined kernel when applied only to DoG computation, which has much simpler kernel code than convolution, results in speedups even for the 640 × 480 case.

Table 1 shows the effectiveness of the combined kernel optimization for different image sizes. For an average-sized image, the time advantage due to the combined kernel is 12.2 %.

Table 2 shows the execution times of our code, and that of Vedaldi [13] for versions with and without the use of up-sampling in the base octave. It can be observed that the version with up-sampling requires significantly more time for the sequential code, but the GPU implementation scales well, with manageable speeds even for HD images.

For an image of dimensions 640 × 480, our implementation runs in 17.861 ms (55.98 fps) for the version without up-sampling of the base octave, and in 51.643 ms (19.36 fps) for the version with up-sampling of the base octave, which is more than real time and better than all of the GPU implementations discussed in the Sect. 3.

The accuracy of our output SIFT points are found by comparing their positions with that obtained by the SIFT executable of David Lowe (author of the SIFT paper Lowe [7]) and with the keypoints obtained from the code given by Vedaldi [13]. As compared to keypoints of Lowe [7], there are some additional points and some missing points in our implementations output, but 85.6 % of the points are sub-pixel accurate; i.e., the location error is less than 1 pixel. This is because we have not implemented the accurate keypoint localization step of the SIFT algorithm, which fits a quadratic in the scale space and localizes the keypoints to a more accurate scale space extrema. Figure 3 shows the keypoints of the proposed implementation (marked by red plus) and those of Vedaldi [13] without the accurate keypoint localization step (marked by green squares).

The effectiveness of descriptor can be seen in Fig. 5 which shows matches between two images of the same scene but obtained from different viewpoints. 108 keypoint matches were obtained among the 147 and 163 keypoints in the two images.

6.1 Application to object tracking

We further test our GPU implementation by considering the object tracking application of SIFT. Object tracking is a popular computer vision problem, in which the location of the object of interest is given in the first frame of a video, and the object location is to be found out for the rest of the video. Among the various approaches to object tracking, there are approaches that use SIFT features. Some of them combine other techniques with SIFT to produce accurate output, robust towards noise and occlusion. In this paper,

Table 2 Comparison of execution time (in ms) of the proposed implementation with that of Vedaldi [13] sequential implementation

Image size	320 × 240	640 × 480	800 × 600	1,024 × 768	1,280 × 1,024	2,048 × 1,536
Proposed (with up-sampling)	16.145	51.643	85.674	141.247	217.553	487.233
Vedalai [13] (with up-sampling)	895.971	3,367.579	5,190.937	8,766.560	15,658.585	41,482.853
Proposed (without up-sampling)	5.361	17.861	29.403	44.116	74.932	162.149
Vedalai [13] (without up-sampling)	241.206	873.572	1,283.937	2,114.593	3,523.696	8,514.868

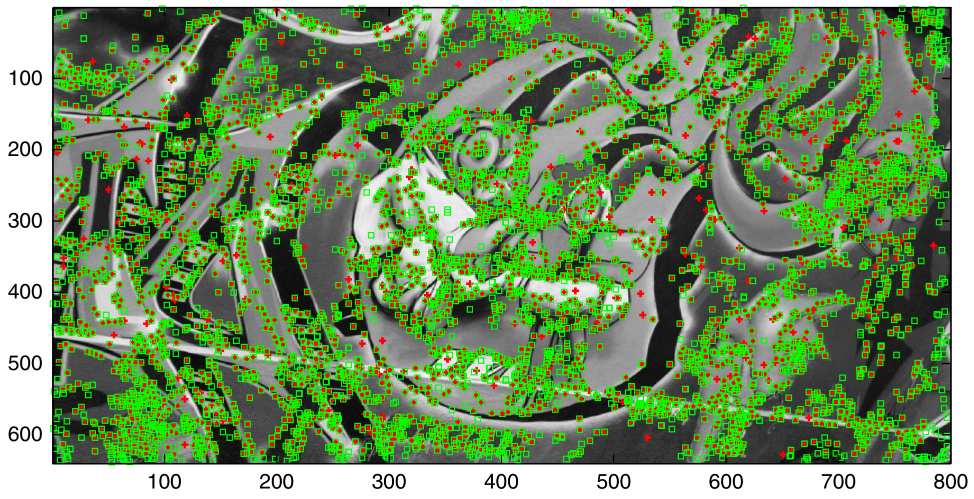

Fig. 3 Comparison of our keypoints (*red plus*) with keypoints of Vedaldi [13] (*green squares*)

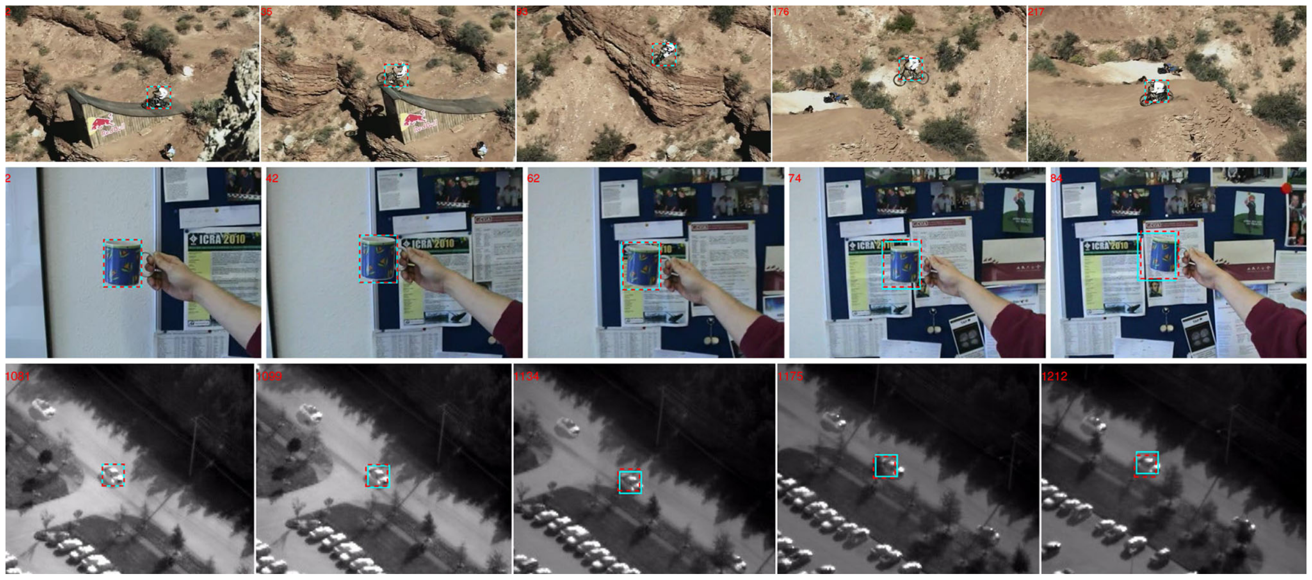

Fig. 4 Tracking results for jump, cup and pktest videos. The *dotted red boxes* represent GPU implementation result and the *cyan boxes* represent Lowe's implementation result

we have considered a naive approach, as our objective is to only compare our implementation with Lowe's implementation.

SIFT features are extracted from all the frames of the video. For every frame, SIFT feature correspondences are found between the previous frame and the current frame. This gives a set of location pairs $(x\text{Previous}_i, y\text{Previous}_i)$ and $(x\text{Current}_i, y\text{Current}_i)$. The difference between the corresponding location vectors gives an estimate of the motion vector of the object. Since we have multiple motion vectors, median of the x and y components are chosen as the object's motion vector $\text{MV} = (\text{mv}_x, \text{mv}_y)$. The object

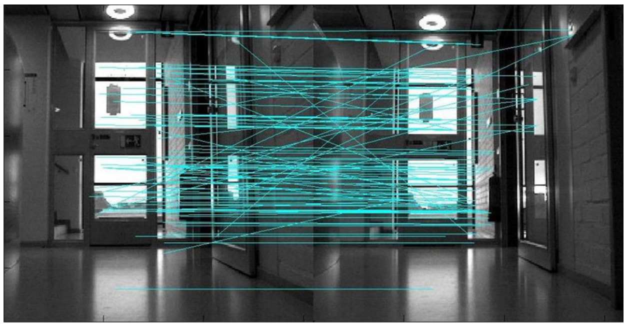

Fig. 5 Keypoint matching by descriptor comparison

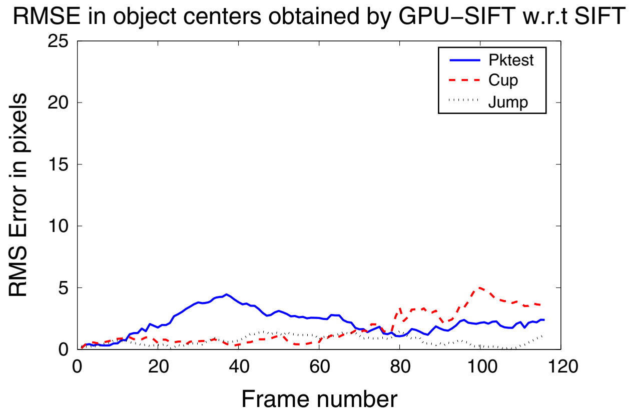

Fig. 6 RMSE in object centers obtained by GPU-SIFT w.r.t SIFT

location in the previous frame is propagated to the current frame using

$$\mathrm{obj}_{x,\mathrm{current}} = \mathrm{obj}_{x,\mathrm{previous}} + \mathrm{mv}_x$$

and

$$\mathrm{obj}_{y,\mathrm{current}} = \mathrm{obj}_{y,\mathrm{previous}} + \mathrm{mv}_y.$$

We have implemented the above-mentioned method to object tracking using both Lowe's SIFT and our GPU-SIFT. Experiments were done using three standard tracking videos publicly available, namely 'jump', 'cup' (obtained from http://www.votchallenge.net/), and 'pktest'. Figure 4 shows the estimated object locations for three videos. The cyan bounding boxes represent the results of Lowe's SIFT implementation and the dotted red boxes represent the results of our GPU-SIFT implementation. It can be seen that the boxes closely match each other.

Figure 6 shows the root mean squared error (RMSE) between the object center locations estimated by the two implementations for each frame considered. It can be noted that the error is below 5 pixels for most of the cases, and is much less than the dimensions of the object. Figure 7 shows RMSE of object location with respect to the ground

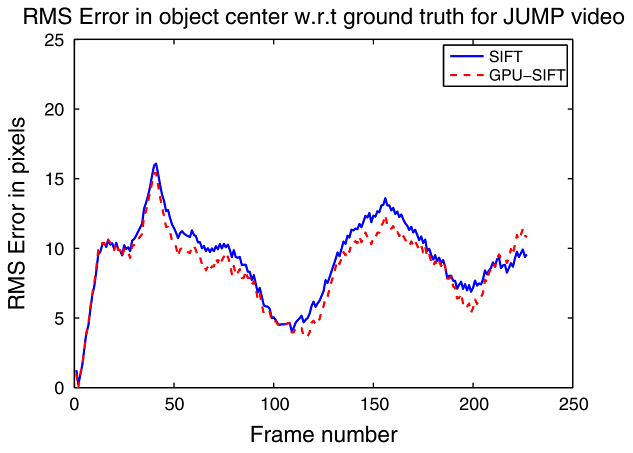

Fig. 7 RMSE in object center w.r.t ground truth for 'JUMP' video

truth for the 'jump' video. The error pattern in the output of both implementations closely matches each other.

7 Discussion

The characteristics and effects of the combined kernel can be inferred from Table 3. Table 3 shows the speed gains obtained due to the difference of Gaussian combined kernel. The total execution time of a kernel includes two parts, namely the GPU time and the time required to invoke the kernel (referred to as CPU time), which involves copying the kernel code to GPU and initializing the GPU. It can be seen from the table that the reduction in GPU time is not consistently good across various image sizes. In fact there is appreciable reduction in GPU time only for smaller grid sizes (320 $\times$ 240 and 800 $\times$ 600). We believe that this can improve with GPUs with larger number of cores. However, reduction in total kernel time (GPU time + CPU time) is consistently high across all image sizes. This is because each combined kernel replaces 30 ordinary kernels, and requires kernel code transfer and GPU initialization only once, as compared to 30 kernel code transfers for the equivalent ordinary kernels.

By introducing the combined kernel optimization, we only add a sequence of simple 'if–else' statements in the kernel code. As for the execution time, this modification does not affect much, but the code becomes slightly complex. The number of conditions may be quite large (e.g., 18) for easy maintainability of code. The conditions have to be derived manually using the layout of the combined grid. This process of generating the 'if–else' code section of the kernel code from the layout can be automated, but is currently done manually in our implementation.

The maximum number of arguments allowed in the kernel function call is usually less than the number of arguments required to be passed to a combined kernel, if the arguments are passed in the traditional method of using

Table 3 Execution time of and speed gain obtained by difference of Gaussian combined kernel (1 combined kernel replaces 30 ordinary kernels)

Image size	No of combined kernel calls	Grid size	Shared memory per block (bytes)	GPU time per combined kernel call (µs)	GPU time per 30 ordinary kernels (µs)	Reduction in GPU time (µs)	Total time (GPU + CPU) per combined kernel call (µs)	Total time per 30 ordinary kernels (µs)	Reduction in total time per 30 ordinary kernels (µs)	Total reduction in time per DoG operation (µs)
320 × 240	1	[125, 22]	96	126.784	175.936	49.152	176	1,424	1,248	1,248
800 × 600	3	[150, 28]	96	193.008	221.728	28.72	231	1,112	881	2,643
1,024 × 768	6	[135, 36]	96	305.115	298.176	−6.939	348	1,204	856	5,136
1,280 × 1,024	9	[160, 33]	96	334.859	350.784	15.925	422	1,277	855	7,695
2,048 × 1,536	20	[155, 36]	96	361	328.768	−32.232	408	1,227	819	16,380

pointers to images. This is because each combined kernel will have to access the entire scale space. To circumvent this problem, we allocate *octaves* × *levels* number of pointers in the device memory of GPU and copy the location of scale space images to these pointers. With this, we only send one double pointer in each combined kernel, instead of several pointers. There is only a slight memory overhead of storing *octaves* × *levels* number of pointers.

As for the memory bandwidth, the combined kernel turns out to be advantageous. It avoids unnecessary repeated transfer of the kernel code from CPU memory to the GPU device memory. Virtually, several individual kernels use the code that is transferred only once, bringing about enhanced memory bandwidth.

8 Conclusions and future work

We have presented an implementation of SIFT on GPU which has a significant advantage in the execution performance as compared to the CPU-based implementations. Additionally, our solution is highly scalable with respect to the image size because of the use of image tiling. The uniqueness of our work as compared to the other GPU implementations in the literature is that we have a well defined optimization technique—the combined kernel optimization which results in around 12 % increase in the execution speed. Moreover, the applicability of combined kernel optimization is not restricted to SIFT; it is a general technique that can be used wherever there are similar independent kernels that are to be applied to different inputs of the same/different sizes. Many of the algorithms that deal with the scale space have the potential to be accelerated using this optimization.

Our optimization is similar to the task level parallelism used in the standard multi-core parallel programming. Typically, GPUs are considered to harness only the data level parallelism, using a large number of threads. This paper has shown the use of a variant of task level parallelism on GPUs, where each of the thread blocks is considered to undertake a separate task.

Extending the SIFT implementation to include matching of SIFT descriptors on GPU is one of our plans for the future. We will be identifying other algorithms that can benefit from the proposed optimization and implement them.

References

1. Acharya, A., Venkatesh Babu, R.: Speeding up SIFT using GPU. Fourth national conference on computer vision. Pattern recognition, Image Processing and Graphics (NCVPRIPG), IEEE, pp. 1–4 (2013)

2. Bay, H., Tuytelaars, T., Van Gool L.: SURF: features. In: ECCV 2006, pp. 404–417. Springer (2006)
3. Fung, J., Mann, S.: Using graphics devices in reverse: GPU-based image processing and computer vision. In: IEEE international conference on multimedia and expo, pp. 9–12 (2008)
4. Harish, P., Narayanan, P.: Accelerating large graph algorithms on the GPU using CUDA. In: High performance computing, pp. 197–208. Springer (2007)
5. Harris, C., Stephens, M.: A combined corner and edge detector. In: Alvey vision conference, Manchester, UK, vol. 15, p. 50 (1988)
6. Heymann, S., Muller, K., Smolic, A., Frohlich, B., Wiegand, T.: SIFT implementation and optimization for general-purpose GPU. In: Proceedings of the international conference in Central Europe on computer graphics, visualization and computer vision, p. 144 (2007)
7. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. Int J Comp Vis **60**(2), 91–110 (2004)
8. NVIDIA Corporation: CUDA C best practices guide (2010)
9. Podlozhnyuk, V.: Histogram calculation in CUDA. NVIDIA Corporation, White Paper (2007a)
10. Podlozhnyuk, V.: Image convolution with CUDA. NVIDIA Corporation white paper, vol. 2097, no 3 (2007b)
11. Shi, J., Tomasi, C.: Good features to track. In: IEEE conference on computer vision and pattern recognition, pp 593–600 (1994)
12. Sinha, S.N., Frahm, J.M., Pollefeys, M., Genc, Y.: Feature tracking and matching in video using programmable graphics hardware. Mach Vis Appl **22**(1), 207–217 (2011)
13. Vedaldi, A.: An open implementation of the SIFT detector and descriptor. UCLA CSD (2007)
14. Warn, S., Emeneker, W., Cothren, J., Apon, A.: Accelerating SIFT on parallel architectures. In: IEEE international conference on cluster computing and workshops, pp 1–4 (2009)
15. Wu, C.: SiftGPU: a GPU implementation of scale invariant feature transform (SIFT). http://cs.unc.edu/ccwu/siftgpu (2007)
16. Zhang, Q., Chen, Y., Zhang, Y., Xu, Y.: SIFT implementation and optimization for multi-core systems. In: IEEE international symposium on parallel and distributed processing, pp. 1–8 (2008)

K. Aniruddha Acharya received his B.E. degree in Telecommunication Engineering from PES institute of Technology, Bangalore. Currently he is pursuing his M.Tech at SERC, Indian Institute of Science, Bangalore, India.

R. Venkatesh Babu received his Ph.D. degree from Dept. of Electrical Engineering, Indian Institute of Science, Bangalore. He held postdoctoral positions at NTNU, Norway and IRISA/INRIA, Rennes, France, through ERCIM fellowship. Subsequently he worked as a research fellow at NTU, Singapore. He spent couple of years working in industry. He is currently an Assistant Professor and convenor of Video Analytics Laboratory at Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, India. His research interests include video analytics, human–computer interaction, computer vision, and compressed domain video processing. He is a senior member of IEEE.

Sathish S. Vadhiyar is an Associate Professor in Supercomputer Education and Research Centre, Indian Institute of Science. He obtained his B.E. degree from the Department of Computer Science and Engineering at Thiagarajar College of Engineering, India in 1997 and received his Master's degree in Computer Science at Clemson University, USA in 1999. He graduated with a PhD from the Computer Science Department at University of Tennessee, USA in 2003. His research areas are in building application frameworks including runtime frameworks for irregular applications, hybrid execution strategies, and programming models for accelerator-based systems, processor allocation, mapping and remapping strategies for Torus networks for different application classes including irregular, multi-physics, climate and weather applications, middleware for production supercomputer systems, and fault tolerance for large-scale systems. Dr. Vadhiyar is a member of IEEE and has published papers in peer-reviewed journals and conferences. He was a tutorial chair in eScience 2007, and session chair in eScience 2007 and ICS 2013, and served on the program committees of conferences related to parallel and grid computing including IPDPS, CCGrid, eScience and HiPC.