Exhausting Resources with CPU/GPU Hybrid Distributed Systems: SiftD A Distributed System for SIFT

Mahdi S. Mohammadi, Mehdi Rezaeian

Abstract— Nowadays, it is impossible to manage large scale computer applications with limited resources of a single system. Consequently, distributed and parallel architectures have become an unavoidable element to provid virtually unlimited resources in such applications. SIFT is a popular image feature extraction algorithm, which is used in numerous image processing applications for image matching. Unfortunately, its usability has been undermined due to extreme time and memory complexity. This paper presents siftD, a distributed application developed for distributing and parallelizing SIFT algorithm. Using simple master-slave architecture, it can exhaust all computing and memory resources in a network of computers. SiftD's runtime distributes tasks across nodes in the network. Inside each node, the algorithm is parallelized and run using all available resources. Not only siftD can fully utilize multi-core processors, but also, it can use Graphical Processing Units (GPUs) to run SIFT in parallel. GPUs are designed for running algorithms in parallel and their peak floating point operation exceeds CPU's. SiftD was tested with different hardware configurations. Results showed it has a great potential for utilizing various hardware architecture. Its performance utilization generally exceeds 93%, which is fairly appropriate. Furthermore, siftD can handle computing gigantic pictures in any sizes.

Index Terms— distributed systems, distributed computing, distributed implementation, feature extraction, GPU programming; parallel processing, SIFT

1 INTRODUCTION

Computer technology has made a great impact on all branches of science and technology. It provides hardware and software resources that can be used to process raw data, extract meaningful information, and store all data for future use. This privilege accelerates scientific and technological progression. As science and technology advance, the requirement for computing and storage resources grows exponentially. Depending on scale, these days nearly every industrial and scientific application needs considerable amount of computing resources. Extracting information from gigantic pictures, streaming high quality video in cloud gaming, searching inside enormous databases in search engines are all examples of such applications. Evidently, traditional system architectures with a multi-core processor and limited amount of memory can not satisfy current needs. Accordingly, researchers have turned to parallel and distributed architecture to address this issue.

Distributed systems have been used in many different computer applications in the last four decades. A distributed system is a system, which its hardware and software components are located in the networked computers and only communicate through message passing [1]. Some applications like World Wide Web (www) are distributed in essence, therefore, they should be implemented using distributed applications. On the other hand, there are many centralized applications that deal with giant data sets and need lots of resources to operate. In these applications, distributed systems are used to meet required demand for computing and storage resources. Parallel systems can also provide huge computing resources. There is no clear distinction between parallel and distributed systems. Compared to distributed systems, in parallel systems components are tightly coupled, they may have synchronized clock,

shared memory, and fast means of communication [2]. Frequently, numbers of parallel systems are linked to form a massive distributed system. Designing, implementing, and maintaining distributed systems are usually less costly and more manageable than creating a single powerful pure parallel system.

Image matching using SIFT algorithm is a good example of a resource demanding application. SIFT is a popular image feature extraction algorithm [3]. SIFT has been used in object recognition, dynamic object tracking, image retrieval and various other fields. While it is a prominent algorithm, it has high time and memory complexity. It could take more than five seconds to extract SIFT features from a normal size image using only single core of a high end CPU [4]. This high time complexity is unacceptable in applications, like image retrieval in search engines, which processing thousands of pictures in matters of seconds is customary. Memory consumption of the algorithm is also extreme. Processing Giga pixel images is common in aerial, satellite, or deep space images processing. As we explain later in section XX, operating SIFT on such large images would need more than 400 GB of main memory. Obviously, a serial implementation of SIFT cannot be used in this type of applications. Utilizing distributed and parallel architectures is a clever choice to make SIFT practical for real world applications.

In this paper, we discuss architecture and implementation of siftD, which is a distributed system, designed and implemented, for distributing and parallelizing SIFT algorithm. Although there are general purpose distributed frameworks, which can be used to distribute various algorithms, we preferred to implement siftD from scratch. The main reason is that normally general purpose distributed frameworks are best suited for processing text based data. Compared to text, images should be handled more delicately when they are being processed in parallel and distributed applications. Texts are comprised of small units named words, consequently, they could be easily partitioned for distributed and parallel processing purposes. On the other hand, images are much bigger than words and usually we prefer to process them indiscriminately. Computing SIFT features from parts of an image instead of entire image would decrease accuracy and increase processing time [4]. This topic will be further discussed in section 4.1.

SiftD can utilize all available resources in a network of computers to get as much computing and memory resources as possible. Its runtime dynamically distributes tasks among nodes. It can use as many computer stations as available with its simple master-slave architecture. Inside each node, the algorithm is parallelized and run using all accessible resources. Contemporary networks can consist of personal and server computers. Server platforms could be equipped with more than one processor. These processors connect together to provide a bigger processing unit. Typically each CPU comprises of more than one processing core. SiftD can utilize all the available processors and their processing cores in a station. In addition to CPUs, siftD can fully utilize all available Graphical Processing Units (GPUs) in the system. GPUs are pro-

cessing units of graphic cards. They are designed to maximize floating-point operation and their peak floating point operation exceeded CPU's [5]. Because of their architecture, GPUs can very well run algorithms in parallel. Hence, running SIFT in parallel using them is quite desirable.

Rest of the paper is arranged as follow: Section II briefly discusses related works to this paper. The next section discusses motivations and backgrounds that are related to paper's context, they include SIFT algorithm, its time and memory complexity, and brief introduction about distributed systems. Section IV discusses SiftD's architecture and implementation in details. Section V presents results from evaluating siftD's different aspects. Finally, section VI concludes the paper.

2 REALTED WORKS

The most well-known implementation of SIFT is SIFTpp [6]. It is an optimized serial implementation of the algorithm. No matter how optimized a serial implementations would be, it cannot be used in real world applications because SIFT time and memory complexity is too high. That is why they are many parallel implementations of the algorithm. Two different types of parallel implementations of algorithm exist: multicore implementations and GPU based implementations.

Multicore implementations generally follow one of two existing approaches to parallelize the algorithm. The first approach is to partition input picture into smaller parts and run entire algorithm on each part independently using multiple threads [7][8][9]. This approach has two advantages. First of all, it is easy to implement. More importantly it can easily be used to process gigantic images on cluster of computers [8]. But accuracy of the algorithm would be degraded due to data loss in borders. Original accuracy can be preserved by concatenating extra data to each partition, which will result in productivity loss. The second approach includes parallelizing algorithm in thread level [10][11][12]. In every step of the algorithm, multiple threads of execution run same code concurrently on a portion of the input image. Unlike first approach, this approach do not need extra data to maintain accuracy, whereas, it is harder to implement.

The second type of parallel implementations of the algorithm is GPU based implementation. GPUs exhibit great performance for parallel implementations, specifically for algorithms that can be easily parallelized. Most of the SIFT algorithm can be efficiently parallelized on GPUs [4]. Two GPU based implementations of the algorithm were presented at the early years of its introduction [13][14]. These implementations use traditional low level programming interfaces. GPU programming using low level interfaces like openGL is complex and time consuming, therefore, it could result in less efficient codes. By means of high level programming frameworks, GPU programming is basically like multi-thread programing. After introduction of CUDA, which is an imminent high level programming framework for GPU, researchers used it to implement SIFT [4][12][15][16][17]. Performance results reported for these implementations are very promising and prove that GPU based implementations could expedite SIFT execution greatly. Although there are many parallel implementa-

tions of the algorithm, there is no known distributed implementation to our knowledge.

3 Motivations and Background

We need to distribute and parallelize SIFT because of excessive time and memory consumptions. In this section, we first introduce SIFT algorithm. Next, SIFT's time and memory complexities will be discussed. Also, a brief introduction to distributed systems is presented at the end of the section.

3.1 SIFT

Scale invariant feature transform (SIFT) is an image feature extraction algorithm [3]. It has gain great popularity among image processing researchers for image matching and object recognition. The popularity of SIFT is due to the fact that the features extracted by this algorithm are invariant to many variables including scale and rotation. SIFT consist of four major stages:

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. The scale space of an image I(x, y), is produced from the convolution of a variable-scale Gaussian function, G(x, y, σ), with input image. Scale-space extrema in the difference-of-Gaussian function convolved with image, D(x,y,σ), are used to efficiently detect stable keypoints.
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 are poorly localized along edges.
3) *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. 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.
4) *Keypoint description:* Last step involves creating descriptors for each keypoints, which was 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 gradient orientations are rotated based on the keypoint orientation, consequently, descriptors are invariant to changes in orientation.

3.2 Time Complexity

We cannot easily calculate SIFT's time complexity. Except for the first stage, where time complexity depends on size of the image, in other three stages, time complexity depends on results from previous stages. Table 1 shows SIFT's processing time for 100 different pictures in three categories. These times are recorded running SIFT++ on a system with Intel Core i-7 2600 processor and 4 GB of DD3 main memory. Suppose we want to use this system for real time object tracking. If we consider the video feed a moderate 480p quality, the pictures' size are usually 480×640. This picture size fits to the first category in Table 1. Consequently, we need more than 20x speed up over this system, so the algorithm can be used in such application. This example demonstrates that SIFT's time complexity is so high that it cannot be used in many of its applications using a serial implementation. Therefore, we need to decrease computation time greatly. One way to do this can be through tampering with algorithm itself [18][19][20]. But this approach can decrease precision and matching ability of features, more importantly, it would not solve the problem in very large scale application. A more promising and flexible approach is to utilize parallel and distributed architecture, which is the purpose of this paper.

TABLE 1

SIFT'S PROCESSING TIME FOR 100 DIFFERENT PICTURES IN THREE CATEGORIES

category	Picture size in pixel	count	Time
small	Less than 0.5 Mega pixel	35	74.5
Medium	0.5 - 1 Mega pixel	35	218.3
Big	1 – 2 Mega pixel	30	375.6
Total	---	100	668

3.3 Memory Complexity

Most of the memory usage of SIFT is spent storing Gaussian Scale Space (GSS), and orientation and magnitude of gradient. The size of GSS could be computed using (1).

$$GSS = sizeof(float) * (S + 3) * \sum_{i=o_{min}}^{O-o_{min}} \frac{W * H}{2^{2i}} \qquad (1)$$

In this equation: S is number of scales per octave, O is number of octaves, omin is index of the first octave, W and H are width and height of the input picture. Amount of memory required to store orientation and magnitude of gradient can be calculated using (2).

$$Gradian = 2 * sizeof(float) * S * \frac{W * H}{2^{2o_{min}}} \qquad (2)$$

Using single precision floating point is quite adequate for computing SIFT features. In most systems, single precision floating point is 4 Byte. SIFT operates on gray scale images in which one byte is stored for each pixel. Taking these facts into account and using Equations 1 and 2, we could say SIFT's memory complexity is almost 384 times the size of input image in Bytes. This is quite extreme memory complexity for an algorithm. This means to execute SIFT on a one Giga pixel image, we need about 400 Giga Byte of main memory. Processing images of this size is typical in some applications like

deep space image studies. Such enormous memory could not be found on normal computer systems. On the hand, parallel and distributed systems could be used to provide virtually unlimited amount of memory. Passing through memory limitations to be able to compute SIFT on gigantic images, in addition to time limitations, was another motivation to create siftD. Memory usage of the algorithm could be decreased by repeating some parts of the algorithm instead of storing results of those computations. Clearly this is not a good idea since it further increases time complexity.

3.4 Distributed Systems

A distributed system can be defined as a system in which hardware or software components are located at networked computers and communicate and coordinate their actions only by passing messages. Distributed systems have a wide range of applications. They could provide us with collective resources of many independent computer systems, which can be used for storing, analyzing and processing large quantities of scientific or industrial data. Challenges of designing and implementing distributed systems can be classified in eight areas [1]: 1) Heterogeneity; 2) Openness; 3) Security; 4) Scalability; 5) Failure handling; 6) Concurrency; 7) Transparency; 8) Quality of service. Depending on nature of our application, some of these challenges might be more important than others. We will discuss important challenges of implementing siftD while introducing it throughout section 4.

Comprehensive discussions of designing and implementing distributed systems is a wide area, which we do not wish to go into it in this paper. Readers can refer to reference books for more information [1][2]. Nonetheless, a briefe introduction of general purpose distributed frameworks can be useful, especially since we need it for later discussions. General purpose distributed frameworks are designed to minimize development effort on programmers part for distributing diverse range of algorithms. These systems usually provide programmers with high-level routines to write their programs in parallel manner; completely hiding issues concerning confusing details of parallelization, fault-tolerance, data distribution and load balancing. Recently developed general purpose distributed frameworks are mostly designed based on a programming model called MapReduce.

MapReduce is a programming model for processing and producing large data sets [23]. The general idea of MapReduce is as follow: user specifies a Map function that processes key/value pairs to generate a set of intermediate key/value pairs, and then an optional user defined Reduce function merges all intermediate values associated with the same intermediate key. Many real world applications can be represented in this key/value format [23]. Codes developed using implementation of MapReduce should be automatically parallelized and executed on a large cluster of nodes and distributing tasks should be completely transparent for programmer. Programmers without any experience with parallel and distributed systems should be able to easily utilize the resources of a large distributed system using these frameworks.

The most widely used and mature implementation of MapReduce is considered to be Hadoop [22]. The Apache Ha-

doop software library uses MapReduce programming model to distribute processing of large data sets across clusters of nodes. It is mostly implemented in Java, consequently, it is highly portable. Beside from computing module it includes other modules, like Hadoop Distributed File System, to handle all distributing related task. There are other programming frameworks, which more or less follow MapReduce conceptual procedure. Piccolo is a data-centric programming model for writing parallel in-memory applications in data centers [23]. Unlike MapReduce, Piccolo allows computation running on different computing nodes to share distributed mutable state via a key-value table interface. In particular, applications can specify locality policies to exploit the locality of shared state access. There is also Oolong that uses shared partitioned table schema same as Piccolo but it is designed for asynchronous applications; its global shared states do not need synchronization [24].

4 SIFTD

This section discusses design and implementation of siftD. SiftD includes three major components: Masters, Workers, and Distributed File Systems (DFSs). Fig. 1 shows a hypothetical network with N computer stations running siftD. Each station can have heterogeneous hardware and software configuration. They can be configured to run any combination of siftD's components depending on their resources and policies of the network. This schema creates a very flexible system, which results in a high volume of scalability. Scalability was one of the most important trends for designing and implementing siftD.

SIFT algorithm is used in wide range of application including: real-time applications, large scale applications, and others. Three main goals were pursued in developing siftD: first, to obtain an adequate return time for a single image. Secondly, a reasonable response time for processing numerous images. Finally, to have a system that can process gigantic images without size limitations. In real time applications every tasks, which in our case is processing of a single image, has a deadline. If this deadline passes, processing that image is not appreciated anymore. Therefore, processing time of a single image should be smaller than a threshold, in other words, we should have small return time for every task. To reach this goal, siftD utilizes parallel architecture, more specifically multi-core CPUs and GPUs. Parallel implementation can speed up computing time of an image in an independent system. On the other hand, in large scale applications like image retrieval in search engines, response time of a request should be in a reasonable time frame. This means processing of large number of tasks should be possible in a short time period. There are also some applications like deep space image processing in which images as large as one Giga pixel are processed. Amount of memory for processing images that large can be catastrophic. No matter how powerful a single independent computer can be, it cannot provide enough resources for these two categories. Utilizing distributed architectures is best solution in these cases.

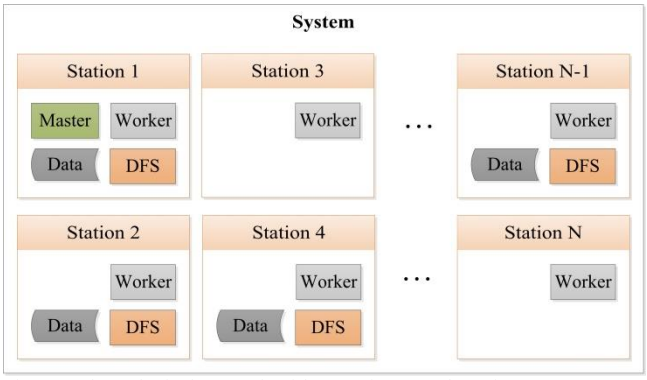

Fig. 1. A hypothetical network with N stations. Each station can run any combination of siftD's components.

4.1 Designing and implementing from scratchs

General purpose distributed frameworks have novel designs, nonetheless, not every application can be optimally distributed using them. As flexible as they could be, they are only suited for a specific class of tasks with special characteristics, most importantly:

- Pertaining large data set should have low data dependency, hence, it can be easily partitioned as little or as big as desired.
- The pertaining algorithm should be simple and parallel in nature. Therefore, it can be simply placed inside a few small functions, like Map and Reduce, and distributed on cluster of nodes.

Most of the text based information retrieval related tasks, like those in search engines, are fit for being distributed by general purpose distributed frameworks, e.g. crawling, indexing, sorting, simple search and other related tasks.

Due to number of characteristics, we prefered to develop a fresh application for distributing SIFT instead of using existing General-purpose frameworks. For starters, SIFT is much more complicated than typical algorithms that are being implemented with general-purpose distributed frameworks. It includes four steps, each one involving lots of independent lengthy computations. This means, SIFT cannot be simply divided into Map and Reduce functions or any other similar concepts. Secondly, SIFT operates on images. Unlike text, images cannot be easily partitioned for processing. It is preferable not to divide them into smaller pieces unless necessary, since, we will end up losing precision and degrading performance. Finally, SIFT includes lots of computations that can be categorized in what we call stream processing [4]. GPUs have better performance for stream processing compared to CPUs. Although there are some limited supports for GPU programming in number of general purpose distributed frameworks including Hadoop but they are not actually optimized for utilizing both CPU and GPU.

There are also numbers of general reasons for developing special distributed application from scratch. Adapting to special hardware properties and utilizing new hardware innovation sometime could be very important. For example consider Remote Direct Memory Access, also called RDMA, which is hardware facility formally found in High Performance cluster

(HPC), but recently also available in 10GB Ethernet [26]. RDMA operations allow a process to read (or write) from a pre-registered memory region of a remote process without involving the CPU on the remote side. Although Mitchell, Geng and Li have designed Pillar [27], which is a general purpose distributed framework with similar design to Piccolo, that partially utilizes RDMA for high performance data reads but this is only one example and there are many other hardware properties and innovations that can be exploited. MapReduce is the dominant schema for designing general purpose distributed frameworks. We expect implementations based on MapReduce programming Model to have highest possible performance for its suitable tasks. This is not necessarily true. Pavlo, Paulson and Rasin showed that Parallel DBMSs can outperform Hadoop even in tasks suitable for MapReduce [28].

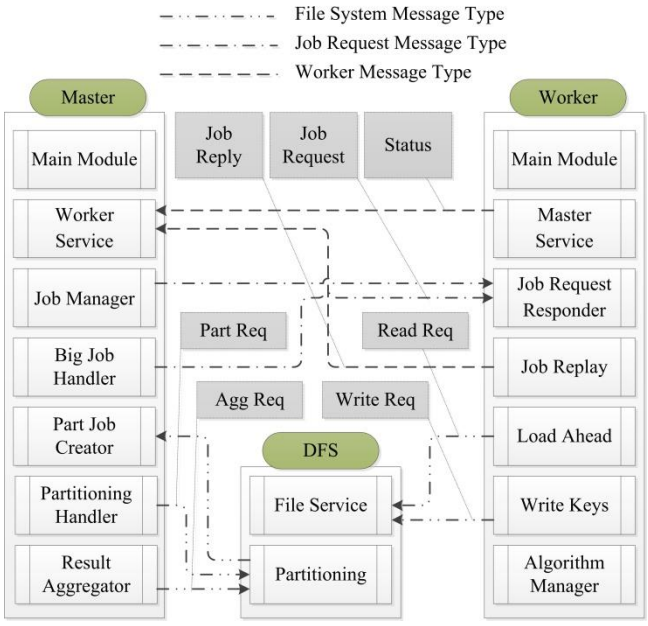

Fig. 2. External communications between major components of siftD.

4.2 Architecture and Operation Flow

SiftD comprises of three main components. There is one Master in the system that distributes tasks across workers. Also, it decides how large pictures should be partitioned and arranges for partitioned results to be aggregated. There could be as many workers as needed, each running inside a station. Workers run algorithm in parallel using multi-core CPUs and GPUs. Usually every system in the network runs an instance of Distributed file system, which handles files exchange and partition/aggregate request from Master. Each component consists of many modules, which will be introduced throughout this section. Fig. 2. shows external communications between major components and those modules that are involved.

Three types of messages are exchanged between components including: worker message, job request message, and file system message. Each message type has more than one application. Worker message type can be used for advertisement purposes, which informs Master about workers status (Status). It also used by job replay module of Worker to inform Master about a task's fate (Job Replay). Job manager and big job manager in Master use Job request messages for requesting a

job from worker (Job Request). File system messages are used for read/write file request (Read Req, Write Req). They are also used for image partitioning request and result aggregate request (Part Req, Agg Req).

Master's internal design and communications between its modules is shown in Fig. 3. In order to avoid complications, there are two job queues: one of them is for regular size images (normal jobs), and the other one is for large images (big jobs) that could not be processed on a single station. Different Modules have their own dedicated pointer for accessing desired queue to operate independently on lists without conflicting with other modules. Names of these pointers are presented in the figure. Each module is implemented in a separate thread. Internal communications between modules is depicted with dotted lines that are accomplished using POSIX's condition variables. Variable's names are mentioned above respective communication line.

Execution flow inside Master is as follow. Initializer uses configuration files to initialize all important variables and two job queues. Every worker has one or two size limit, one for CPUs and one for GPUs. Limits inform Master about biggest images that workers can handle. Therefore Master always has a complete list of workers' size limit. In order to distinguish normal and big jobs, most repeated value (mod) in workers' size limit list has been used. Consequently, those images that are smaller than this quantity are placed in normal job queue and those bigger are placed in Big Job queue. Mod value has been picked since it would create most flexibility for scheduling since there would be more workers able to do one particular job. Both Online Job Scheduler and Initializer use this notion to choose where to put each job. Inside Normal Jobs Queue any entry is a picture that should be processed. Job Manager uses a function, named 'worker finder,' to find a worker for a job. Worker finder function will be discussed in section 4.4. If no free worker exists when looking for a suitable worker, Job Manager will sleep only to be notified when there is a free worker.

Processing big jobs is more complex than normal jobs. First, Partitioning Handler calculates the size of each partition, this is discussed in section 4.4. Then, it sends a partitioning request to systems holding the picture. Part Job Creator gets response messages about images being partitioned. It writes characteristics of big jobs partitions in Big Job Queue then notifies Big Job Handler about the new big job that should be processed. Big Job Handler treats every partition of a big job same as Job Manager treats a normal job. When a big job finishes completely, Result Aggregator gets notified. It arranges for partitioned results to be aggregated by sending aggregation request.

Worker Service receives status and job replay messages from workers. If it gets status message from a new worker that was not in Master's workers list, adds the new worker to the list. It notifies Task Manager when a job replay arrives. It also sends a free worker notification. Upon receiving a job replay signal, Task Manager checks conditions. It sends signals to Job Manager and Big Job Manager about failed jobs, so they would schedule them again. If part of big jobs finishes, it

checks wither an entire big jobs has finished or not. It will notify Result Aggregator when a big job finishes completely.

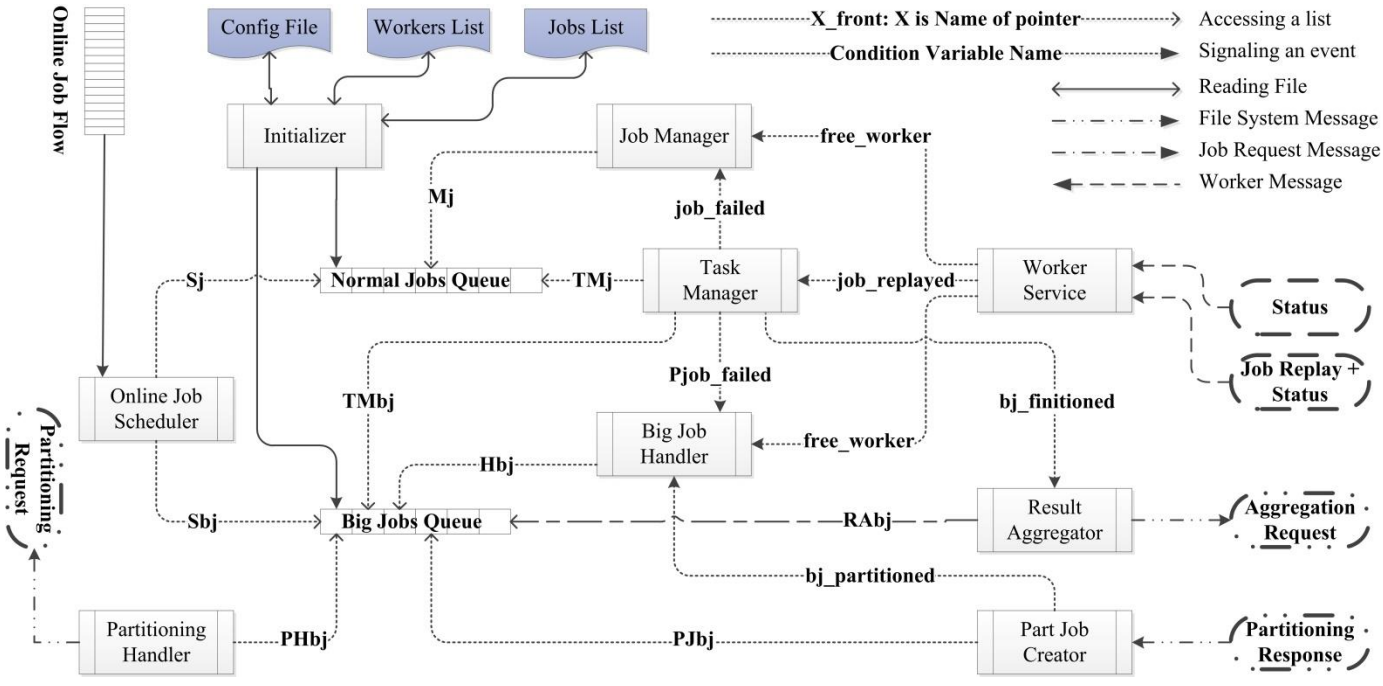

Figure. 3. Internal design of Master and communications between its modules

Worker's internal design is illustrated in Fig. 4. Like Master, Initializer sets all global and important variables using configuration files. Master Service sends advertisement messages about workers status periodically. Job processing procedure starts with Job Request Responder receiving a job request. It places the new job inside the jobs queue, and then signals the Load Ahead that a new job has received. Load Ahead tries to fetch jobs to main memory. If it succeeds, it signals Algorithm Manager that there is new job ready to be processed. It also notifies Job Replay wither new job was fetched successfully or not. This module has been discussed in more details at section 4.4. Algorithm Manager runs SIFT algorithm on requested images in parallel utilizing CPUs and GPUs. When Algorithm Manager extracts a new set of keys, it signals Keys Handler. Wither it succeeds processing an images or not, it signals Load Ahead and Job Replay. Keys Handler moves newly extracted keys to their appropriate destination. Job Replay informs Master about a job status in Worker. First time it does that is when a job is fetched successfully or unsuccessfully, which it gets a notification from Load Ahead. Then if job was fetched without an error and was processed, it informs Master that wither job was successfully carried out or not.

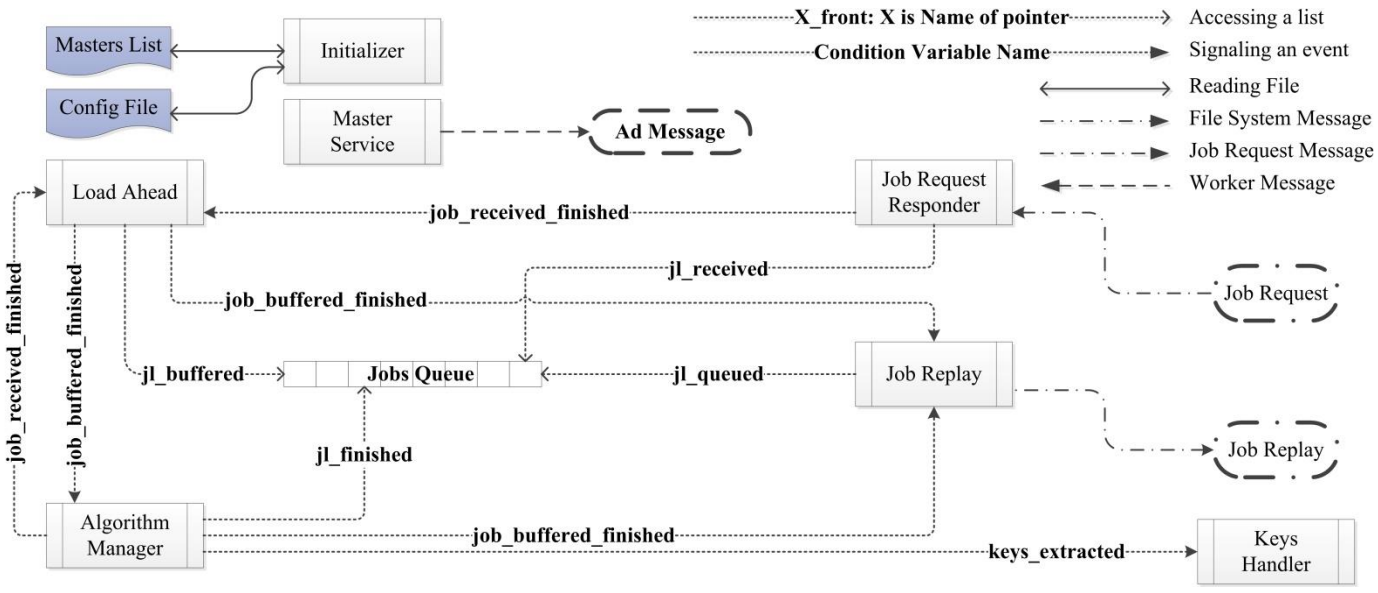

Figure. 4. Internal design of Worker and communications between its modules

File system distributes files across network. It tries to spread files evenly among stations. It also creates extra copies of files according to policies employed by system administrator. A file description structure contains file name and address in the system. Therefore, a job request that contains file description includes file address in the system. In order to handle file transfer in the system, there is a module called Resolver that resides in Worker. This module can actually considered a part of DFS. Any file transfer must go through this module. It keeps records about extra copies of files in the system that DFS informs it about. File transfer requests categories into three types concerning requested file address: Their address is same as requester address; their address is different from requester but there is extra copy in requester station; their address is different from requester and there is no extra copy. First two types are handled locally using fast read/write system interrupts. But third type is carried out using TCP/IP protocol.

4.3 Implementation

SiftD has been implemented entirely in C++ using only POSIX and ANSI C++ standard libraries [29]. Implementation is portable across most of Linux and UNIX platforms. It uses two separate implementations of SIFT algorithm, one for utilizing CPUs and another one for utilizing GPUs. SiftCU is an optimize CUDA based implementation of SIFT algorithm to utilizes Graphical Processing Unites [4]. SiftD uses siftCU to run the algorithm on GPUs. Each major component of the system is implemented in a separate process and each module is a separate thread inside its components. This paradigm would help improve performance since it decreases context switch time and maximizes multi-core architecture utilization. Most of the modules are only a single running thread in the system. But three of Worker's modules including Load Ahead, Algorithm Manager, and Keys Handler have more than one instance, running simultaneously in the system. Those mod-

ules do lengthy computations, consequently, in order to maintain high resource occupancy, we need more than one instance of them. Threads are implemented using POSIX's Pthread library.

External communications between components has been carried out with POSIX sockets. All normal messages including job request and replay, worker status, and file request use UDP protocol as their underling transport layer protocol. On the other hand, TCP protocol has been used for file transfer. POSIX's condition variables have been used for internal communication between modules. Heterogeneity is a big concern when dealing with many threads trying to access to shared memories. Condition variables are excellent choice for inter thread communications, not only because they are fast but also because they are protected with mutex variables. Therefore, any mutual access to shared memories that are related to a condition variable can be avoided. There are two queues in Master and one queue in Worker as discussed in previous subsection. Many modules may access these queues simultaneously. All modules have their dedicated pointers to a queue, but some modules use other module's pointers to differentiate various situations. To avoid any unnecessary race condition, there are specific mutex variables for protecting those queue pointers that are accessed with more than one module.).

4.4 Performance Considerations

This section discusses three important considerations about improving performance in siftD's design and implementation.

4.4.1 In Advance Task Fetching

One of the existing problems in distributed systems is over head of data and message communication in under lying network. Compared to internal data transfer inside a computer system, data communication using network facilities can have a large delay. In order to alleviate negative effects of this problem in siftD, there is a module in Worker that caches data before it is being needed. Load ahead module always fetches number of requested tasks on worker's main memory in advance. This process can almost completely remove extra network transferring overhead. Beside from network delay, another data transfer delay related issue exists in siftD. Right now memory access inside graphic cards is much faster than main memory [7]. But transferring data between main memory and graphic cards main memory would create another overhead. To deal with this concern, Algorithm Manager fetches an extra task for any thread running algorithm on GPU to graphic memory while another task is been running.

4.4.2 Adaptable Worker Selection

Job and Big job manger modules use same function, named worker finder, to find an adequate worker for their tasks. The worker finder function uses three measures for choosing the right worker to guarantee best possible load balancing in the system. They include file's location in the system, size limitation, and worker's business. First priority for choosing right worker for a job is its pertaining file's location. Worker finder tries to appoint a job to a worker that already holds job's file.

As discussed earlier, each worker has one or two size limitations attributed to it. If worker's size limitation is close to file size the best possible performance could be reached. Therefore worker finder tries to choose worker with closest size limitation to a file size. Business is calculated based on workers resources and number of jobs already appointed to the worker. A worker's resources are represented with an integer number, which is calculated based on number processors, GPUs, and their properties. There is a maximum for number of jobs that can be appointed to a single worker that is calculated based on workers business. In order to avoid starvation for bigger jobs, worker finder always reserves 20 percent of "reserved workers" capacity for doing bigger jobs. Reserved workers are the ones that their size limitation is 80 percent of maximum size limitation. When Job Manager and Big Job Manager call worker finder, if all workers would be in full capacity (business), they will sleep until there is at least one worker for appointing jobs.

4.4.3 Appropriate Large Picture Partitioning

Different methods can be used to partition large images for computing SIFT features [7][8] [9]. According to the results presented in previous works and our studies, the best way that it can be done in a distributed system is plaid partitioning. It gives better load sharing while maintaining descriptors' matching accuracy. But more importantly, this method of segmenting has far greater scalability compared to other methods. To maximize scalability, partitions should have equal size and they should be as squared shaped as possible. Number of part for an image is calculated using (3):

$$f = \frac{h * w}{u} \qquad (3)$$

In this equation: 'h' and 'w' are image's height and width, and u is size of one partition (size limitation). Now for calculating number of horizontal and vertical intersections in image, (4) and (5) are used:

$$hf = \sqrt{\frac{h}{w} * f} \qquad (4)$$

$$wf = \sqrt{\frac{w}{h} * f} \qquad (5)$$

But 'hf', and 'wf' are real numbers, which means when they have decimal parts we are force to divide picture into same size squared shaped parts (for absolute part) and rectangular shaped parts (for decimal part). This is not reasonable since it would decrease scalability and accuracy. Instead, we use their closest integral multiply to 'f' for partitioning the image. Considering floor and ceil functions for mapping real to integral numbers, there is four possible integral multiply for two real numbers: 1. ceil(hf)×ceil(wf); 2. ceil(hf)×floor(wf); 3. floor(hf)×ceil(wf); 4. floor(hf)×floor(wf). The integral pairs, which have closest multiply to 'f' value, is chosen for partitioning.

4.5 Failure Handling

Failure handling is an important issue in any computer system especially distributed systems. We have classified failures to three distinct categories: 1) Costumery failures; 2) Problematic failures; 3) Fatal failures.

Costumery failures are those that can easily be predicted and prevented before even happening. For example, occasionally a read/write request to normal or socket file could not be carried out on a single call due to low buffer memory or small Maximum Transfer Unit. These type of failures are addressed with simple precautionary mechanisms, for example read/write request for a file is carried more than once if required until entire file is been transferred.

Second types of failures are simple failures that cannot be predicted or they have been neglected, either way there is no precautionary mechanism to prevent them. These failures would not disturb entire task flow in system, for example file transfer in network can fail for many unpredictable reasons. These failures are get logged in a permanent file for future reference. There are mechanisms in the system that each of them can deal with a group of these failures, for example if a file transfer in network fails, it will get transferred again.

Fatal failures are different from first two types of failures. They cannot be easily dealt with and can have catastrophic effect on entire execution flow in the system. There are three types of fatal failure: 1) Process fatal failure; 2) Node fatal failure; 3) Network fatal failure. Every major component of siftD runs on separate process. They could cease running due to unpredictable events. We can arrange for processes to automatically be restarted after failure through UNIX 'init' process. Master and Workers record all critical data pertaining to queues' status and vital variables occasionally, consequently, they can recover their entire process state when restart after a fatal failure. Distributed file system can revive itself partially. It records those status data that are related to partitioning images but not the data related to file transfer. If DFS fails during a file transfer, it would not transfer file again after reviving itself rather than it waits for a new file transfer request.

In a node failure an entire computer station in the network would stop working temporarily or permanently. Temporary nodes failures, like a system restart, are similar to process failure. On the other hand, when a permanent (or long term) failure happens to a node, its resources cannot be used anymore. Task Manager in Master can detect this type of failure with two mechanisms: 1) Not receiving advertisement messages from particular Worker after a deadline; 2) Sets timer for each job, which expires after a deadline. First mechanism can detect node failure relating to stations running Workers and second one can detect node failure relating to stations running DFSs. When a possible node failure is detected, jobs appointed to worker running in that node are reappointed to other workers by managers. There is no recovery procedure for failure that happens for node running the Master. Network failures are very similar to station failures. But a single network failure could affect more than one node all at once. Nonetheless, their recovery procedures are the exactly like node failure.

5 Evaluations and Results

This section presents tests results for evaluating siftD performance and its capability to utilize different hardware resources. Tests were completed using Ubuntu 12.04 operating system, nevertheless, system has been tested on some other operating systems, like CentOS 6, for portability check.

5.1 Heterogeneity

SiftD was tested using numerous hardware platforms to evaluate its capacity to harness various resources. Images set used in this test are shown in Table. I. Table. II shows all hardware configuration properties used in the test. Notice that configuration No. 3 is actually a network of five computer stations each with a 2 core multi-core processor (hence 2 × 5 core in total). Also configuration No. 6 is a combination of No. 3 and No. 4 (a single station with both CPU and GPU). Similarly No. 7 is a combination of No. 1 and No. 5. Finally, Last configuration is combination of all systems (No. 2, No. 6, and No. 7). All systems were connected using 100Mb Ethernet network connections.

TABLE 2

VARIOUS HARDWARE CONFIGURATIONS FOR HARDWARE UTILIZATION CAPABILITIES EVALUATION

Config	Processor	Number of Cores	Memory
No. 1	Intel core i-7 4700MQ	4	4 GB DDR3
No. 2	Intel core i-7 2600	4	4 GB DDR3
No. 3	NVidia GeForce 440GT	Graphic	1 GB DDR3
No. 4	Intel core i-3 2100 × 5	2 × 5	2×4 GB DDR3
No. 5	NVidia GeForce 750M	Graphic	2 GB DDR5
No. 6	Intel core i-7 2600 & NVidia GeForce 440GT	4 + Graphic	4 GB DDR3 + 1 GB GDR3
No. 7	Intel core i-7 4700MQ & NVidia GeForce 750M	4 + Graphic	4 GB DDR3 + 2 GB GDR5
No. 8	**All (various)**	various	Various

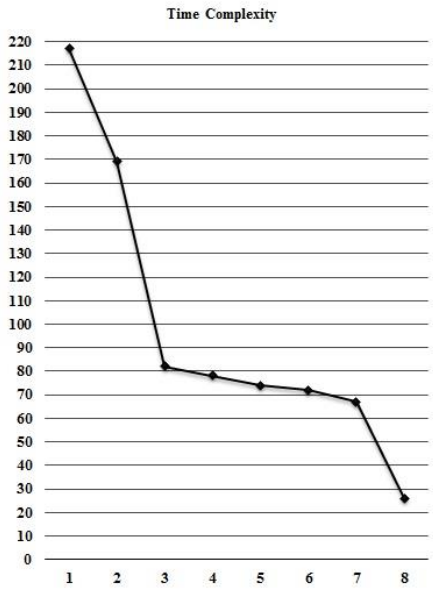

Figure. 5. Return time for handling arbitrary jobs with five hardware configuration in Table II.

Results are shown in Fig. 5. You can see using more computing power results in smaller processing time. It is obvious that system has great capability to utilize different hardware configurations in parallel.

5.2 Productivity

In order to evaluate siftD's productivity, we preferred to use computer stations with identical hardware configuration. Using a network of identical nodes is more meaningful for evaluating productivity and makes assessments simpler. Each node used in this test have same configuration as configuration No. 4 in Table. II, except for in this test only one processing core in each node was used. Table III. shows processing time of image set in Table I. for different number of stations running siftD.

TABLE 3

RETURN TIME FOR HANDLING ARBITRARY JOBS WITH DIFFERENT NUMBER OF COMPUTER NODES.

Number of Stations	1	2	3	4	5
Processing Time	749	401	263	199	158

Equation (1) was used to calculate system's performance, notice that processing time for a single station is considered as base measure:

$$P_n = \frac{t_n}{t_1 \times n} \times 100 \qquad (6)$$

In this equation, Pn and tn respectively are productivity and time complexity of system when using N number of stations; t1 is time complexity for a single station. Figure 6 shows system productivity for using different number of stations. General productivity is higher than 93 percent, which could be considered satisfactory. The slight discrepancy in performance can be attributed to disruptions in load balancing and file transfer in the network. Higher performance could be reached using a better load balancing and faster network connections.

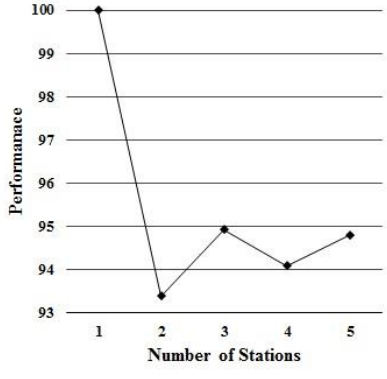

Figure. 6. SiftD's productivity for handling arbitrary jobs with different number of computer nodes.

5.3 Gigantic Pictures

System throughput for handling gigantic picture was evaluted by processing a large picture. The Image used in this test

was a deep space picture with the size of 7000 × 9000 (63 mega pixel), which was derived from the Hubble Telescope website [28]. Table IV shows times recorded for processing this image by incremental number of stations. Fig. 6. depicts system productivity, which are calculated same as before. The results show that siftD has high capability to handle large images with modest productivity. Performance discrepancies in this case are much lower than handling arbitrary jobs. This is due to the fact that load balancing is much better. Since all tasks are parts of a single images and they are all same size, therefore, load balancing naturally becomes more appropriate compared to handling arbitrary jobs.

TABLE. 4

PROCESSING TIME RESULTS WHEN USING DIFFERENT NUMBER OF STATIONS

Number of Stations	1	2	3	4	5
Processing Time	674	356	238	179	143

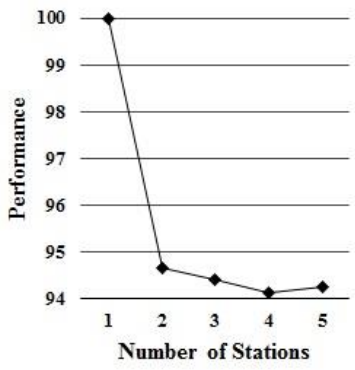

Figure. 7. SiftD's productivity for handling a gigantic picture with different number of computer nodes.

6 CONCLUSION

In this paper, we presented a specialized distributed application for distributing SIFT algorithm. SIFT is a popular image feature extraction algorithm. SiftD is able to utilize both CPUs and GPUs. Its implementation and architecture were discussed in details. Although, there are general purpose distributed frameworks that can be utilize to develop a distributed application for SIFT, but we preferred to implement an application from scratch. General-purpose frameworks are better suited for simple algorithms and associated data sets with limited dependency. On the other hand, for applications involving complex algorithms and high data dependency, it is better to design and implement new applications. We described characteristics of SIFT algorithm and its associated data. SIFT algorithm is not suited to be distributed with general purpose frameworks. We evaluated siftD's productivity and throughput. Test results showed siftD has high capacity for utilizing various hardware architectures. Its productivity is generally higher than 93 percent. We did not use high performance network connections, therefore, this is an adequate performance. We could gain better performance utilizing more sophisticated hardware systems and better load balancing. Furthermore, siftD is able to handle processing giant image appropriately.

ACKNOWLEDGMENTS

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

REFERENCES

[1] D.S. G. Coulouris, J. Dollimore, T. Kindberg and G. Blair, Distributed Systems: Concepts and Design, Fifth Edition, Addison Wesley, 2011.
[2] Sukumar Ghosh: Distributed Systems: An Algorithmic Approach, 2006 CRC Press (ISBN 1584885645)
[3] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," Journal of Computer Vision, vol. 60, no. 2, pp. 91-110, 2004.
[4] M. S. Mohammadi, M. Rezaeian, "Towards Affordable Computing: SiftCU A Simple but Elegant GPU-Based Implementation Of SIFT," International Journal of Computer Applications, vol. 90, no. 7, pp. 30-37, 2014.
[5] A. R. Brodtkorba, T. R. Hagen, M. L. Satera, "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.
[6] A. Vedaldi, "siftpp," 2006. [Online]. Available: http://www.vlfeat.org/~vedaldi/code/siftpp.html.
[7] Guiyuan Jiang, Guiling Zhang and Dakun Zhang, "A Distributed Dynamic Parallel Algorithm for SIFT Feature Extraction," in 3rd International Symposium on Parallel Architectures, Algorithms and Programming (PAAP), pp.381-385, 2010.
[8] Stanislav Bobovych, "Parallelizing Scale Invariant Feature Transform On A Distributed Memory Cluster", inquiry, The University of Arkansas Undergraduate Research Journal, vol. 12, pp. 42-48, 2011.
[9] Mingling Zheng, Zhenlong Song, Ke Xu, Hengzhu Liu, "Parallelization and Optimization of SIFT Feature Extraction on Cluster System." In Conference on World Academy of Science, Engineering and Technology,, pp. 273-277, 2012.
[10] H. Feng, E. Li, Y. Chen, and Y. Zhang, "Parallelization and characterization of sift on multi-core systems," IEEE International Symposium on Workload Characterization (IISWC'08), pp. 14–23, 2008.
[11] Q. Zhang, Y. Chen, Y. Zhang, and Y. Xu, "Sift implementation and optimization for multi-core systems," IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008), pp. 1–8, 2008.
[12] Seth Warn, Wesley Emeneker , Jackson Cothren,Amy Apon, "Accelerating SIFT on Parallel Architectures," in IEEE International Conference on Cluster Computing and Workshops (CLUSTER'09), pp.1-4. 2009.
[13] F. Remondino, "Detectors and descriptors for photogrammetric applications," International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, vol. 36, no. 4, pp. 49-54, 2006.
[14] 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.
[15] 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.
[16] 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.
[17] 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.
[18] K.A. Peker, "Binary SIFT: Fast image retrieval using binary quantized SIFT features," in Content-Based Multimedia Indexing (CBMI), 2011 9th International Workshop on, Madrid, Spain, 2011.
[19] N. Zhen-Sheng, "Binary SIFT: Fast image retrieval using binary quantized SIFT features," in Digital Home (ICDH), 2012 Fourth International Conference on, Guangzhou, China, 2012.
[20] C. Liang-Chi, C. Tian-Sheuan, C. Jiun-Yen, N.Y.C. Chang, "Fast SIFT Design for Real-Time Visual Feature Extraction," Image Processing, IEEE Transactions on , vol. 22, no. 8, pp. 3158-3167, 2013.
[21] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, 2004.
[22] The Apache Software Foundation, "Hadoop Home," Available: http://hadoop.apache.org, Online, 2014.
[23] R. Power and J. Li, "Piccolo: Building Fast, Distributed Programs with Partitioned Tables," in OSDI'10 Proceedings of the 9th USENIX conference on Operating systems design and implementation, Berkeley, CA, USA, 2010.
[24] C. Mitchell, R. Power and J. Li, "Oolong: asynchronous distributed applications made easy," in APSYS 12 Proceedings of the Asia-Pacific Workshop on Systems, Seoul, South Korea, 2012.
[25] C. Mitchell, Y. Geng and J. Li, "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store," in USENIX ATC'13 Proceedings of the 2013 USENIX conference on Annual Technical Conference, Berkeley, CA, USA, 2013.
[26] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in SIGMOD 09: Proceedings of the 35th SIGMOD international conference on Management of data, New York, NY, USA, 2009.
[27] IEEE standards association, "POSIX Home," Available: http://standards.ieee.org/develop/wg/POSIX.html, Online, 2014.
[28] Hubble Site, "Hubble Image Ghallery," [Online]. Available: http://hubblesite.org/newscenter/archive/releases/2014/05/image/b/. [Accessed 2014].