SiftCU: An Accelerated Cuda Based Implementation of SIFT

Mahdi S. Mohammadi[1], Mehdi Rezaeian[2]

[1] Electrical and Computer Engineering Department, Yazd University, Yazd, Iran
m.s.mohammadi.k@stu.yazduni.ac.ir
[2] Electrical and Computer Engineering Department, Yazd University, Yazd, Iran
mrezaeian@yazduni.ac.ir

Abstract. Scale Invariant Feature Transform (SIFT) is a popular image feature extraction algorithm. SIFT's features are invariant to many image related variables including scale and change in viewpoint. Despite its broad capabilities, it is computationally expensive. This characteristic makes it hard for researchers to use SIFT in their works especially in real time application. This is a common problem with many image-processing related algorithm. Utilizing graphical processing unit (GPU) through parallel programming is an affordable solution for this issue. In this paper we present a GPU-based implementation of SIFT using Compute Unified Device Architecture (CUDA) programming framework. We compare our CUDA-based implementation, namely siftCU, with CPU-based serial implementations of SIFT both in feature matching accuracy and time consumption. Results show our implementation can gain 4x speed up over serial CPU implementation even though we have used a low end graphic card while using a powerful CPU for test platform.

Keywords: CUDA, GPGPU, Image Processing, Feature Extraction, Parallel Programming, SIFT

1 Introduction

Like any other scientific field, researchers studying image processing face many different difficulties to establish an adequate work. A common and important difficulty is providing enough computing resources. Researchers usually need high amount of computing resources to implement and test their devised systems. Although researcher try to optimize their proposed algorithms, most of the time, they end up spending considerable amount of time and money to test their proposed system. To alleviate effects of these problems, it is highly recommended to all researchers to harness cheap and high performance GPUs. In this paper we have tried to address this issue with presenting a CUDA-based implementation of SIFT algorithm.

Graphical processing unit (GPU) is processing unit of a graphic card. Historically GPUs had been used only in basic computer graphic tasks. The traditional form of use for GPUs changed when Nvidia Company introduced CUDA programming framework at the end of 2007. CUDA is a framework for general-purpose programming on GPUs

(GPGPU). Because of different architecture comparing to CPUs, GPUs have greater potential for performing stream processing. Stream processing is a computer-programming paradigm. Simply, it means emulating parallel processing through SIMD (single instruction multiple data). GPU's peak performance greatly exceeds CPU's peak performance when comparing two GPU and CPU that are on same price range.

SIFT is an image feature extraction algorithm. SIFT features extracted are invariant to image scale and rotation, and are shown to provide robust matching across a substantial range of affine distortion, change in 3D viewpoint, addition of noise, and change in illumination [1]. SIFT is a very popular algorithm for image matching. It has been used in object recognition, image retrieval and many other fields. We use CUDA framework for our implementation, because it is a well-documented and highly supported schema for GPU-based programming.

Remaining sections are arranged as follow: first, we are going to review related works. Section 3 briefly describes SIFT algorithm and CUDA programming framework. Our GPU-based implementation of SIFT (siftCU) is discussed in section 4. Results for accuracy and speed up compression between SIFTpp and siftCU are presented at section 5. Finally, section 6 concludes the paper.

2 Related Works

Since SIFT introduction in 2004, there have been attempts to implement it on GPU. Sinh, Farhm, Pollefeys and Genc presented "GPU-SIFT" in 2006 [2]. This was before CUDA or any other popular GPGPU-based programming framework release. In their work, they used more traditional GPU-based programming interface; openGL/Cg. Heymann and his colleagues also presented a GPU-based implementation using traditional GPU-programming interfaces [3]. Heymann and et al. reported a 20 frames/sec processing speed. After introduction of CUDA in 2007, researchers started using this new framework for GPGPU. In 2009, Warn, Emeneker, Cothren and Apon used CUDA for running SIFT on NVIDA's GPUs [4]. They only implemented part of the algorithm on GPU. Warn and et al. reported that GPGPU-based implementation had 1.9X speed up over CPU-based implementation. Their test platform included an Intel Xeon E6550 CPU and an Nvidia FX 5800 graphic card. In 2011, Huang and his associates used a CUDA-based SIFT for registration of Synthetic Aperture Radar (SAR) images [5]. In addition to SIFT, they have also implemented SAR image features registration with CUDA. Huang and et al. reported that CUDA-based implementation of entire process showed a 19.6X speed up over CPU-based implementation while using a powerful GPU, namely C2050, and Intel Xeon E5506 as test platform. Recently, Yang and Chen used GPGPU-based SIFT, implemented on CUDA, for moving foreground detection in dynamic background [6]. Using Nvidia GeForce 9800GT, which is a low end GPU, they still managed to gain a 1.3X speed up.

3 Background

3.1 SIFT Algorithm

SIFT is a very popular image feature extraction algorithm presented by Lowe at 2004 [1]. The popularity of SIFT is due to the fact that the features extracted by this algorithm are invariant to many image related variables including scale and rotation. As described by Lowe, SIFT algorithm includes four steps [1]:

- **Scale-space extrema detection**: At this step, we detect location and scale of candidates that can be repeatedly assigned under differing views of the same object. This can be done by searching for stable keypoints over all scales, using a function of scale known as scale space. Lowe proposed we could use scale-space extrema in the difference-of-Gaussian function convolved with image as candidates.
- **Keypoint localization**: Every extrema found in the first step of the algorithm is a keypoint candidate. Nevertheless, before accepting any candidate, a detailed fit must performed to the nearby data for location, scale, and ratio of principal curvatures. This process can determine two measures: one for rejecting points that have low contrast, and one for rejecting points that poorly localized along edges.
- **Orientation assignment**: Third step in SIFT algorithm is to assign orientation to each keypoint. After assigning a consistent orientation to each keypoint according to properties of local image, the keypoint descriptor will be computed relative to this orientation. This process makes keypoints descriptor invariance to image rotation.
- **Keypoint description**: Last step in SIFT algorithm is to create descriptors for keypoints, which were found in previous steps. It begins with sampling gradient magnitudes and orientations inside an area around the location of the keypoint. The coordinates of the feature vector and the gradient orientations are rotated based on the keypoint orientation, so descriptors are invariant to changes in orientation.

3.2 GPU Computing Using CUDA

Graphics **P**rocessing **U**nit (GPU) is processing unit of a graphic card. Traditionally developers used these processing units exclusively for basic computer graphic application. In the last decade when their great capability, for stream processing, were noticed; developers started to use them for general-purpose programming. At the early years programming on GPUs were hard and complicated. In the last six years, this has been changed due to introduction of programming frameworks like CUDA. These frameworks make GPU-based programming easier and more accessible for everyone.

The CUDA programs are typically comprise of two parts: ‘Host Code’ and ‘Device Code’. The host code part of CUDA codes can be any C++ standard code (or other C++ library code). This part of CUDA program will be compiled with a standard C++ compiler available on host machine, like gnu compiler, and run on host CPU. The second part of CUDA programs is the part that will run on GPU and it is consist of some kernels. Kernels are like normal functions but they run on GPU instead of CPU.

The CUDA execution flow is built upon the idea of launching a kernel with a grid comprising of blocks. A single block comprises of a collection of threads. Each thread has a unique identifier within its block, and each block has a unique global identifier. These are combined to create a unique global identifier per thread [7]. There are three type of memory available in CUDA: registers, shared memory, and global memory. Registers are fastest memory available on GPU. They are local for each thread. Usually the variables defined inside a kernel are placed inside registers. Threads inside a block can share a fast onboard memory called shared memory. If accessed properly, shared memory could be as fast as registers. Finally, there is a large but slow memory accessible from all running threads in a device called Global memory.

4 siftCU

Our implementation of SIFT in CUDA has three main stages. Nevertheless since SIFT operates on grayscale images, there is a preprocessing stage to convert color images to 256 level grayscale images. Our implementation codes and the pictures used for analyzing are freely available here [8]. We have tried to execute as much of the code as possible on the device because transferring data between host and device memory is time-consuming and abundant data transfer will impair performance [9]. In addition, we want to harness GPU's high performance computation power as much as possible. You can see general execution flow of our implementation in Figure 1. Except for calculating Gaussian filters, we only have kernel calls in host code and all calculation take place in device. Next, we will discuss each major stages of implementation.

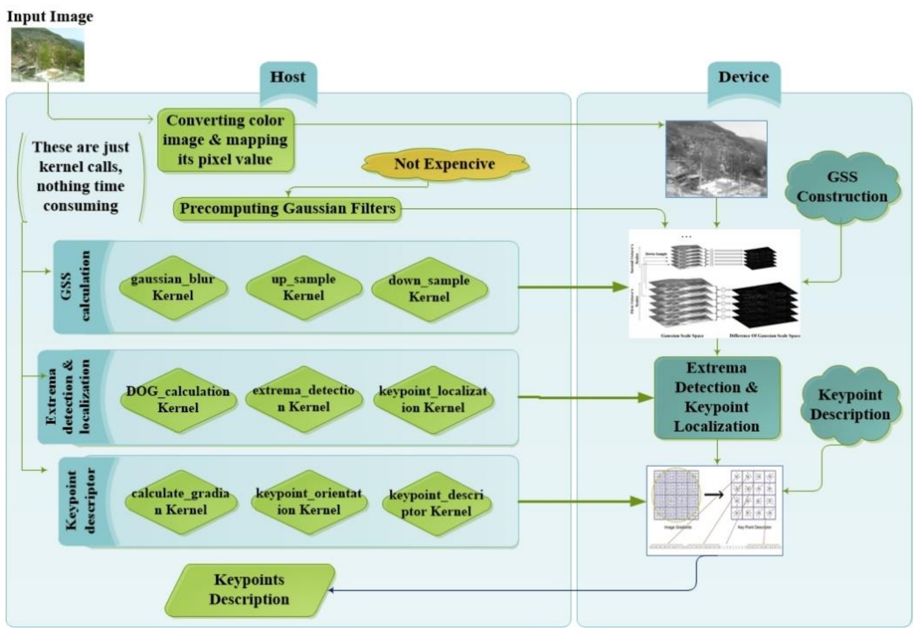
Input Image
Host
Device
Converting color image & mapping its pixel value
These are just kernel calls, nothing time consuming
Not Expencive
Precomputing Gaussian Filters
GSS Construction
GSS calculation
gaussian_blur Kernel
up_sample Kernel
down_sample Kernel
Extrema detection & localization
DOG_calculation Kernel
extrema_detection Kernel
keypoint_localization Kernel
Extrema Detection & Keypoint Localization
Keypoint Description
Keypoint descriptor
calculate_gradian Kernel
keypoint_orientation Kernel
keypoint_descriptor Kernel
Keypoints Description

Figure. 1. SiftCU execution flow

4.1 GSS Construction

In first step, we need to construct Gaussian scale space. A Gaussian filter should be applied to input image recursively at the right spatial space with right scale (sigma). There are two options for performing this process: applying a single 2-dimensional Gaussian filter; or applying two 1-dimensional Gaussian filter, one vertically and one horizontally. Computationally speaking second option is more efficient than former [10]. You can see filter operation process in figure 2.

As you can see in figure 2, to construct a GSS image, a kernel first applies a Gaussian filter on rows of pervious scale (Left image) while storing it in a transpose manner (Middle image). Then 'same filter' is applied on rows of transposed results to obtain next GSS image (Right image). Storing intermediate results in a transpose form would optimize memory access. In addition, this way we get to use the same kernel for both horizontal and vertical filter. This process is in contrast with approach used in [5], which uses two distinct kernels for this process. We could have merge horizontal and vertical kernels into single kernel but generally speaking for the sake of performance it is better to divide task into smaller parts as long as this would not result in abundant data transfer [9].

To further optimizing our code, we only directly compute all scales for the first octave. After that, first three scale of subsequent octaves will be computed by down sampling three top scales of previous octave. Lowe suggested this practice, only for the first scale of each octave. Nevertheless, it is obvious that this process can also work for second and third scale of each octave.

4.2 Extrema Detection and Keypoint Localization

In order to optimize memory access by using data localization, we process this stage independently for each octave. First, the DOG images for an entire octave are computed, then one kernels per scale search for extrema in that octave. We implemented DOG calculation kernel so each thread will calculate one pixel for each scale of an octave.

After construction of DOG images, a kernel will check whether a pixel value is extrema or not. This kernel stores pixel location and scale as candidates in case it is an extrema. Then another kernel, running one thread per each extrema, extracts keypoints

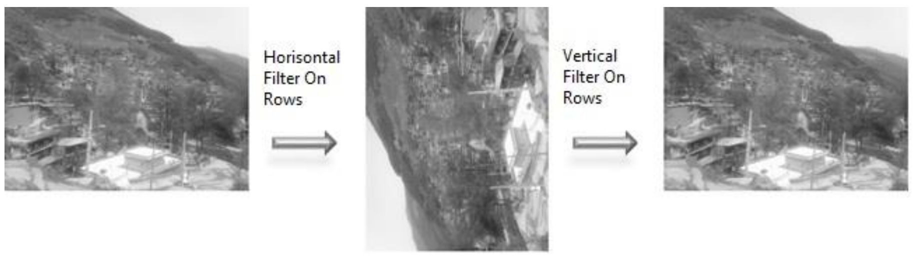

Figure. 2. To construct a GSS image, we first apply Gaussian filter on rows of pervious scale (Left image) while storing it in a transpose manner (Middle image), then apply same filter on rows to obtain next GSS image (Right image).

from candidates. Second kernel checks candidates with contrast and edge response threshold if a candidate satisfies conditions, we choose it as a keypoint.

When trying to store a candidate or keypoint data, threads may access shared array simultaneously. In this situation, we need a mechanism to avoid race condition. Race conditions means simultaneous access of shared data by at least two different process or thread. This could lead to an anomaly in shared data's final value. CUDA has some atomic functions that we could use them to implement semaphore like 'lock variable' for our purposes. When kernel wants to store a detected extrema, we use atomicAdd function on a counter variable to do this. This counter is shared among an entire scale. The partial code bellow demonstrates this process:

```
if(stat > 0) // is an Extremum?
{
    int in = atomicAdd( counter ,1);
    extrema[in].iy = y; extrema[in].ix = x;
    extrema[in].o = o;   extrema[in].is = s;
}
```

Using synchronization functions like atomicAdd can impair overall performance [9]. In order to decrease scale of synchronization, which results in less performance impairment, we use different part of an array for each scale of an octave. We cannot completely avoid using this function since there is no other option for avoiding race condition.

In this stage, shared memory could not be utilized because of data dependency. We investigated some approaches to exploit shared memory. However, all of them produced very complicated code with lots of exceptions. Exceptions mean branch inside code. Too many branches will have great negative impact on performance.

4.3 Orientation Assignment and Keypoint Descriptor

In this stage, we process each octave independently like previous stage. First, a kernel calculates magnitude and orientation of image gradient for an entire scale. Each thread processes one pixel's data. Then one thread per available keypoint will be created and run orientation assignment kernel. This kernel assigns at least one orientation for each keypoint. As we discussed in the orientation assignment subsection, when describing sift algorithm, a single keypoint locations could be assigned more than one orientation. When a thread assigns more than one orientation to a single keypoint, it creates a new keypoint for every extra orientation. Newly created keypoints have the same location and scale as the original keypoint. The only difference between them is their orientation. After calculating orientations of an entire octave's keypoints, next, we call keypoint descriptor kernel. Here, each thread computes descriptor for one keypoint.

Although the entire process of computing descriptor for one keypoint is a huge job. Nevertheless, we choose to assign this job just to a single thread. That is because, due to high data dependency, its segmentation can decrease performance. Besides, we process keypoints of an entire octave simultaneously. Since higher parallelization can result in higher performance, we were able to achieve adequate speed up over CPU-based implementation.

5 Compression and Results

In this section, we are going to test our implementation for time consumption and accuracy. We compare siftCU with SIFTpp in both speed up and accuracy. SIFTpp is a well-known open source implementation of SIFT algorithm in C++ [11]. It is a serial implementation and considered to be optimized and reliable. Our implementation is inspired by SIFTpp. Before presenting comparison results, this section analyze siftCU memory usage.

5.1 System Set Up and Configuration

In test setup, we used an Intel Core i7-2600 with 4 GB DDR3 RAM for host configuration and an Nvidia GeForce GT 440 with 1 GB DDR3 Memory for device configuration. The processor that we used for testing is one of the most powerful processors for PC computers available in the market. This processor currently has price range around 350$. On the other hand, the GPU is a low end GPU and it can be bought for not more than 70$. Part of our objective were to make sift execution affordable. By affordability, we mean both affordability of execution time and financial source spend on providing appropriate hardware for execution.

Table 1 Running time: siftCU Vs. SIFTpp. All time scales are in milliseconds.

Subject	Stage	SIFTpp	siftCU	Speed up
Image 1 (549*800)	1	437	96	**4.55**
	2	38	18	**2.11**
	3	907	436	**2.08**
	Overall	**1382**	**550**	**2.51**
Image 2 (678*1024)	1	681	128	**5.32**
	2	56	27	**2.07**
	3	1512	641	**2.36**
	Overall	**2249**	**796**	**2.83**
Image 3 (1400*1000)	1	1369	248	**5.52**
	2	151	55	**2.75**
	3	5305	1415	**3.75**
	Overall	**6825**	**1718**	**3.97**
Image 4 (1600*1200)	1	1908	357	**5.34**
	2	162	73	**2.22**
	3	4122	1147	**3.59**
	Overall	**6192**	**1577**	**3.93**

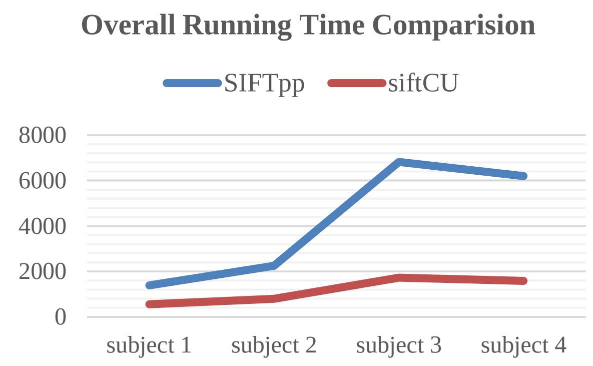

Figure. 3. SiftCU & SIFTpp overall execution time

5.2 Speed Up Test

You can see test results for speed up in table 1. Results have been broke down for three main stages. We were able to achieve a very good speed up in first stage through effective shared memory utilization. There is high data dependency in second and third stages, therefore, we were not able to gain speed up as much as first stage. Nevertheless because of the reasons explained in section 5, implementing all stages in GPU is preferable to implementing some of them in Host. Figure 5 shows overall time consumption of SIFTpp and siftCU for four images in table 1. SiftCU's speedup over SIFTpp grows as picture size and number of extracted keypoints grow. Although image 3 is actually smaller than image 4 but it takes longer to process. That is because image 3 has much more keypoints compared to image 4. This results show Nvidia GeForce 440 GT cannot make siftCU practical for real time applications but if we use a high end GPU, like Nvidia GTX 670, it can easily satisfy real time demands.

Table. 2. Feature matching results for SIFTpp and siftCU

	SIFTpp			**siftCU**		
	Keys	**match**	mismatch	Keys	**match**	mismatch
Set 1	5455	**115**	0	4925	**181**	1
	1315			1217		
Set 2	5455	**7**	5	4926	**19**	5
	1785			1566		
Set 3	5455	**137**	3	4926	**265**	1
	3199			2514		

5.3 Precision Test

The keypoint matching is based on a simple but effective matching algorithm described by Lowe [1]. When you want to see whether a keypoint has a match inside bunch of other keypoints, the algorithm is as follow:

- Find first and second closest descriptor to sample keypoint descriptor. The distance factor is Euclidean distance.
- If first distance is smaller than six-tenth of second distance then first distance is a match otherwise sample does not have a match.

You can see precision comparison results in table 2. The results from two implementations for each set are not identical. This can be attributed to small differences in the value of some variables and thresholds. Nevertheless, as it is obvious, our implementation has not impaired precision of matching. We demonstrated keypoint matching between test sets in figure 4.

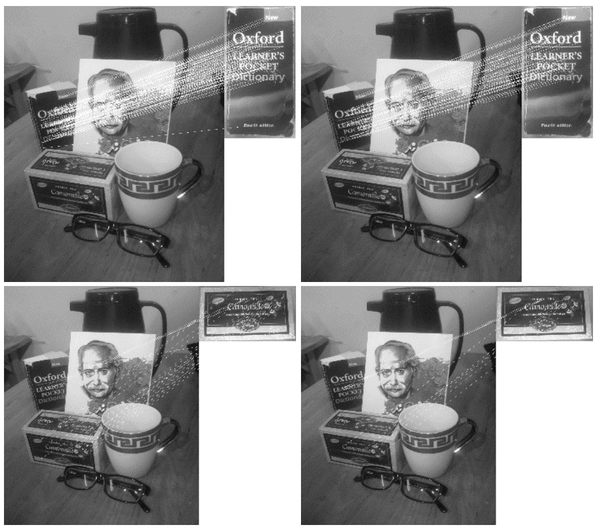

Figure. 4. Feature matching for images in table 2. Top) set 1: left siftCU, right SIFTpp; Bottom) set 2: left siftCU, right SIFTpp.

6 Conclusion

In this paper, a GPU-based implementation of sift feature extraction algorithm was presented. We used CUDA, which is a C++ based framework for GPGPU. Our implementation, namely siftCU, is simpler and more efficient than previous works. We compare siftCU with SIFTpp in speed up. Part of our objective were to make sift execution affordable. By affordability, we mean both affordability of execution time and financial resources. SIFTpp was executed on a powerful but expensive CPU, namely Intel Core i7-2600. On the other hand, siftCU was executed on a cheap low end GPU, namely

Nvidia GeForce 440 GT. Results showed that our implementation could gain 4x speed up over SIFTpp. This means if we utilize all 4 processing cores of Core i7-2600 using a multi-core implementation, in best case scenario for multi-core implementation, our implementation would match up with that implementation in speed up. The results are satisfactory considering the CPU, we used in test set up worth more than five times the GPU that was used. Our implementation could gain more than 30x speed up if we were to use a high end GPU like Nvidia GTX 670 that is in same price range with Core i7-2600, since it has 14 times more cores than 440 GT, and more than 6 times memory bandwidth. SiftCU can be easily used for real time applications of SIFT utilizing high end graphic cards but a low end graphic card like 440 GT cannot satisfy real time demands. We also showed that siftCU could be as precise as any other implementation of SIFT.

7 References

[1] Lowe, D. G.: Distinctive Image Features from Scale-Invariant Keypoints. Journal of Computer Vision, vol. 60, no. 2, pp. 91-110 (2004)

[2] Sinha, S.N., Frahm, J.M., M. Pollefeys, Genc, Y.: GPU-based Video Feature Tracking and Matching. In: Workshop on Edge Computing Using New Commodity Architectures, Chapel Hill, North Carolina (2006)

[3] Heymann, S., Muller, K., Smolic, A., Froehlich, B., Wiegand, T.: SIFT Implementation and Optimization for General-Purpose GPU. In: International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG), Plzen, Czech Republic (2007)

[4] Warn, S., Emeneker, W., Cothren, J., Apon, A..: Accelerating SIFT on Parallel Architectures. In: IEEE International Conference on Cluster Computing and Workshops, New Orleans, LA (2009)

[5] Huang, Y., Liu, J., Tu, M., Li, S., Deng, J.: Research on CUDA-based SIFT Registration of SAR Image. In: 2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming, , Tianjin (2011)

[6] Yang, Y., Chen, W.: Parallel Algorithm for Moving Foreground Detection in Dynamic Background. In: Fifth International Symposium on Computational Intelligence and Design, Hangzhou (2012)

[7] Brodtkorba, A.R., Hagen, T.R., Satra, M.L.: Graphics processing unit (GPU) programming strategies and trends in GPU computing. Jurnal of Parallel and Distributed Computing, vol. 73, no. 1, pp. 4-13 (2013)

[8] Mohammadi, M.S., Personal Page, http://ce.yazd.ac.ir/rezaeian/Mahdi_SM

[9] NVIDIA, CUDA C Best Practices Guide, NVIDIA, (2013)

[10] Szeliski, R.: Computer Vision: Algorithms and Applications. Springer (2011)

[11] SIFTpp Home, http://www.vlfeat.org/~vedaldi/code/siftpp.html