Towards Affordable Computing: SiftCU a Simple but Elegant GPU-based Implementation of SIFT

Mahdi S. Mohammadi
Electrical and Computer Engineering Department
Yazd University
Yazd, Iran

Mehdi Rezaeian
Electrical and Computer Engineering Department
Yazd University
Yazd, Iran

ABSTRACT

This article presents a fully functional GPU-based implementation of Scale Invariant Feature Transform (SIFT) algorithm. SIFT is a popular image feature extraction algorithm. Although it is a powerful algorithm for image matching but it is also computationally very expensive. This makes it difficult to use especially in real time applications. We purpose to expedite SIFT through GPU-based implementation. There has been some related works on this issue since SIFT was introduced. Our focus is solely on describing GPU-based implementation. We will discuss our implementation in detail. Our implementation is simpler and more efficient than previous works. Part of this paper's purpose is to discuss challenges and strategies related to implementing SIFT like image processing algorithms on GPU. In addition, we are going to present a full comparison between serial implementations of SIFT and our GPU-based implementation, namely siftCU, both in accuracy and time consumption.

General Terms

Image Processing, Parallel Computing

Keywords

CUDA, GPGPU, GPU programming, Image Retrieval, SIFT

1. INTRODUCTION

Researchers, studying image processing, face many different difficulties so they can establish an adequate work. One of the most important and common of these problems is providing computing power. Researchers usually need high amount of computing power to implement and test their designed systems. Furthermore obtaining sufficient hardware to provide acceptable computing power never been cheap. 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 consider using GPU-based programming in their implementations. In this paper we are going to describe detailed GPU-based implementation of SIFT algorithm. This is a good example of GPU-based programming.

Title of this paper mentions affordable computing; affordability of two resources, namely time and money, is our main concern in this paper. 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 (Compute Unified Device Architecture) programming framework at the end of 2007. CUDA is a framework for general-purpose programming on GPUs. Not only it is very efficient and effective but also it is very easy to use. In the last few years, some other schemes also have been introduced for GPU-based programming. Because of different architecture comparing to CPUs, GPUs have greater potential for performing stream processing. Stream processing is a computer-programming paradigm. Simply put, it means emulating parallel processing through SIMD (single instruction multiple data). GPU's performance greatly exceeds CPU's performance when comparing two GPU and CPU that are on same price range. Image processing algorithms are mostly parallel in nature. This is exactly the reason why it is an effective paradigm to use GPU-based programming for implementing image-processing algorithms.

SIFT is a feature extraction algorithm. The features extracted by SIFT 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. The features are highly distinctive, in the sense that a single feature can be correctly matched with high probability against a large database of features from many images [1]. SIFT is a very popular algorithm for image matching. It has been used in object recognition, dynamic object tracking, image retrieval and many other fields. There are at least three reasons why SIFT is a very good example for promoting GPU-based programming: First, SIFT is highly used algorithm; furthermore, it is computationally expensive; lastly, it is very complicated. Our concentration will be demonstrating challenges to implement SIFT in different steps. Researchers can use similar approaches to those described here for implementing their own algorithm. We use CUDA framework, which is a well-documented and highly supported schema for GPU-based programming.

Rest of the paper is arranged as follow: first, we are going to review related works in section Related Works. Then the next section describes SIFT algorithm. Section GPU programming discusses GPGPU and more specifically CUDA programming framework briefly. Our GPU-based implementation of SIFT (siftCU) is discussed in section siftCU. Results for accuracy and speed up compression between SIFTpp and siftCU are presented at section comparison and results. The next section concludes the paper.

2. RELATED WORKS

There has been attempts to implement SIFT on GPU since 2006. Although results reported by these works are satisfactory but most of them did not just focused on this matter and beside from GPU-based implementation, those works also discussed other issues. Sinh et al. presented "GPU-SIFT" in early 2006 [2]. This was before CUDA or any other popular GPU-based programming framework release. In their work, they used more traditional GPU-based programming interface; openGL/Cg. Using a Nvidia Geforce 7900GTX, they reported a 10X speedup over a CPU implementation. Heymann and his colleges presented the next work. Like the first one, they also used traditional GPU-programming [3]. Heymann and his colleges reported they reached 20 frames/sec processing speed without specifying their hardware configuration. After these works, researchers started using new introduced GPU-based programming framework: CUDA. In 2009, Warn et al. used CUDA for running SIFT on NVIDA's GPUs [4]. In their implementation, only parts of the algorithm that related to production of difference of Gaussian (DOG) space run on GPU. For DOG space production part of the SIFT, They reported 13X speed up for their implementation running on a NVIDA FX 5800 GPU. They also claimed overall SIFT running time for GPU-based implementation had only 1.9X speed up over CPU-based implementation. In 2011, Huang and his associates used a CUDA-based SIFT for registration of SAR images [5]. In addition to SIFT, they have also implemented Synthetic Aperture Radar (SAR) image features registration with CUDA. They have test their implementation on two image set with a powerful GPU namely C2050. Huang and his associates, had reported that CUDA-based implementation of entire process including both SIFT feature extraction and feature registration had a 19.6X speed up over CPU-based implementation. Yamazaki et al. have presented an improved SIFT by changing DOG filter to bilateral filter [6]. According to their experiment results, they claimed that this change improved SIFT's features matching precision by a factor of 3 but with an increase in return time by factor of 8. For making their algorithm usable, they had implemented it on GPU and speed it up by a factor of almost 7 making its time performance comparable by serial CPU-based original SIFT. Recently, Yang and Chen used GPU-based SIFT, implemented on CUDA, for moving foreground detection in dynamic background [7]. Using Nvidia Geforce 9800GT, which is a low end GPU, they still managed to gain a 1.3X speed up.

3. SIFT ALGORITHM

Scale invariant feature transform (SIFT) is an image feature extraction algorithm. Lowe introduced SIFT in his paper at 2004 [1]. It has gain great popularity among image processing researchers for using in different form of image matching and object recognition. The popularity of SIFT is due the fact that the features extracted by this algorithm are invariant to many variables including scale and rotation. As described by Lowe, SIFT consist of four major stage [1]:

1. Scale-space extrema detection
2. Keypoint localization
3. Orientation assignment
4. Keypoint description

Next, we briefly describe these stages.

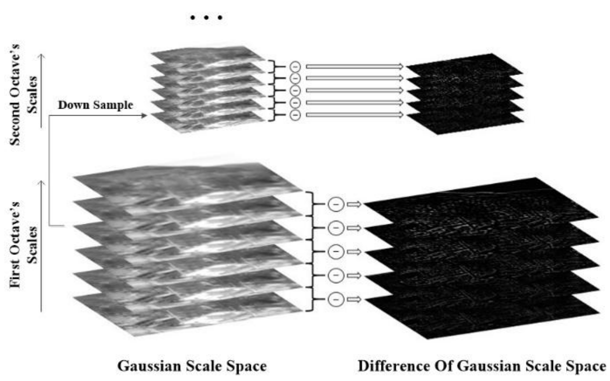

Figure 1: DOG images construction Process. Left: Scale space of Gaussian. Right: DOG images

3.1 Scale-space extrema detection

The first stage of the algorithm detects 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. According to Lowe the scale space of an image I(x, y), is produced from the convolution of a variable-scale Gaussian, G(x, y, σ). Lowe proposed to efficiently detect stable keypoints; we can use scale-space extrema in the difference-of-Gaussian function convolved with image, D(x,y,σ) [1]. Figure 1 shows how this process works. Scale space comprises both Gaussian scale space and spatial space domain. In each octave where every image has the same spatial size, to produce each scale, initial image is repeatedly convolved with a variable Gaussian mask that has an incremental sigma. Difference rate of each scale, k, can be calculated from number of scales per octave namely, S, and it is: k = 2^1/S.

3.2 Keypoint Localization

Every extrema found in 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 (which makes them sensitive to noise), and one for rejecting points that poorly localized along edges. First measure is determined by a Taylor expansion of the scale-space function, D(x, y, σ), the functions value at extremum, $D(\hat{x})$, should be used to reject points.

3.3 Orientation assignment

Third step in SIFT algorithm is to assign orientation to each keypoint. After assigning a consistent orientation to each

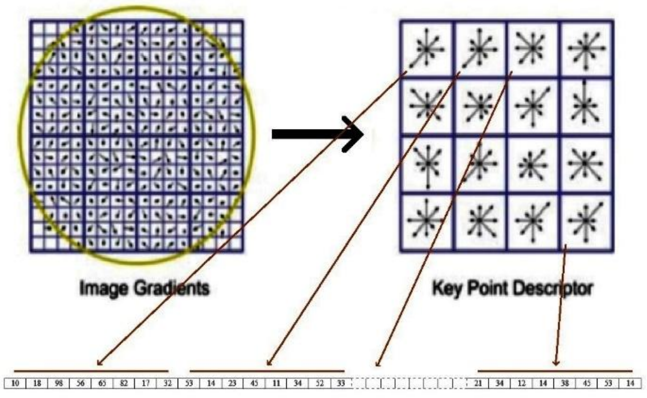

Figure 2: Keypoint descriptor construction

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. Orientation is computed from a smoothed image. Gaussian smoothed image, L, with closest scale to the scale of the keypoint is selected; so that all computations are performed in a scale-invariant manner [1].

3.4 Keypoint descriptor

Last step in SIFT algorithm is to create descriptors for keypoints which were found in previous steps. You can see the feature vector computing process in Figure 2. 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. A single feature vector's grid is presented on the right side of Figure 2. The feature vector is a vector containing the values of all the orientation histogram entries; you can see this vector construction at the bottom of Figure 2. Lowe originally used 4×4×8 = 128 element feature vector for each keypoint[1].

4. GPU PROGRAMMING

4.1 GPGPU

GPGPU stands for General-Purpose computation on Graphics Processing Units, also known as GPU Computing [8]. Graphics Processing Unit (GPU) is processing unit of a graphic card. Traditionally developers used these processing units exclusively for basic computer graphic application. When their great capability for stream processing were discovered; developers started to use them for verity of applications. At early years, programming on GPUs were hard and complicated. In the last six years or so this has been changed due to introduction of programming frameworks like CUDA and openCL. These frame works make GPU-based programming easier and more accessible for everyone.

The multicore CPU is composed of a handful of complex cores with large caches. The cores are optimized for single-threaded performance and can handle up to two hardware threads per core using hyper-threading. In contrast, a GPU is composed of hundreds of simpler cores that can handle thousands of concurrent hardware threads. GPUs are designed to maximize floating-point throughput [9]. Figure 3 depicts architectural difference between CPU and GPU.

Harris first coined out the term GPGPU in 2002 when he recognized an early trend of using GPUs for non-graphics applications [8]. Nevertheless, researchers did not realize its true potential until 2007 when Nvidia Corporation released CUDA programming framework for easy and efficient programming on its GPUs using C++ programming language.

4.2 CUDA

Compute Unified Device Architecture (CUDA) is a programming framework for writing programs to run on GPUs [11]. 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 the host machine and run on CPU. The second part of CUDA programs is the part that will run on GPU and it is consist of some kernels. Kernels are similar to normal functions but run on GPU instead of CPU.

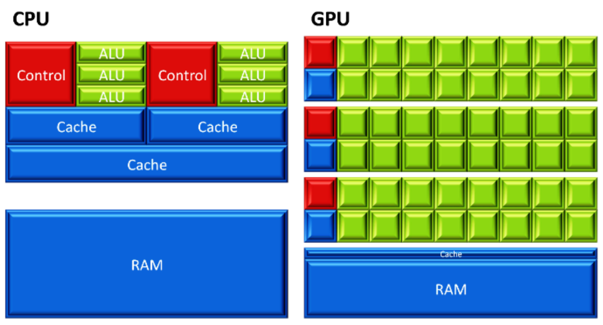

Figure 3: Left: CPU architecture. Right: GPU architecture

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. Threads within the same block can synchronize and collaborate by means of fast-shared memory. The grid and block dimensions can be one, two, or three dimensional, and they determine the number of threads that will be used. 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 [9].

There are three type of memory available for programmers in CUDA. In a decreasing order for access speed: registers, shared memory, and global memory. Registers are fastest memory available on GPU. They are local for each thread. 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 called global memory.

5. 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. To do this we approximated human's vision perception of color. Our implementation codes and the pictures used for analyzing are freely available here [12]. We try to execute as much of code as possible on device when writing programmers for GPU. There are three reasons for this: First, transferring data between host memory, namely RAM, and device memory is time consuming so abundant data transfer can impair performance [13]; secondly, we want to harness GPU's high performance computation power as much as possible; lastly, we do not want to occupy CPU so it can be used for other services. You can see general execution flow of our implementation inside the Host (CPU and RAM memory) and the Device (GPU and device memory) in figure 4. 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 stage of implementation in detail, but first we will present some basic important point about siftCU and CUDA.

5.1 Important Considerations

There are some important recommendations about CUDA programming framework:

- Like original C++ programming, it is much optimal to access consecutive elements in rows than accessing consecutive elements in columns when working with two-dimensional arrays.

- Each thread block can share small amount of very fast access memory called Shared Memory. We should try our best to utilize this facility as much as possible.
- CUDA has optimal memory access for amount of data that are factor of 2, 4, 8, 16, 32 or 64 [12]. When working with two or higher dimensional arrays, to optimize memory access, width of arrays should be multiply of biggest number in listed numbers. If original width does not satisfy this condition then we add padding data to make it so.

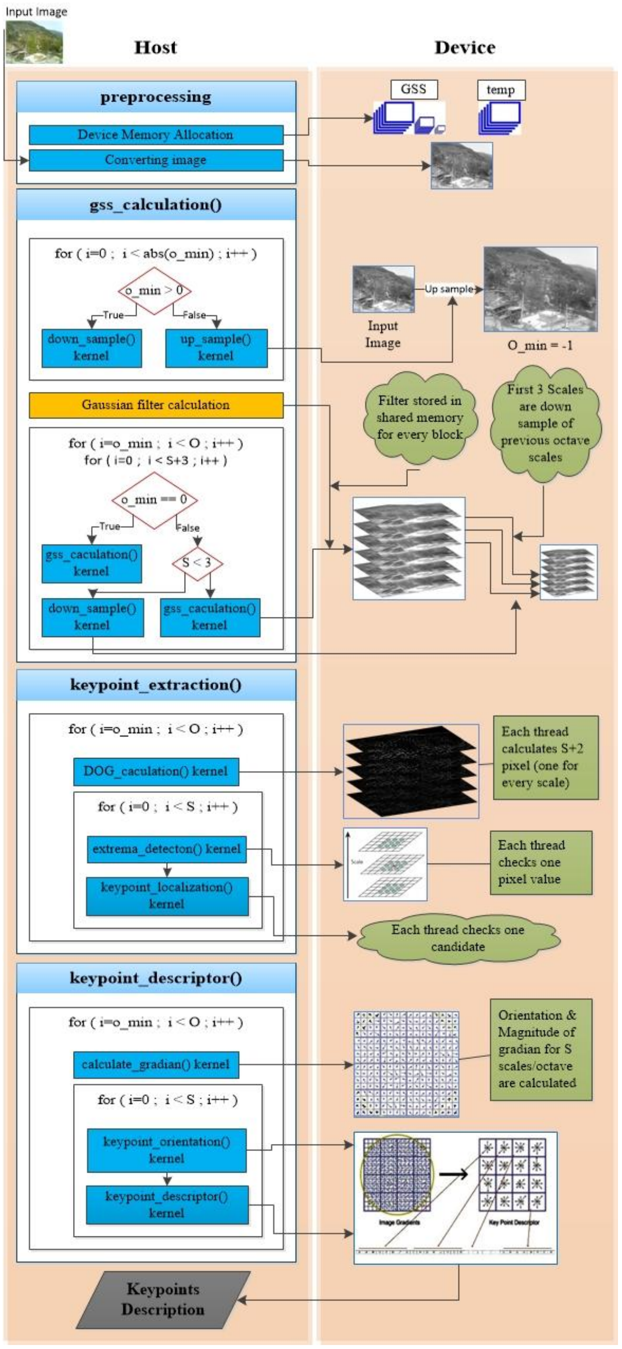
Input Image
Host
Device
preprocessing
Device Memory Allocation
Converting image
GSS
temp
gss_calculation()
for (i=0 ; i < abs(o_min) ; i++)
o_min > 0
True
False
down_sample() kernel
up_sample() kernel
Up sample
Input Image
O_min = -1
Gaussian filter calculation
Filter stored in shared memory for every block
First 3 Scales are down sample of previous octave scales
for (i=o_min ; i < O ; i++)
for (i=0 ; i < S+3 ; i++)
o_min == 0
True
False
gss_caculation() kernel
S < 3
down_sample() kernel
gss_caculation() kernel
keypoint_extraction()
for (i=o_min ; i < O ; i++)
DOG_caculation() kernel
Each thread calculates S+2 pixel (one for every scale)
for (i=0 ; i < S ; i++)
extrema_detecton() kernel
keypoint_localization() kernel
Each thread checks one pixel value
Each thread checks one candidate
keypoint_descriptor()
for (i=o_min ; i < O ; i++)
calculate_gradian() kernel
Orientation & Magnitude of gradian for S scales/octave are calculated
for (i=0 ; i < S ; i++)
keypoint_orientation() kernel
keypoint_descriptor() kernel
Image Gradients
Key Point Descriptor
Keypoints Description

Figure 4: siftCU execution flow

After converting images to 256 level grayscale, we map pixel values to floating point range [0,1]. This way, we could have good precision when creating DOG images. We then move image to device memory. We store all images including original image, GSS images, and DOG images in a way so it confirms third suggestion about CUDA programming.

5.2 GSS Construction

In first step, we need to construct Gaussian scale space. We should apply Gaussian filter to input image recursively at the right spatial space with right scale (sigma). There are two options for this: applying a single 2-dimensional Gaussian filter, or applying two 1-dimensional Gaussian filter, one vertically and one horizontally. Computationally specking second option is more efficient than former [14]. You can see filter operation process in figure 5.

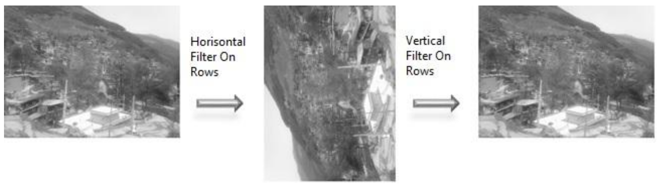

Figure 5: To construct a GSS image, we first apply Gaussian filter on rows of pervious scale (Left image) while storing it in a transpose manner (Middle image), then apply same filter on rows to obtain next GSS image (Right image)

As you can see in figure 5, 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 again to obtain next GSS image (Right image). The reason for storing medial results in a transpose form is to optimize memory access. In addition, this way we get to use the same kernel for both horizontal and vertical filter. This contrast [5] approach which uses two distinct kernels for this process.

In most cases, the convenient approach to use shared memory for a 2-dimintional array is to load a squared area. Nevertheless, programmers should always use shared memory in a manner that best suits performance of their code. That is why in our code, each block loads specified number of rows from sample image to shared memory. After that, each thread applies Gaussian filter to small number of pixels. Number of loaded rows depends on image's width and amount of shared memory dedicated to one block of threads. We have chosen to dedicate 3200 floating point (3200*4 Byte) of shared memory for each block. This number obtained experimentally. It is optimize for all CUDA devices except for version 3.5, which have higher amount of shared memory. For example if image width is 256 then 12 rows, (floor (3200/256)), will be loaded to shared memory.

Beside from rows, we also load Gaussian filters into shared memory. In total S+3 distinguished filters are needed. These filters are precomputed in host. Considering, it is not computationally expensive, it is better not to compute them inside device. We will end up repeatedly calculating them for each block if we compute them inside device code.

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. The siftCU's execution flow in figure 4 demonstrates this process. 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.

5.3 Extrema Detection and Keypoint Localization

In order to optimize memory access by using data localization, we operate this stage independently for each octave. First, we compute DOG images for an entire octave, then search for extrema in each scale of that octave.

After construction of DOG images, a kernel will check whether a pixel value is extrema or not. The kernel stores pixel location and scale if it is an extrema. Then another kernel, running one thread per each detected extrema, extracts keypoints from them. The kernel checks extrema with contrast threshold (equation 6) and edge response threshold (equation 10). If it is suitable, we choose it as keypoint.

There is an important fact about GPU-based programming that we should always keep in mind. GPU's computation power is distributed among several small processing units unlike CPU that have small number of physical processing units. Therefore, it is recommended that programmers divide their algorithm into smaller parts. In other word:

- ✓ Each kernel code, a single thread job, should be as small as possible, as long as, smaller part does not cause massive data transfer.

Let us demonstrate this fact effect by an example. We implemented extrema detection and keypoint localization in two ways: two separated kernels, and a single kernel. When we test our implementation for speed up the first approach, namely separated kernels, caused entire second stage perform 40% faster than second approach.

When detecting extrema, at very least, we process every pixel of an entire scale in parallel. This could be the case even for an entire octave or all octaves depending on device capabilities, picture size. This parallelism is also exist in keypoint localization. Considering it, 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. Before using a lock variable, it is better to decrease access parallelism without hurting computing parallelism. The code stores extrema array in a way so virtually it has separate parts for every scale. Virtually means, there is one array but we keep a fix list of scales starting index in the array. This approach isolates access parallelism in scale level. Now 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 [13]. 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. In the CUDA programming executing same identical instruction sets, boosts overall performance. Identical instruction sets is only possible in branch free code. In the cases like this stage, it is better not to insist on using shared memory where it can lead to performance decline.

5.4 ORIENTATION ASSIGNMENT & KEYPOINT DECRIPTORS

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.

6. COMPERASION AND RESULTS

In this section, we are going to test our implementation for time consumption and accuracy. The siftCU will be compared with SIFTpp in both speed up and accuracy. SIFTpp is a well-known open source implementation of SIFT algorithm in C++ [15]. It is a serial implementation and considered to be optimized and reliable. Our implementation is inspired by SIFTpp. For accuracy check, in addition to SIFTpp, siftCU feature matching results are also compared with Lowe's executable sift implementation. Before presenting comparison results, this section analyze siftCU for memory usage and computation break down of each stage.

6.1 Memory usage of SIFTCU

Most of siftCU's memory usage includes storing Gaussian scale space in main memory. GSS size is deterministic and it depends on input image size:

$$G \qquad (1)$$

In equation 1, S is number of scales per octave which typically is 3, O is number of octaves, W and H are the input image width and height, the constant coefficient, 4, is size of single precision floating point in typical systems. Beside from GSS, a big temporary space for storing some variables, used throughout the program, is also needed. We allocate temporary memory once at the beginning and use it throughout the program. That is because, memory allocation is a time consuming operation in CUDA framework. Therefore, we should avoid it as much as we can.

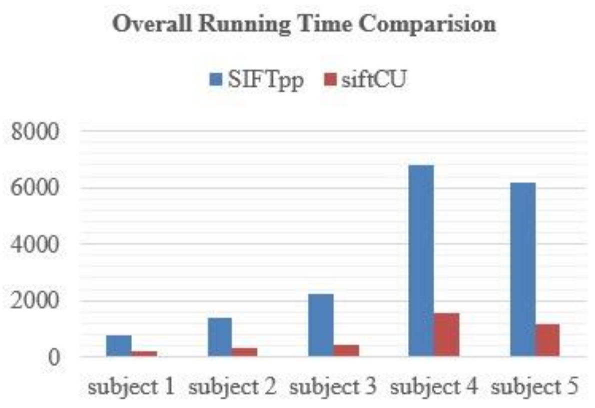

Figure 1: siftCU & SIFTpp overall execution time

6.2 Stges Time Consumption Breakdown

There are three main stages in our implementation. First stage, GSS calculation is a very time consuming process. Unlike the other two stage, GSS calculation only depends on input image size with a direct linear relation. Therefore, as the size of input image grows, this stage time consumption grows too; this can be seen in table 1. Compared to first and third stage, second stage is less inordinate. Unfortunately, because of massive data dependency we were not able to speed up this stage as much as other stages.

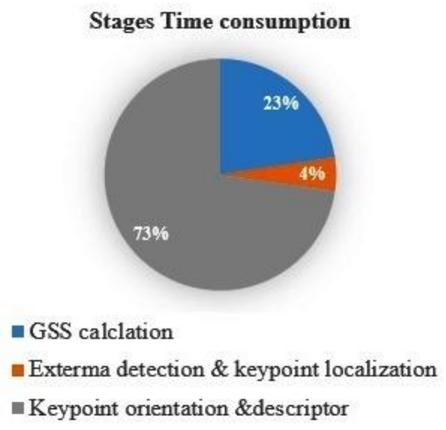

Figure 2: Stages time consumption breakdown

Last stage, keypoint orientation and descriptor, usually is the most time consuming stage. This stage time consumption mostly depends on number of extracted features, which in turn depends on both image's content and size. Figure 6 shows time consumption breakdown of stages for test image 5 in table 1.

6.3 System Set Up

Our test environment setup is as follow:

Host configuration:

- Intel Core i7-2600 @ 3.40GHz
- 4 GB DDR3 RAM

Device configuration:

- Nvidia GeForce GT 440
- 1 GB DDR3 Memory

The processor that we used for testing is one of the most powerful processors for PC computer available in the market. This processor currently has price not less than 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. To test whether our code were able to reach this objective or not, we chose this combination: a cheap affordable GPU versus a powerful but expensive CPU.

Table 1: Time consumption of siftCU compared to SIFTpp. All time scales are in milliseconds

Subject	Stage	SIFTpp	siftCU	Speed up
Subject 1 (386*768)	1	285	57	**5.00**
	2	24	13	**1.85**
	3	481	390	**1.23**
	Total	**790**	**460**	**1.72**
Subject 2 (549*800)	1	437	96	**4.55**
	2	38	18	**2.11**
	3	907	436	**2.08**
	Total	**1382**	**550**	**2.51**
Subject 3 (678*1024)	1	681	128	**5.32**
	2	56	27	**2.07**
	3	1512	641	**2.36**
	Total	**2249**	**796**	**2.83**
Subject 4 (1400*1000)	1	1369	248	**5.52**
	2	151	55	**2.75**
	3	5305	1415	**3.75**
	Total	**6825**	**1718**	**3.97**
Subject 5 (1600*1200)	1	1908	357	**5.34**
	2	162	73	**2.22**
	3	4122	1147	**3.59**
	Total	**6192**	**1577**	**3.93**

6.4 Speed Up Test

You can see test results for speed up in table 1. Results have been broke down for three main stages. Since there is lower data dependency in first stage and also effective shared memory utilization, we were able to achieve a very good speed up. On the contrary, in second stage, GPU-based implementation could not gain much over CPU-based implementation since there are lots of data dependency. Like second stage, there is high data dependency in third stage; we were not able to gain speed up as much as first stage in this stage. Figure7 shows overall time consumption of SIFTpp and siftCU for five images in table 1. As expected, siftCU's speedup over SIFTpp grows as picture size and number of extracted keypoints grow. Although image 4 is actually smaller than image 5 but it takes longer to process. That is because image 4 has much more keypoints compared to image 5.

6.5 Precision Test

For the sake of completeness, we compared our implementation with both SIFTpp and Lowe's his own implementation (which is only available as executable file). 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 all three implementation for each set are not identical. This can be attributed to small differences in the value of some variables and thresholds like size of sampling area around keypoint for orientation assignment or contrast threshold. Nevertheless, as it is obvious, our implementation has not impaired precision of matching. We demonstrated keypoint matching between test sets in figure 8.

7. CONCLUSION

In this paper, a GPU-based implementation of sift feature extraction algorithm was described in detail. We used CUDA, which is a C++ based framework for GPU programming. Other researchers who need to use SIFT algorithm in part of their work can easily utilize our implementation. We have discussed some useful strategies and have given some important recommendations for implementing parallel algorithms in GPU. Using these strategies and recommendations is not limited to implementing SIFT. We compared 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 source spend on providing appropriate hardware for execution. To test whether our code were able to reach this objective or not, 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. Nvidia GTX 670 is as expensive as Intel Core i7-2600. It has 14 times more core than 440 GT, twice the memory, and more than 6 times memory bandwidth. We also showed that siftCU, our implementation, could be as precise as any other implementation of sift. Figure 8 shows descriptor matching results for precision test images in table 2.

8. ACKNOWLEDGEMENT

The authors would like to thank Yazd University's computing & computer center for providing them with proper facilities used in this research.

9. REFERENCES

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

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

[3] S. Heymann, K. Muller, A. Smolic, B. Froehlich and T. Wiegand, "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] S. Warn, W. Emeneker, J. Cothren and A. Apon, "Accelerating SIFT on Parallel Architectures," in IEEE International Conference on Cluster Computing and Workshops, 2009. CLUSTER '09., New Orleans, LA, 2009.

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

[6] T. Yamazaki, T. Fujikawa and J. Katto, "Improving the performance of SIFT using Bilateral Filter and its application to generic object recognition," in 2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Kyoto, 2012.

[7] Y. YANG and W. CHEN, "Parallel Algorithm for Moving Foreground Detection in Dynamic Background," in Fifth International Symposium on Computational Intelligence and Design, Hangzhou, 2012.

[8] M. Harris, "GPGPU," 2013. [Online]. Available: http://www.gpgpu.org.

[9] T. R. H. M. L. S. André R. Brodtkorba, "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.

[10] "openCL Home," KHRONOS GROUP, 2013. [Online]. Available: http://www.khronos.org/opencl/.

[11] "CUDA Home," Nvidia, 2013. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html.

[12] M. S. Mohammadi, "Personal Page," [Online]. Available: http://ce.yazd.ac.ir/rezaeian/Mahdi_SM/.

[13] NVIDIA, "CUDA C Best Practices Guide," NVIDIA, 2013.

[14] R. Szeliski, Computer Vision: Algorithms and Applications, Springer, 2011.

[15] A. Vedaldi, "siftpp," 2006. [Online]. Available: http://www.vlfeat.org/~vedaldi/code/siftpp.html.

Table 2: Feature matching results for siftDemo (Lowe's demo), SIFTpp and siftCU

		siftDemo			SIFTpp			siftCU		
	Images size	Keypoints	**Total match**	Incorrect match	Keypoints	**Total match**	Incorrect match	Keypoints	**Total match**	Incorrect match
Set 1	1200*1600	3575	**216**	2	5455	**115**	1	4926	**182**	1
	386*768	1446			1315			1217		
Set 2	1200*1600	3575	**23**	6	5455	**12**	5	4926	**24**	5
	600*371	1255			1785			1566		
Set 3	1200*1600	3575	**233**	1	5455	**140**	3	4926	**266**	1
	549*800	2718			3199			2514		
Set 4	1600*1200	7632	**793**	6	14061	**1085**	9	11515	**876**	7
	678*1024	2272			4645			4191		

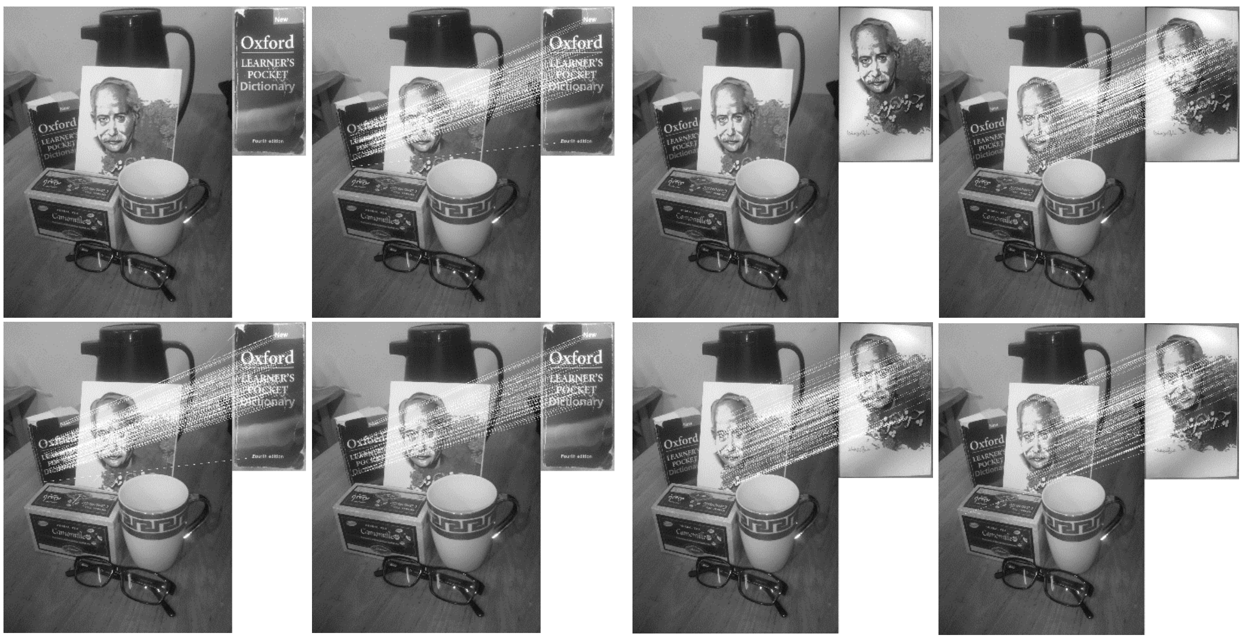

Figure 3: Feature matching for image sets in table 2: Left) set 1; Right) set 3. Images are ordered as Follow in each set: Top left) Images without matching; Top right) siftCU; Bottom left) siftDemo; Bottom right) SITFpp