Speeding up SIFT using GPU

Aniruddha Acharya K
Video Anaytics Laboratory
Supercomputer Education and Research Center
Indian Institute of Science, Bangalore, India
Email: aniruddha753@gmail.com

R. Venkatesh Babu
Video Anaytics Laboratory
Supercomputer Education and Research Center
Indian Institute of Science, Bangalore, India
Email: venky@serc.iisc.in

***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 640×480 image. One of the main contributions of our work is the novel combined kernel optimization that has led to a significant improvement of 21.79% in the execution speed. We compare our results with the existing implementations in the literature that accelerate SIFT, and find that our implementation has better speedup than the most of them.**

***Keywords*—GPU, SIFT, parallel programming**

I. 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 variety of image feature descriptors in the literature. Some of the well known interest point detectors are: Harris corners [12], SIFT [7] and SURF [13]. Some feature extractors identify lines as features and regions as features. Of the many feature extraction algorithms, SIFT (Scale Invariant Feature Transform) [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 applications that process large scale image and video databases. For a video based human-computer interaction application, it is necessary that the video analysis happen 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++ [6] takes around 3.3s on a 2.4GHz processor for a 640×480 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 applications. As compared to the typical CPUs which have 4 to 16 cores, GPUs have a huge number of processor cores, 160 to 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 [8]. Image processing being the inverse process of computer graphics is also suitable for execution on GPUs as can be seen in [9]. This paper presents a parallel implementation of SIFT on GPU and describes some of the techniques that have led to the speed up 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 use for 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 section 5. Our results show significant improvement in the SIFT execution time for all the image resolutions considered.

The paper is organized as follows. Section 2 describes other GPU implementations of SIFT. Section 3 briefly describes the SIFT algorithm. Section 4 gives an overview of CUDA and GPU. Section 5 describes the methodology adopted in out implementation. Section 6 describes the experiments conducted, section 7 discusses about the results obtained and section 9 concludes the paper.

II. Related work

The problem of speeding up the SIFT algorithm has been worked on by researchers in the past. Sinha et al. [1] have implemented SIFT on GPU. 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. have used OpenGL/CG for their implementation on NVIDIA Geforce 7900 GTX card and report a speed of 10 fps for a 640×480 video.

Heymann et al. [2] 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 report an execution speed of 17.24 fps for a 640×480 video.

Zhang et al. [3] 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 640×480 image.

Wu [5] has implemented two versions of SIFT on GPU, one using GLSL and the other using CUDA. These implementations execute at 23.0 fps and 27.1 fps respectively for a 640×480 image on the NVIDIA 8800 GTX GPU.

Warn et al. [4] explore and compare 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 databases. In the GPU version, only the convolution step of SIFT was offloaded to GPU. System used is NVIDIA FX 5800 and the implementation was done using CUDA. A speed up of 1.9x is reported for a 4136×1424 image as compared to execution on a 2.33 GHz core.

III. SIFT

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

A. Scale space construction

This stage involves *(i)* construction of multiple Gaussian images, and *(ii)* computing the difference of Gaussian images. The input image is repeatedly convolved with Gaussians of different variances to form a set of $O \times (S+3)$ Gaussian 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 (-1^{st} octave), the input image is super-sampled by a factor of two using linear interpolation. Next, the difference of successive Gaussian images is computed to produce $O \times (S+2)$ difference of Gaussian images forming the scale space.

B. Keypoint detection and localization

This stage involves finding the scale space extrema. For each of the scale space images and for each of the pixels in them, values of 26 neighbors are inspected to detect local extrema. The standard deviation corresponding to the scale in which the keypoint is obtained is regarded as the scale of the keypoint.

C. 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.

D. 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 separate 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.

IV. 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 programmer's 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.

V. METHODOLOGY

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

2-D gaussian kernels are known to be separable kernels. That is, a 2-D convolution by a Gaussian kernel is equivalent to two consecutive 1-D 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 2-D separable convolution [10] for parallelizing construction of each of the scale space Gaussian images.

Combined kernel optimization: The original sequential SIFT algorithm proceeds by computing the Gaussian 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 the pseudo code:

```
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.

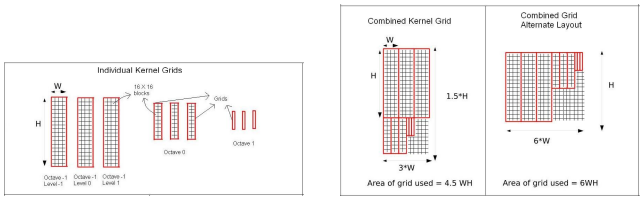


Fig. 1. Left - Individual kernel grids for convolution, Right - Combined kernel grid, Alternate placement

In case all the sub-sampling steps are done just after the super-sampling, the base level Gaussian images of all octaves are available before the start of the loop in the pseudo code, 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.

Let the number of thread blocks required per convolution kernel of each level of the -1^{st} octave be (W, H),
i.e. $NB^{-1} = (W, H)$. (NB stands for Number of Blocks)
Since we maintain the number of threads per block (16×16) across all octaves and levels, we have
$NB^0 = (W/2, H/2)$, for the 0^{th} octave, and
$NB^1 = (W/4, H/4)$, for the 1^{st} octave.

All the thread blocks of convolution, 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. Figure 1 depicts the process of merging individual thread blocks. The total number of blocks $NB^{COMBINED}$ of new combined kernel depends on the way the individual kernels are laid out. It can be seen from Figure 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 right 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 that arises 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 -1^{st} 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 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 are 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.

Difference of Gaussian computation involves subtracting images and is straight forward 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.

VI. 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 4GB 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 [11]. 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 (QCIF) to 2480×1536 (HD). The number of octaves is fixed to 6, and the number of levels within each octave is 3. Figure 2 shows the speed up of our code with respect to the CPU implementation of Vedaldi [6] for both for versions with and without the use of up-sampling in the base octave.

Table I 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 21.79%.

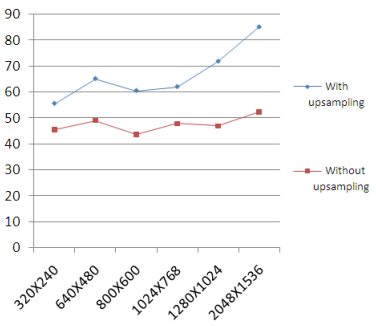
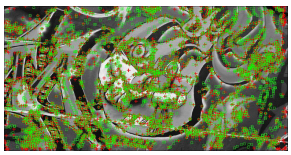


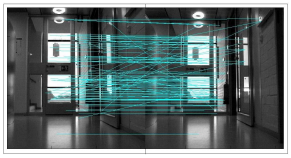
Fig. 2. Speed up of our code w.r.t [6] vs. image size

TABLE I. EFFECT OF COMBINED KERNEL OPTIMIZATION

	Decrease in Convolution time	Decrease in DoG time	Total time advantage	Speed-up (in %)
320X240	620ms	523ms	1,143ms	21.32%
640X480	445ms	489ms	934ms	5.22%
800X600	14,131ms	2,063ms	16,194ms	55.07%
1024X768	4,735ms	3,780ms	8,515ms	19.30%
1280X1024	5,881ms	4,439ms	10,320ms	13.77%
2048X1536	15,278ms	10,750ms	26,028ms	16.05%



(a) Comparison of our keypoints (green squares) with David Lowe's points (red plus)



(b) Keypoint matching by descriptor comparison

Fig. 3. Keypoint accuracy and descriptor efficiency

For an image of dimensions 640×480, our implementation runs in 17.861 ms (55.98 fps) for the version without upsampling of the base octave, and in 51.643 ms (19.36 fps) for the version with upsampling of the base octave, which is more than real time and better than all of the GPU implementations discussed in section 2.

The accuracy of our output SIFT points are found by comparing their positions with that obtained by David Lowes SIFT executable. 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. The left subfigure of Figure 3 shows the keypoints of this implementation (marked by green squares) and David Lowes implementation (marked by red plus). A small percentage of the points do not match with David Lowes points. 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.

The effectiveness of descriptor can be seen in the right subfigure of Figure 3 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.

VII. DISCUSSION

The proposed GPU implementation achieves considerable speed up for all the image sizes considered. The general trend of speed up with respect to image size can be inferred from the graph in Figure 2. There is a slight dip in the speed up from 640×480 to 800×600 image sizes. This is because we impose tiling of the image for image sizes more than 640×480. It may seem that introducing tiling has decreased the performance slightly, but that is true only for small image sizes. Moreover, large images cannot be handled if tiling is not done, because of memory constraints of the GPU. It can be observed from the graph that for large images the speed-up is almost linear, and has not reached saturation. This trend will continue for the unexplored larger images too, since we do not rely on more device resources for larger images.

VIII. 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. 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] Sudipta N. Sinha, Jan-Michael Frahm, Marc Pollefeys, and Yakup Genc, *"Feature Tracking and Matching in Video Using Programmable Graphics Hardware"*, Vision and Applications, March 2007.

[2] Heymann, S., Frhlich, B., Medien, F., Mller, K., Wiegand, T.,*SIFT implementation and optimization for general-purpose GPU.* In WSCG07 (2007).

[3] Q. Zhang, Y. Chen, Y. Zhang, and Y. Xu, *Sift implementation and optimization for multi-core systems* in Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, 2008, pp. 18

[4] Warn, S.; Emeneker, W.; Cothren, J.; Apon, A., *"Accelerating SIFT on parallel architectures"* Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on , vol., no., pp.1,4, Aug. 31 2009-Sept. 4 2009, doi: 10.1109/CLUSTR.2009.5289155

[5] Wu,C.: *SiftGPU: A GPU implementation of scale invariant feature transform (SIFT).* http://cs.unc.edu/ ccwu/siftgpu (2007)

[6] Vedaldi, A.: *An open implementation of the SIFT detector and descriptor*. Tech. Rep. 070012, UCLA CSD (2007)

[7] Lowe, D.G.: *Distinctive image features from scale-invariant keypoints*. International Journal of Computer Vision. 60(2), 91110 (2004). doi:10.1023/B:VISI.0000029664.99615.94.

[8] Harish and P. Narayanan. *Accelerating large graph algorithms on the GPU using CUDA*. In Proceedings of 2007 International Conference on High Performance Computing, Dec 2007

[9] Fung, J.; Mann, S., *"Using graphics devices in reverse: GPU-based Image Processing and Computer Vision"*, Multimedia and Expo, 2008 IEEE International Conference on , vol., no., pp.9,12, June 23 2008-April 26 2008 doi: 10.1109/ICME.2008.4607358

[10] V.Podlozhnyuk, *Image Convolution with CUDA*. NVIDIA whitepaper

[11] *CUDA C BEST PRACTICES GUIDE*, NVIDIA design guide

[12] Harris, Chris, and Mike Stephens. *"A combined corner and edge detector"*, Alvey vision conference. Vol. 15. 1988.

[13] Bay, Herbert, Tinne Tuytelaars, and Luc Van Gool. *"Surf: Speeded up robust features"*, Computer Vision ECCV 2006. Springer Berlin Heidelberg, 2006. 404-417.