Liu T, Liu Y, Li Q *et al.* SEIP: System for efficient image processing on distributed platform. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 30(6): 1215–1232 Nov. 2015. DOI 10.1007/s11390-015-1595-1

SEIP: System for Efficient Image Processing on Distributed Platform

Tao Liu (刘 弢), Yi Liu (刘 轶), *Member, CCF*, Qin Li (李 钦), Xiang-Rong Wang (王香荣), Fei Gao (高 飞) Yan-Chao Zhu (朱延超), and De-Pei Qian (钱德沛), *Fellow, CCF*

School of Computer Science and Engineering, Beihang University, Beijing 100191, China

E-mail: thomasball@126.com; yi.liu@buaa.edu.cn; {qin.li, xiangrong.wang, fei.gao, yanchao.zhu}@jsi.buaa.edu.cn depeiq@buaa.edu.cn

Received May 15, 2015; revised October 13, 2015.

Abstract Nowadays, there exist numerous images in the Internet, and with the development of cloud computing and big data applications, many of those images need to be processed for different kinds of applications by using specific image processing algorithms. Meanwhile, there already exist many kinds of image processing algorithms and their variations, while new algorithms are still emerging. Consequently, an ongoing problem is how to improve the efficiency of massive image processing and support the integration of existing implementations of image processing algorithms into the systems. This paper proposes a distributed image processing system named SEIP, which is built on Hadoop, and employs extensible in-node architecture to support various kinds of image processing algorithms on distributed platforms with GPU accelerators. The system also uses a pipeline-based framework to accelerate massive image file processing. A demonstration application for image feature extraction is designed. The system is evaluated in a small-scale Hadoop cluster with GPU accelerators, and the experimental results show the usability and efficiency of SEIP.

Keywords big data, distributed system, image processing, GPU, parallel programming framework

1 Introduction

In the era of big data, demands for massive file processing grow rapidly, in which image data occupies considerable proportion, such as pictures embedded in web pages, photos released in social network, pictures of goods in shopping websites, and so on. Commonly, these images need to be processed for different kinds of applications, like content-based image retrieval (CBIR), image annotation and classification, and image content recognition. Due to the volume of images and the complexity of related algorithms, it is necessary to use distributed systems with accelerators (e.g., GPU) to process these massive images. According to [1], there are four advantages of distributed systems over isolated computers: 1) data sharing, which allows many users or machines to access a common database; 2) device sharing, which allows many users or machines to share their devices; 3) communications, that is, all the machines can communicate with each other more easily than isolated computers; 4) flexibility, i.e., a distributed computer can spread the workload over the available machines in an effective way. Compared with the single-node environment, by employing distributed systems, we can obtain increased performance, increased reliability, and increased flexibility[2].

To support efficient data processing in distributed systems, there exist some representative programming models, such as MapReduce[3], Spark[4], Storm①, all of which have open source implementations and are suitable for different application scenarios. The MapReduce framework is considered as an effective way for big data analysis due to its high scalability and the ability of parallel processing of non-structured or semi-structured data[3]. Multitudinous applications are re-

Regular Paper

Special Section on Networking and Distributed Computing for Big Data

The work was supported by the National Natural Science Foundation of China (NSFC) under Grant No. 61133004, the National High Technology Research and Development 863 Program of China under Grant No. 2012AA01A302, and the NSFC Projects of International Cooperation and Exchanges under Grant No. 61361126011.

①Storm. http://storm-project.net/, Oct. 2015.

lying on the MapReduce framework, especially on the open source implementation of MapReduce framework — Hadoop[5], which provides a platform for users to develop and run distributed applications. Spark[4] is developed by UC Berkeley, which is a kind of in-memory computing parallel framework and suitable for iterative computations, such as machine learning and data mining. RDDs (resilient distributed datasets) that can be persisted in memory across computing nodes[6] are utilized by Spark. Storm② is an open source distributed real-time computation system, which makes it easy to reliably process unbounded streams of data, and does for real-time processing what Hadoop does for batch processing. Storm② is suitable for real-time analytics, online machine learning, continuous computation, and more[5]. To consider the application scenarios that these distributed systems are suitable for, it is more suitable to employ Hadoop for massive image files processing.

Most image processing algorithms have high complexities and are suitable for accelerators, especially general purpose graphic process unit (GPGPU or GPU for short). In recent years, GPGPU has been widely used in parallel processing. With support of CUDA③ and other parallel programming models for GPU, such as Brook+④ and OpenCL⑤, parallel programming on GPU has become convenient, powerful and extensive.

On processing massive image files with distributed platform and accelerators, two issues need to be addressed. Firstly, massive image processing is both I/O intensive and computing intensive, which should be managed concurrently through multi-threading with multi-core processors or GPU in-node, while simplifying parallel programming. In addition, to avoid that file I/O becomes the overall system bottleneck, data transfer between CPU and GPU should be optimized. Secondly, there exist considerable image processing algorithms and their variations for different kinds of image-related applications, while new algorithms are still emerging. Most of them were implemented as prototypes when they were proposed, and some of them have GPU-version implementations. A kind of system architecture which can easily integrate the existing CPU/GPU image processing algorithms can be proposed to deal with the above two issues. In other words, we should use available resource as much as possible rather than write everything by ourselves.

This paper proposes a distributed image processing system named SEIP (System for Efficient Image Processing on Distributed Platform), which is designed based on Hadoop[5] derived from MapReduce[3]. SEIP supports GPU-acceleration for image processing algorithms and can integrate existing CPU/GPU implementations of various kinds of algorithms. To make the system basically usable, basic image processing algorithms such as transformation of image size and color space are already integrated into the system. Two typical image feature extraction algorithms, LBP (Local Binary Patterns)[7-8] and SURF (Speeded-Up Robust Features)[9], are implemented both in CPU and GPU. A demonstration application for image feature extracting is also implemented. Additionally, SEIP uses a pipeline-based framework proposed in this paper to simplify parallel programming in application layer and accelerate I/O operations in image file processing.

In brief, our SEIP system has following characteristics.

1) SEIP employs extensible in-node architecture to support various kinds of image processing algorithms on distributed platform with GPU accelerators. By using general-purpose interfaces, image processing modules at bottom layer are extensible or pluggable; hence, various kinds of algorithm implementations for single nodes can be integrated into the system.

2) SEIP uses a pipeline-based framework for massive image file processing, in which files can be processed in parallel in multiple stages with transparent prefetching in each node by using simplified programming interface. Based on the framework, users can define their own image processing logic by re-writing several callback functions.

The rest of this paper is organized as follows. Section 2 presents the in-node architecture of the system and the implementation of two typical image feature extraction algorithms, and the in-node architecture corresponds to the first characteristic of SEIP. Section 3 brings a pipeline-based framework for massive file processing, corresponding to the second characteristic of SEIP. Section 4 details the design of SEIP system, and shows a demonstrated application. Section 5 evaluates

②Storm. http://storm-project.net/, Oct. 2015.

③NVIDIA CUDA Programming Guide 5.0. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Oct. 2015.

④AMD Brook+ Presentation. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/AMD-Brookplus.pdf, Oct. 2015.

⑤OpenCL—The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl, Oct. 2015.

the system. Section 6 introduces related work. Finally, conclusions are given in Section 7.

2 Acceleration of Image Processing

2.1 In-Node Extensible Architecture for Image Processing Algorithms

Currently, there exist many kinds of image processing algorithms and their variations, while new algorithms are still emerging. Most of them were implemented as prototypes when they were proposed or designed. Even more, some of them have implementations on GPU accelerators.

In this background, one focus of the SEIP system is how to integrate existing implementations of algorithms rather than to implement them one by one by ourselves or application developers. To achieve this goal, the SEIP system employs extensible in-node architecture to support various kinds of image processing algorithms on distributed platform with GPU accelerators.

Fig.1 shows the in-node extensible architecture of SEIP. Map/Reduce tasks, written in Java, invoke image processing algorithms via JNI⑥ and SEIP adaptive interface, which provides unified interface for image processing modules, such as basic algorithms, LBP[8], SURF[9], SIFT (Scale-Invariant Feature Transform)[10], Daisy[11], and so on. The image processing modules can be extended or substituted by customizing existing third-party implementations of algorithms and re-encapsulating them with the SEIP unified interface. The implementation of algorithms can be CPU-version, GPU-version, or both.

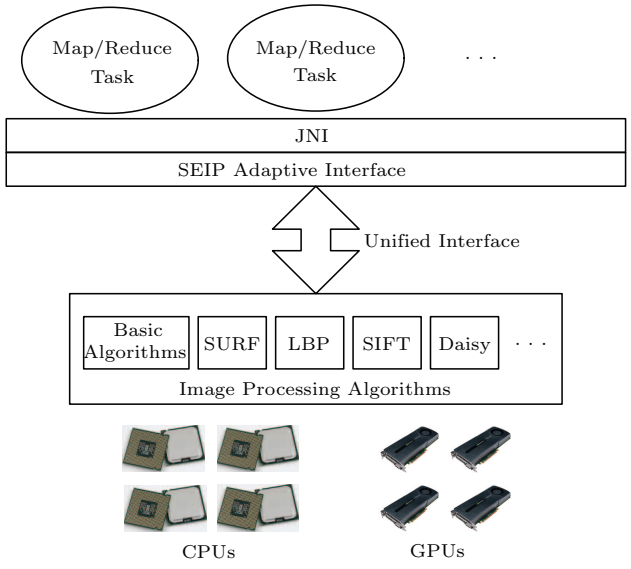

Fig.1. In-node extensible architecture for image processing algorithms.

The benefit of the architecture is that users can easily integrate their own or third-party implementations of algorithms into the system, while the system provides implementations of classic algorithms on both CPU and GPU, which can be used or modified to meet the requirements of specific applications.

2.2 Implementation of LBP and SURF Algorithms on GPU

To demonstrate the extensibility of the system and make the system basically usable, we implement two typical image feature extraction algorithms, LBP[7-8] and SURF[9] with CUDA⑦, and integrate them into the system. The data transfers between CPU and GPU are also optimized by using asynchronous mode.

LBP is a type of image feature used for classification in computer vision[8]⑧. This algorithm compares each pixel in an image with the surrounding ones and saves the result as sets of binary numbers. This paper accelerates LBP with GPU by using multiple CUDA threads to deal with thousand of pixels, i.e., computing the sets of binary numbers of all the pixels with atomicAdd in CUDA 4.0⑨. Finally, all the binary numbers are analyzed, and the results are constructed into an LBP histogram.

SURF[9] is an improved algorithm of classical SIFT[10], which "is faster than SIFT and has good performance as the same as SIFT"[12]. According to the algorithm, the integral image is firstly calculated based on original image, and then convolutions are calculated with integral image data by box filter to build scale spaces. Thus, the Hessian matrixes of pixels in all the scale spaces are obtained. From the Hessian matrixes of pixels, the feature points are chosen by the method of NMS (Non-Maxima Suppression). Finally, according to the Haar wavelet responses of each feature point's neighborhood, the dominant orientation and the SURF descriptor for one feature point are estimated. The

⑥Java native interface. http://docs.oracle.com/javase/7/docs/technotes/guides/jni/, Oct. 2015.

⑦NVIDIA CUDA programming guide 5.0. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Oct. 2015.

⑧Local binary patterns. http://en.wikipedia.org/wiki/Local_binary_patterns, Oct. 2015.

⑨CUDA Toolkit 4.0. https://developer.nvidia.com/cuda-toolkit-40, Oct. 2015.

count of feature points in each image is uncertain based on the Hessian threshold value, and the lower this value is, the more feature points could be described. We set Hessian threshold value to 100 on both GPU and CPU versions of SURF program. The acceleration of SURF with GPU mainly focuses on building scale spaces, estimating the dominant orientations and the SURF descriptor of all the feature points.

The GPU-version SURF program is written with CUDA 4.0[10] and modified from the implementations in OpenCV SURF GPU[11] and SURF[12].

Furthermore, to avoid that frequent data transfers between CPU and GPU become system bottleneck, this paper employs asynchronous transfer for multiple images between GPU and the CPU, which is an optimization for GPU-version implementation.

Image processing in the GPU device can be divided into three stages. In the first stage, image data is transferred from the host to the GPU memory. In the second stage, CUDA kernels are executed in the GPU device to process the image, e.g., image feature extraction, image content recognition, and image detection. The third stage is responsible to transfer results back to the host. To reduce the number of transmissions between the host and the GPU, processing results of multiple images can be transferred together.

Fig.2 shows the sequential execution and the asynchronous parallel execution of six images in GPU. In sequential execution, the first two stages must be executed in sequence, and the third stage of the six images can be executed at one time. In asynchronous parallel execution, three CUDA streams are created to process the six images. The first two stages among different images can be executed concurrently in CUDA streams. For example, when image 0 has been transferred into GPU memory by CUDA stream 0, it is not required to wait for the completion of CUDA kernel of this image, and image 1 can enter its first stage by employing CUDA stream 1. Similarly, other images can also be processed concurrently, and the degree of parallelism depends on the number of created CUDA steams. Thus, multiple images can be processed concurrently in asynchronous execution, which takes full advantage of the asynchronies between the host and the GPU device, and is more efficient than sequential execution.

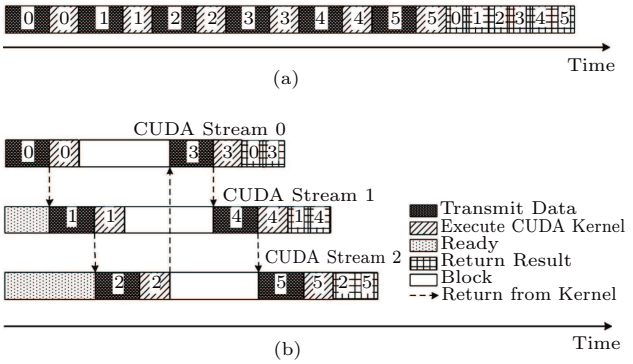

Fig.2. (a) Sequential execution vs (b) asynchronous parallel execution for image processing in GPU.

The original CPU-version codes of LBP and SURF come from LBP[13] and OpenCV SURF CPU[14] respectively. We re-write the codes for GPU and optimize asynchronous mode.

Table 1 shows the performance of LBP and SURF on processing 10 000 images in single-node environment, which is a dual-way x86 server with two Intel Xeon 5650 (6 cores, 12 threads) processors and one NVIDIA Tesla C2075 (448 cores)[15]. All the images are from Corel-1k[16] , which contains 1 000 images (256×384 pixels) and we duplicate the images 10 times to obtain 10 000 images. T_{CPU}, T_{GPU} and T_{GPUAsync} indicate the processing time (millisecond) of CPU, GPU and GPU asynchronous versions respectively. The speedup of GPU version algorithm over CPU version is indicated by $Speedup_{\mathrm{GPU}}$, and that of GPU asynchronous

Table 1. Performance Comparison: GPU vs GPU Stream Versions of LBP and SURF

Performance	LBP	SURF
T_{CPU} (ms)	145 391.00	3 271 945.00
T_{GPU} (ms)	33 135.00	95 859.00
T_{GPUAsync} (ms)	28 416.00	90 146.00
$Speedup_{\mathrm{GPU}}$	4.39	34.13
$Speedup_{\mathrm{GPUAsync}}$	5.12	36.30

[10] CUDA Toolkit 4.0. https://developer.nvidia.com/cuda-toolkit-40, Oct. 2015.

[11] http://docs.opencv.org/modules/nonfree/doc/feature_detection.html, Oct. 2015.

[12] Speeded Up Robust Features. http://asrl.utias.utoronto.ca/code/gpusurf/, Oct. 2015.

[13] LBP (C language). http://www.ee.oulu.fi/∼topiolli/cpplibs/files/LBP.c, Oct. 2015.

[14] OpenCV SURF CPU. http://docs.opencv.org/doc/tutorials/features2d/feature_homography/feature_homography.html, Oct. 2015.

[15] NVIDIA Tesla 2075. http://www.nvidia.com/docs/IO/43395/NV-DS-Tesla-C2075.pdf, Oct. 2015.

[16] Corel-1k. http://wang.ist.psu.edu/docs/related/, Oct. 2015.

version algorithm over CPU version is indicated by $Speedup_{\mathrm{GPUAsync}}$.

For LBP algorithm, the speedup of GPU-version over CPU is 4.39x, and that of GPU asynchronous version over CPU is 5.12x. The efficiency ratio of GPU asynchronous version over GPU $((1/T_{\mathrm{GPUAsync}})/(1/T_{\mathrm{GPU}}))$ is about 1.17. For the SURF algorithm, the speedup of the GPU-version over CPU is 34.13x, and that of GPU asynchronous version over CPU is 36.30x. The efficiency ratio of GPU asynchronous version over GPU is about 1.06. The above results show that GPU can achieve better performance than CPU on LBP and SURF algorithms, and the asynchronous optimization on GPU has further performance improvements.

2.3 Unified Interface

According to the unified interface, to add an existing implementation of an algorithm to the system, users just need to re-encapsulate their source code by defining an interface function, insert the function name into the dispatch list *AlgorithmDispatcher*, and define a number for the algorithm in the enum *algorithmNum*. After that, MapReduce applications that are written in Java at upper layer can invoke the algorithm by calling the entry of unified interface *AlgorithmEntry* with the algorithm number just defined. The brief C code is as what Fig.3 shows.

```
/*define algorithm number*/
enum algorithmNum{
   ALGORITHM_LBP_CPU;
   ALGORITHM_LBP_GPU;
   ALGORITHM_SURF_CPU;
   ALGORITHM_SURF_GPU;
   ...
};
/*dispatch list of interface functions*/
void (*AlgorithmDispatcher[ ])(char *, void * )
={
   LbpCalledBySEIP_CPU,
   LbpCalledBySEIP_GPU,
   SurfCalledBySEIP_CPU,
   SurfCalledBySEIP_GPU,
   ...
};
/*entry of the unified interface*/
AlgorithmEntry(int algorithm,
char *imageNames[ ], void *params){
   /*call interface function of
   specified algorithm */
   (*AlgorithmDispatcher[algorithm])(
   imageNames, params );
}
```

Fig.3. Example of adding existing implementations of algorithms to the system.

3 Acceleration of Massive File Processing

Images are commonly stored as individual files. To process a large number of image files efficiently, two problems need to be considered. The first problem is the coordination between parallelism and system I/O bottleneck. It is obvious that programmers should accelerate image processing through high parallelism by using multi-core processors in-node. However, blindly increasing the number of working processes/threads will bring numerous file accesses, which may aggravate system I/O bottleneck. Moreover, multiple parallel working processes/threads that access files simultaneously may interfere with each other and cause extra performance loss. The second problem is the complexity of in-node parallel programming. Compared with sequential programming, writing multi-threaded programs is always an extra burden for programmers.

SEIP uses a pipeline-based framework for massive image files processing, in which files can be processed in parallel in multiple stages with transparent prefetching in each node by using simplified programming interface. Based on the framework, users can define their own image processing logic by re-writing several callback functions.

3.1 In-Node Pipeline Framework for Massive Image Files

According to the design of pipeline-based framework, the processing of an image file is composed of following stages.

1) Stage 1: prefetching stage, which is responsible for loading the image file into in-node memory. In massive image files processing, the application knows the name and location of each file, which guarantees the accuracy of prefetching. Prefetching can be divided into two categories: predictive prefetching and informed prefetching[13]. Predictive prefetching is implemented at the file system level and predicts future read/accesses based on past I/O system call patterns. Predictive prefetching remains application-independent. Informed prefetching informs the file system of future data according to the hints provided by applications. Compared with predictive prefetching, informed prefetching is more accurate because of the support from applications.

2) Stage 2: image processing stage, which is responsible for processing the image.

3) Stage 3: result generation stage, which is responsible for accumulating and saving processing results.

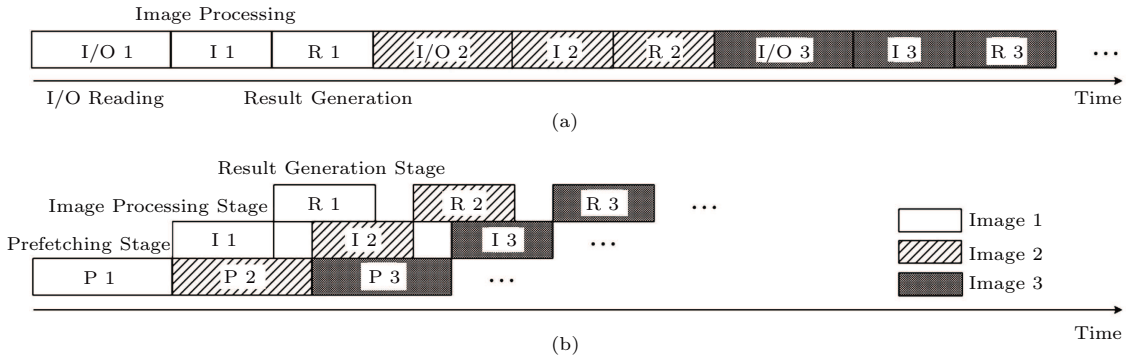

Fig.4. (a) Normal image processing vs (b) pipelining image processing. I/O: I/O reading; P: prefetching stage; I: image processing (stage); R: result generation (stage).

Normal image processing and pipelining image processing are demonstrated in Fig.4. In normal image processing, every image goes through I/O reading, image processing, and result generation in turn. When the image is in I/O reading or result generation, most of computing resources in the node are idle, which degrades the system performance. In comparison, we divide the entire process into prefetching stage, image processing stage, and result generation stage in pipelining image processing. Multiple images are handled in parallel in different stages, e.g., when image 1 is in image processing stage, image 2 has been prefetched into memory.

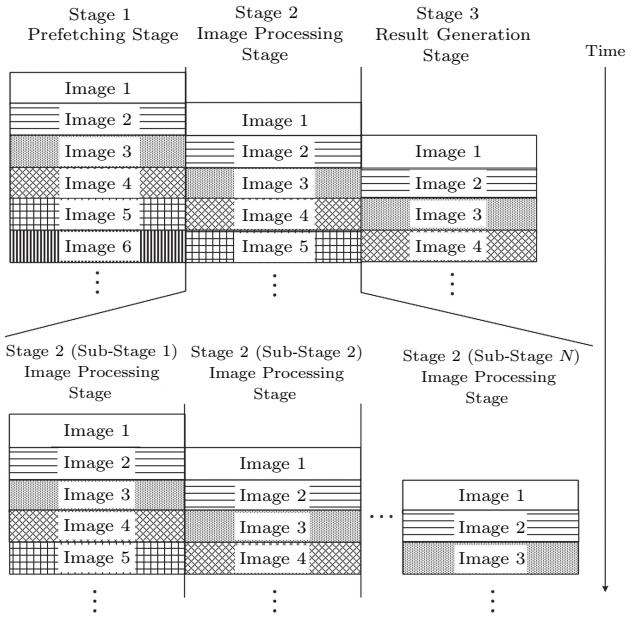

Fig.5. Images in the pipeline framework.

As shown in Fig.5, pipeline for massive file processing has the ability to handle multiple images simultaneously. If the complexity of the algorithm is high, the image processing stage could be divided into multiple sub-stages. Working processes/threads could run in stages to process images concurrently, and combine with the effect of file prefetching. In this way, in-node multi-core resources can be utilized more efficiently.

3.2 Programming Interface

The pipeline-based framework provides a simplified programming interface in Java for upper-layer applications, in which class MIP is the main class and methods of the class are listed in Table 2.

Table 2. Methods in Class MIP

Method	Description
public MIP (String pipelineName, int bufferSize, int stageNumber)	Constructor for initializing pipeline framework, setting the pipeline name, size of buffer and stage number
public void MIPSetFileDirectory (String pipelineName, String path)	Setting the addresses of massive images
public void MIPSetStage (String pipelineName, String stageName, CallbackFunc stageFunc)	Setting stage name, corresponding function and initializing the output buffer implicitly
public Object MIPStageInput (String stageName)	Setting the stage's input
public void MIPStageOutput (Object outputData)	Setting the stage's output
public void MIPCheck (String pipelineName)	Checking the end of pipeline, and waiting for stages to quit
public void MIPEnd (String pipelineName)	Ending pipeline and destructing pipeline buffers implicitly

The pipeline framework is implemented as multi-threading. Double-end queues are used for exchanging data between neighboring stages. As shown in Fig.6, in prefetching stage, a number of images are prefetched

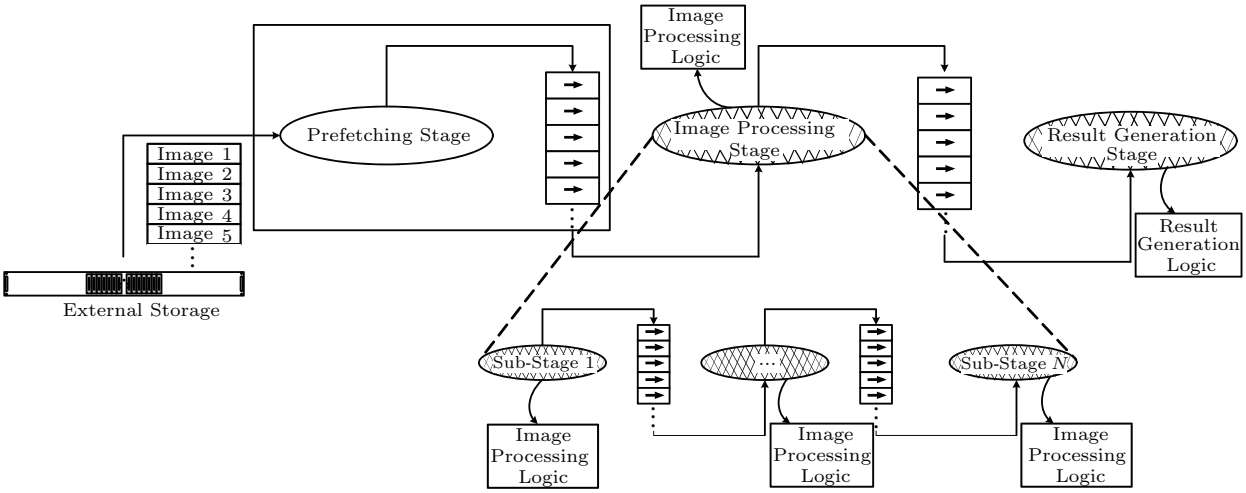

Fig.6. Parallel framework for massive images processing.

from external storage into double-end queue by dedicated thread according to the list of image files, and the size of queue is decided by users when initializing the pipeline framework. Image processing stage fetches data from the prefetching double-end queue. If image processing is complex, this stage can be divided into sub-stages, in which image data is also pipeline processed, such as sub-stage 1, sub-stage 2, and sub-stage N. It is noted that, whether dividing image processing stage and the number of sub-stages are decided by users, and the programming interfaces for dividing have been offered by the pipeline framework. In result generation stage, users can handle image processing results. Users can define their own image processing logic by rewriting several callback functions, as shown in the shaded parts in Fig.6. Callback functions can be written through invoking *CallbackFunc* interface provided by the pipeline framework.

3.3 CPU and GPU Co-Processing in the Pipeline Framework

SEIP uses hybrid CPU-GPU architecture to process massive image data, in which CPU and GPU can compute simultaneously. In image processing stage of the pipeline framework, applications can invoke CPU-version algorithms, GPU-version algorithms, or both. As shown in Fig.7, the prefetched images can be processed by both CPU and GPU resources in the image processing stage.

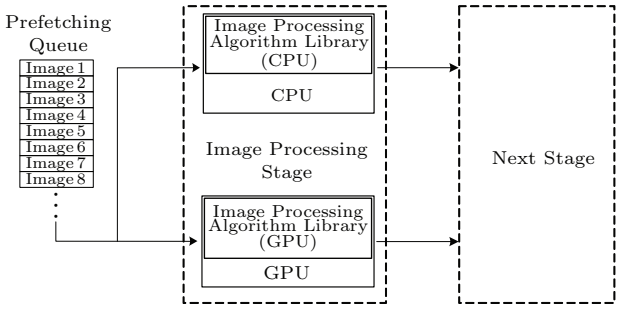

Fig.7. Hybrid CPU-GPU architecture in SEIP.

3.4 Application Development

Implementing an application or integrating an existing one in SEIP involves two parts of work.

1) *Integrating Image Processing Algorithms into the System.* There are two cases according to the implementation status of the algorithms used by the application. In the first case, if the algorithms have CPU-/GPU-version source codes for single-node environment, which is common for classic algorithms, the integration can be done by simply re-encapsulating the original source code with the entry functions of SEIP. By employing the entry functions of SEIP, users do not need to design, write and test complex adaptation codes to interact with the upper code in the distributed system anymore. In the second case, if the algorithm is totally novel and there is no implementation, the integration needs to be started by writing codes for the algorithm. This case also has another situation that the algorithm only has CPU-version code while users want to use GPU to accelerate image processing. In

this situation, users need to write GPU-version codes by their own, which is a fairly complicated work due to the programmability of GPUs, and one thing worth noting is that GPU-version code is not mandatory in the system.

2) *Writing Upper-Layer Program in MapReduce Style*. The program can accelerate the processing of massive image files by employing the pipeline framework of the system, and invoking low-layer image processing algorithms via unified interface. The pipeline framework is an option for users, who may choose to use serial programs or write multi-threaded programs in the node by their own. By using this pipeline framework, the I/O limitation from the distributed file system to memory may be alleviated.

4 System Architecture and Implementation

4.1 System Architecture

SEIP is built on Hadoop, and consists of one master and multiple workers in a cluster, and the master and workers are also namenode and datanodes of HDFS[14] respectively. The master controls multiple workers by allocating tasks to them, and workers equipped with GPU and multi-core processors make massive image data processing concurrently.

Fig.8 shows the system architecture of SEIP, where modules in ellipse are application-specific and need to be programmed or customized for an application, and modules in rectangle are generally application-independent.

The master in SEIP is responsible for image data pre-processing, and parallel task allocation and scheduling. Image pre-processing includes image normalization in size, color space, etc. Users can add individual external service into the application service module, for example, crawling images from the Internet. The module "combine massive image files" is an optimization of the splits for small-size files by combining multiple image files together so that a task can process multiple images at one time. The module "parallel task allocation and scheduling" provides MapReduce framework to applications by allocating tasks to multiple workers and collecting final results.

Workers in SEIP are responsible for image processing by invoking the GPU/CPU processing modules at bottom layer through the unified interface. Under the control of the master, the "parallel task management" modules in workers receive tasks from the master and start the processing. The "pipeline framework" module supports pipeline processing of image files for map/reduce tasks.

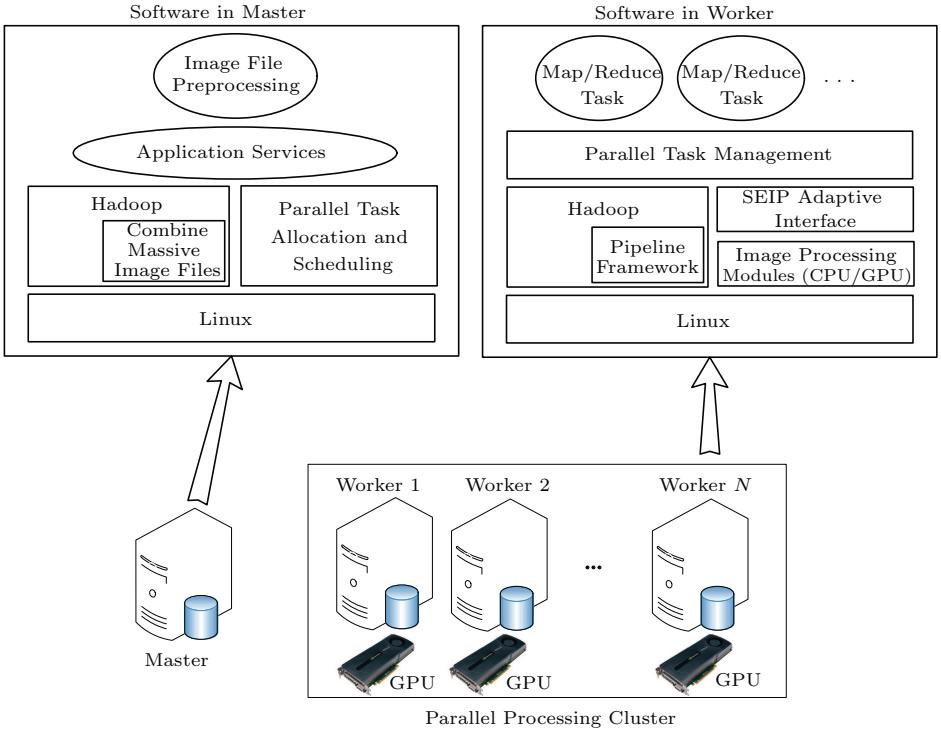

Fig.8. SEIP system architecture.

4.2 Image Data Processing

Image processing mainly consists of pipeline framework, parallel task allocation and scheduling.

4.2.1 Pipeline Framework in SEIP

As shown in Fig.9, a number of images can be packaged into one split by the module "combine massive image files", and then this split is allocated to a map task, which means that one map task can process thousands of images at one time. When packaging and allocating the splits, the module "combine massive image files" has taken into account the locality of image files.

The system combines the pipeline framework for massive image processing with MapReduce framework. The pipeline framework is implemented in the map. As described in Section 3, in the prefetching stage, image files are prefetched from HDFS to the memory based on the spilt allocated to the map task. If the image processing is complex, the image processing stage can be divided into multiple sub-stages, in which image data is also pipeline-processed, such as sub-stage 1, sub-stage 2, and so on. In this stage, different kinds of image processing algorithms at bottom layer are invoked. In the last stage, the result generation stage, processing results are assigned to reduce tasks by the master. Shuffle is not involved in image data processing.

It is noted that the pipeline framework is an option for users, who may choose to use serial programs or write multi-threaded programs in the node by their own.

4.2.2 Parallel Task Allocation and Scheduling

Parallel task allocation and scheduling in SEIP is mainly implemented on MapReduce framework, shown in Fig.10.

1) *Master*. The master in SEIP is responsible for image data pre-processing, parallel task allocation and scheduling, and controlling data transmissions among workers. The module "combine massive image files" is implemented in the master to manage the input split for processing thousands of images at one time by one map task. The input split includes the image information in HDFS.

2) *Workers* (*Map/Reduce*). After receiving the splits from the master, workers (Map) process the images by using in-node pipeline framework and image processing modules (CPU/GPU). The workers (Map) communicate the status with the master during the entire process, and the processing results are assigned to workers (Reduce) by the master. The result is in the form of key/value pair, such as (image address in HDFS, image processing result). All the key/value pairs are clustered and stored by workers (Reduce).

3) *Adaptive Interface*. The adaptive interface provides unified interface for upper-layer applications to invoke different kinds of image processing algorithms at bottom layer. The applications in workers (Map/Reduce) invoke the entry function of adaptive interface with arguments including image addresses, the specified algorithm and related parameters, and the adaptive interface dispatches the function call to the interface function of the corresponding algorithm. By using the corresponding algorithms, the images are pro-

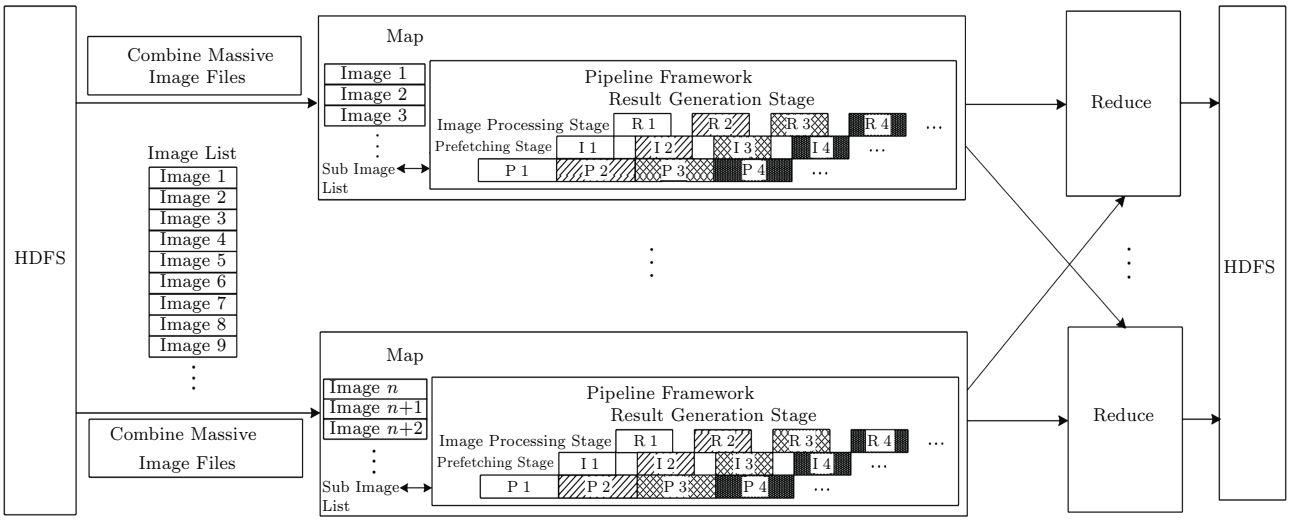

Fig.9. In-node parallel image processing under MapReduce framework.

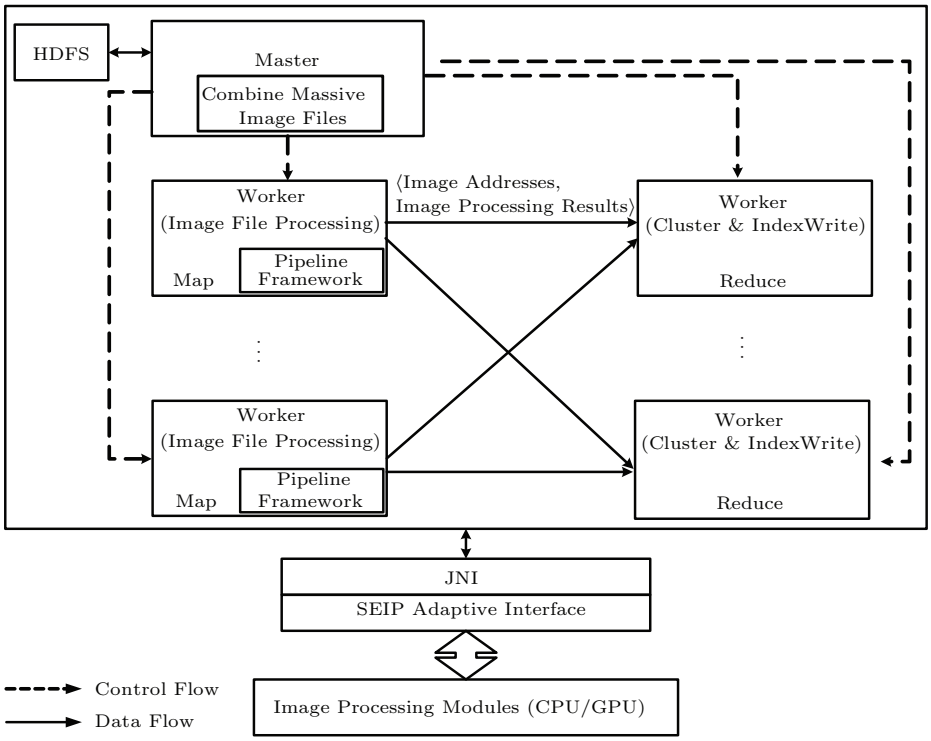

Fig.10. Parallel task allocation for image processing under the MapReduce framework.

cessed, and then results are returned to upper-layer applications.

4.3 Demonstrated Application

SEIP provides a general-purpose architecture to different kinds of image-related applications. As a demonstration, we develop an application, which simultaneously extracts LBP and SURF features of massive images, and then clusters and stores the image features. The application can be used as a backend system for applications such as content-based image retrieval, image annotation and classification, image content recognition, and so on.

The result of each image file includes its HDFS address, LBP feature data, and SURF feature data. Examples of HDFS address and LBP feature data are shown in Fig.11, which are expressed as String and float respectively[10].

For SURF feature data, because this kind of feature has many feature points, the result just saves the ten strongest SURF feature points, including their scales and orientation determined during the descriptor extraction process. For example, one SURF feature point's position and scale (vector (cv::KeyPoint)) are shown in Fig.12(a), and its SURF feature descriptor ((cv::Mat)) is shown in Fig.12(b).

```
hdfs://namenode:9000/data/picture/281.jpge,
0.02233867,0.04260894,0.01723662,0.03998897,0.17367622,
0.18360452,0.07246277,0.09107833,0.21883619,0.13816878
```

Fig.11. Examples of HDFS address and LBP feature data.

```
1.5515529632568359e+01, 3.0457647705078125e+02, 20.,
2.9893371582031250e+02, 2.1044113281250000e+04, 1, -1,
```

(a)

```
2.29092943e-03, 1.02277030e-03, 3.82741797e-03,3.51991830e-03,
-2.01794710e-02, 1.83875617e-02, 2.01794710e-02,1.83875617e-02,
-9.30161215e-03, 3.50645324e-03, 1.13981832e-02,6.75009191e-03,
4.57128137e-03, -3.26982234e-03, 5.72825689e-03,3.80155258e-03,
2.33589765e-02, -3.64588685e-02, 2.33589765e-02, 3.71857435e-02,
2.09339112e-01, -3.25226411e-02, 3.14705640e-01,2.99012333e-01,
-1.62219882e-01, 1.92898318e-01, 1.64862931e-01, 2.54309475e-01,
3.05538252e-03, 4.46586451e-03, 3.16889137e-02,1.86142288e-02,
1.28854187e-02, 3.62768956e-03, 1.37294801e-02,8.88268184e-03,
4.12445664e-01, -4.13150005e-02, 4.12445664e-01,1.74366385e-01,
-2.05291346e-01, 1.31799774e-02, 2.49552369e-01, 3.29077810e-01,
-3.96939320e-03, 1.50407488e-02, 3.26431170e-02, 6.06870390e-02,
1.42121245e-03, -9.14069358e-04, 1.43858779e-03, 5.91992587e-03,
3.61681581e-02, -3.66259068e-02, 4.24261875e-02,6.07904494e-02,
2.99054524e-03, -4.76840660e-02, 1.77362487e-02,5.17117009e-02,
-1.48533692e-03, -3.86029435e-03, 7.67699769e-03,6.56409469e-03,
```

(b)

Fig.12. Description of SURF feature point. (a) SURF feature point's position and scale. (b) SURF feature descriptor.

5 Evaluation

We evaluate SEIP with the demonstration application mentioned above. In this section, we will briefly introduce the experimental setup that includes the test platform, methods and data we use. Then we show the performance of SEIP with different optimizations and performance comparison with GPU pipeline framework and GPU multi-threading. Result verification and overhead evaluation are also given at the end.

5.1 Experimental Setup

The experimental system is a 4-node cluster connected with Gigabit Ethernet. Each node is a dual-way x86 server equipped with two Intel Xeon 5650 (6 cores, 12 threads) processors, 16 GB memory, eight 300 GB enterprise disks (10000 RPM, 32 MB cache), and one NVIDIA Tesla C2075 (448 cores)[17]. One server is used as both master and worker, and the other three servers are only used as workers. Linux (Redhat 5.6 Enterprise Edition, 64-bit) and Hadoop (version 0.20.203.0) are used in the evaluation.

All the experiments are based on the demonstration application in Subsection 4.3 to evaluate the performance of the system by simultaneously extracting LBP and SURF features of massive images (by 10 000 increments from 10 000 to 100 000) with four different optimizations including: 1) CPUs and pipeline framework; 2) only GPUs; 3) GPUs and pipeline framework; 4) CPU-GPU hybrid computation and pipeline framework. The four optimizations are compared with the unoptimized program, which simultaneously extracts LBP and SURF features with only CPUs in the cluster, rather than GPUs and pipeline framework. In addition, to evaluate the efficiency of our in-node pipeline framework, we also test a hand-coded multi-threading program with pipeline framework.

Images used in the evaluation come from Corel-1k[18], which contains 1 000 images (256×384 pixels), and we duplicate the images repeatedly to obtain 100 000 images. Three kinds of resolutions, 256×384 pixels, 400×400 pixels and 800×800 pixels, are used in the evaluation, and the latter two kinds are enlarged from original Corel-1k[18].

5.2 Results

The demonstrated application speedup in this paper means the ratio of the execution time of the unoptimized program and that of the optimized program.

The labels of different programs are shown in Table 3. Four speedups can be expressed as: "Cluster (CPU Pipeline) vs Cluster (CPU)", "Cluster (NVIDIA-GPU) vs Cluster (CPU)", "Cluster (NVIDIA-GPU Pipeline) vs Cluster (CPU)", and "Cluster (NVIDIA-GPU+CPU Pipeline) vs Cluster (CPU)".

Table 3. Label of Different Programs

Label of Different Programs	Description
Cluster (CPU)	Unoptimized program, with only CPUs in cluster, without GPUs and pipeline framework
Cluster (CPU Pipeline)	Optimized program, CPUs and pipeline framework in cluster
Cluster (NVIDIA-GPU)	Optimized program, only GPUs in cluster
Cluster (NVIDIA-GPU Pipeline)	Optimized program, GPUs and pipeline framework in cluster
Cluster (NVIDIA-GPU + CPU Pipeline)	Optimized program, CPU-GPU hybrid computing and pipeline framework in cluster

Fig.13 shows the speedup of different optimizations vs Cluster (CPU) on images with 256×384 pixels. As shown in the figure, CPU-GPU hybrid computation with pipeline framework achieves the highest performance, with an average speedup 13.6x over the Cluster (CPU), while the speedup of the program equipped with GPUs and pipeline framework is in the second place. The reason is that when CPU and GPU are collaboratively computing, the idle CPU resources are also used for image feature extraction, and this method is more efficient than GPU-only. It is also shown in Fig.13 that the program equipped with GPUs and pipeline framework achieves higher speedup than the program only equipped with GPUs, which demonstrates the effectiveness of in-node pipeline framework in the processing of massive files. The average speedup of "Cluster (CPU pipeline) vs Cluster (CPU)" is only about 1.05x. The reason is that the time cost of feature extraction in CPU is about 500∼600 ms per image, and

[17] http://www.nvidia.com/docs/IO/43395/NV-DS-Tesla-C2075.pdf, Oct. 2015.

[18] Corel-1k. http://wang.ist.psu.edu/docs/related/, Oct. 2015.

is longer than file prefetching time. Consequently, the prefetching has almost no effect in this situation.

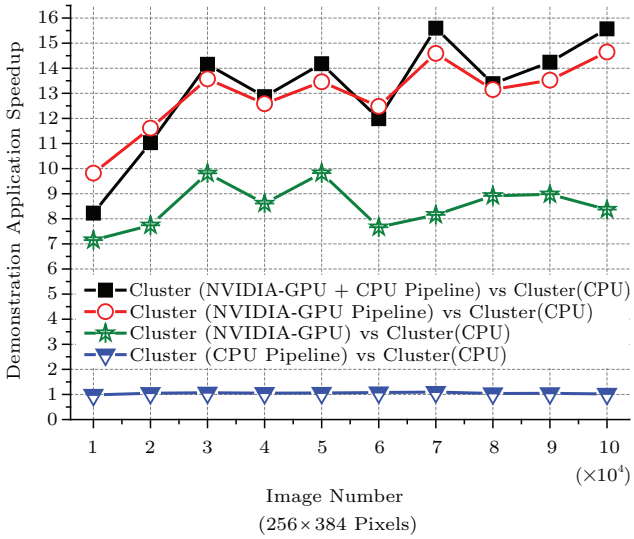

Fig.13. Speedup of different optimizations vs Cluster (CPU) (256×384 pixels).

The processing time bars of different optimizations vs Cluster (CPU) on images with 256×384 pixels are shown in Fig.14. As shown in Fig.14, Cluster (CPU) is the most time-consuming, and Cluster (CPU pipeline) takes the second time-consuming place. From Fig.13 and Fig.14, we can see that for images

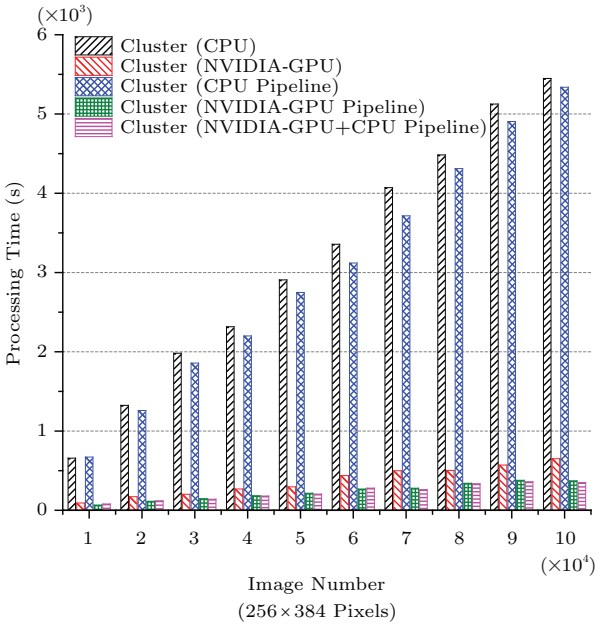

Fig.14. Processing time of different optimizations vs Cluster (CPU) (256×384 pixels).

(256×384 pixels), the efficiency ranking of different programs (from the highest to the lowest) is that:

Cluster (NVIDIA-GPU + CPU pipeline) >
Cluster (NVIDIA-GPU pipeline) >
Cluster (NVIDIA-GPU) >
Cluster (CPU pipeline) >
Cluster (CPU).

Fig.15 shows the speedup of different optimizations vs Cluster (CPU) on images with larger size, i.e., 400×400 pixels. As shown in the figure, the CPU-GPU hybrid computation with pipeline framework also achieves the highest performance with average speedup 16x, which is higher than that in Fig.13. The reason is that the computation complexity of feature extraction is in proportion to the size of the images, that is, the more pixels the images have, the higher the computation complexity is, and the more benefits the CPU-GPU hybrid computation gains. The processing time of different optimizations vs Cluster (CPU) on images with 400×400 pixels is shown in Fig.16.

From Fig.15 and Fig.16, we can see that for 400×400 pixels images, the efficiency ranking of different programs is the same as that of 256×384 pixels images.

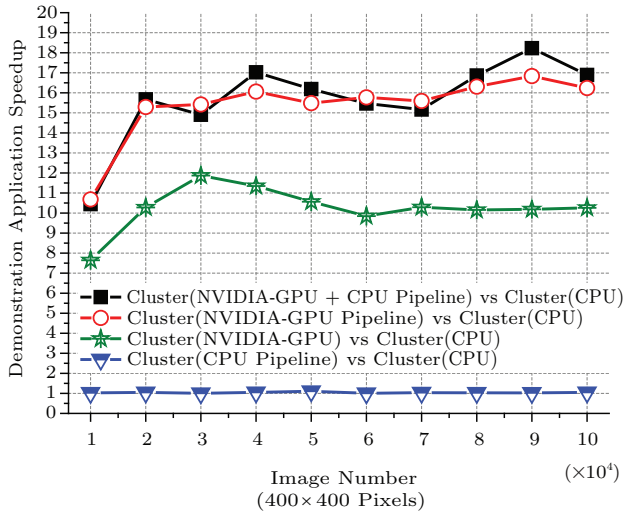

Fig.15. Speedup of different optimizations vs Cluster (CPU) (400×400 pixels).

Fig.17 and Fig.18 show results on images with further larger size, 800×800 pixels, where the average speedup of CPU-GPU hybrid computation with pipeline framework rises to 25x. The efficiency ranking of different programs is still the same as that of 256×384 pixels images.

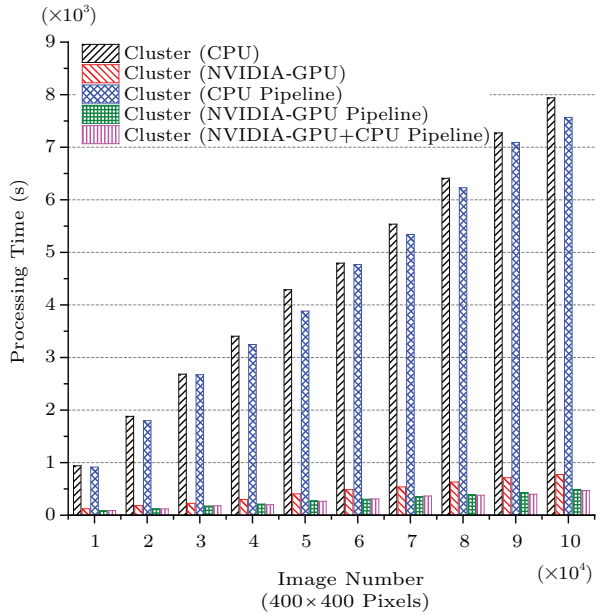

Fig.16. Processing time of different optimizations vs Cluster (CPU) (400 × 400 pixels).

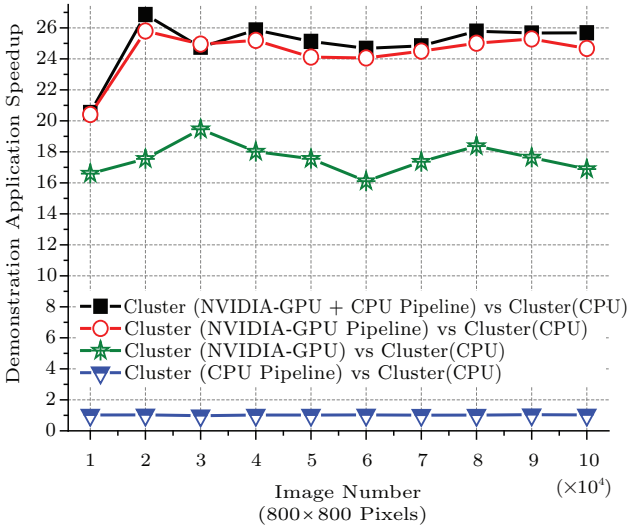

Fig.17. Speedup of different optimizations vs Cluster (CPU) (800 × 800 pixels).

As mentioned in Subsection 3.4, when writing programs for map tasks, users may choose to write multi-threaded programs by their own instead of using pipeline framework. To evaluate effects of these two options, hand-coded multi-threading programs are also tested and compared with the pipeline framework. In the evaluation, we use three threads to process image files concurrently.

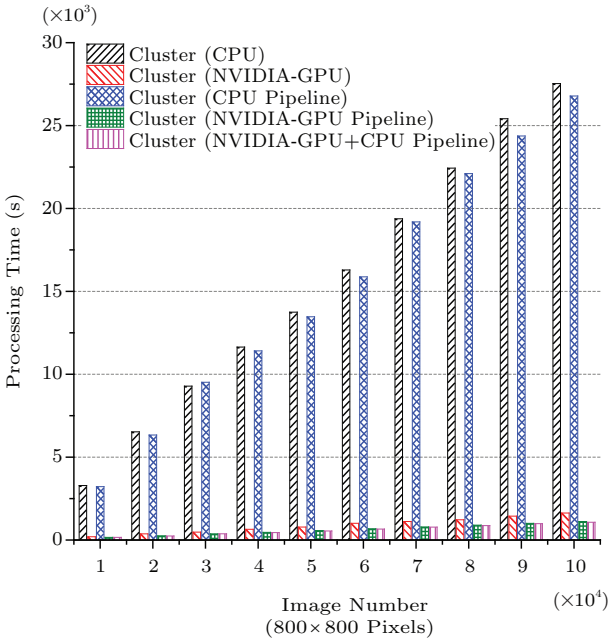

Fig.18. Processing time of different optimizations vs Cluster (CPU) (800 × 800 pixels).

Fig.19 shows the performance comparison between pipeline framework and multi-threading with different kinds of images. The lines in the sub-figures refer to the speedups of pipeline framework over multi-threading, which are calculated as $T_{\text{Cluster(multi-threading)}}/\ T_{\text{Cluster(pipeline framework)}}$.

As shown in the sub-figures (Fig.19(a), Fig.19(b) and Fig.19(c)), the speedup value is 1.38x for 256 × 384 pixels, 1.31x for 400 × 400 pixels and 1.25x for 800 × 800 pixels, respectively. The reason is that multiple parallel working processes/threads that access files and use GPU resource simultaneously may interfere with each other, which causes extra performance loss. Meanwhile, pipeline framework does not run into this problem.

5.3 Results Verification

The results of demonstration application are verified. Because the test dataset is expanded from Corel-1k[19] (includes 1 000 pictures, 256 × 384 pixels), we only need to extract LBP and SURF features of Corel-1k[19] simultaneously in single-node environment, by employing CPU and GPU version programs. The CPU-version programs are from OpenCV SURF GPU[20] and OpenCV SURF

[19] Corel-1k. http://wang.ist.psu.edu/docs/related/, Oct. 2015.

[20] http://docs.opencv.org/modules/nonfree/doc/feature_detection.html, Oct. 2015.

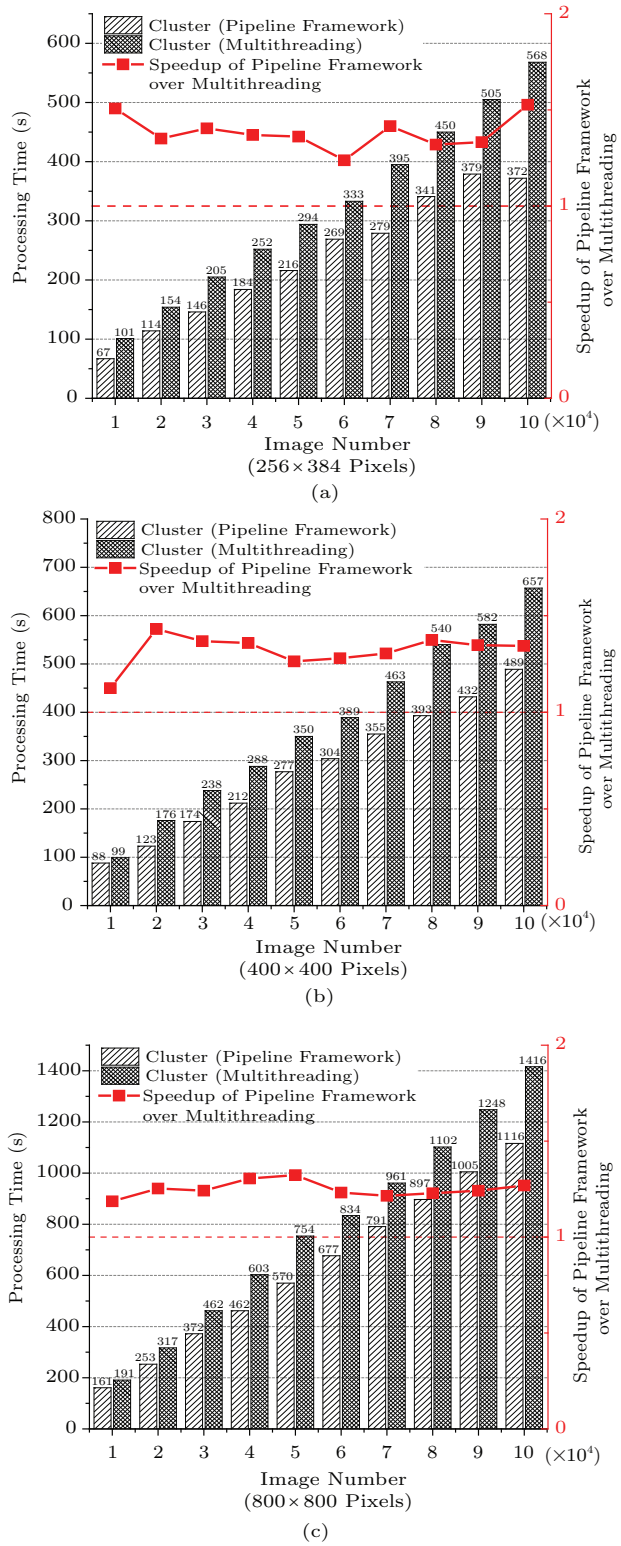

Fig.19. Performance comparison between pipeline framework and multithreading. (a) 256×384 pixels images. (b) 400×400 pixels images. (c) 800×800 pixels images.

CPU[21], which are provided by the algorithm founders or widely approved by third-party library, and results are certainly correct and can be the correct standards, while GPU-version ones are implemented by ourselves. The two kinds of results (GPU-version, CPU-version) from single-node environment are the same by comparison. Further, we simultaneously extract LBP and SURF features of Corel-1k[22] in SEIP, by implementing five programs. The programs include Cluster (CPU), Cluster (CPU pipeline), Cluster (NVIDIA-GPU), Cluster (NVIDIA-GPU pipeline), and Cluster (NVIDIA-GPU+CPU pipeline). Five kinds of results are obtained. Finally, we compare the results with the ones from single-node environment by employing euclidean distance, and all the results are equal.

Furthermore, the pictures expanded from Corel-1k[22] with 400×400 pixels and 800×800 pixels are all compared in the same way, and the results from SEIP are also equal to the ones from single-node environment.

5.4 Overhead Evaluation

The using of pipeline framework may bring some overhead, due to its stage interactions mainly. This subsection makes an evaluation on it with different image sets and numbers. The overhead of pipeline framework includes the time of pipeline initialization and that of stage interactions. Pipeline initialization needs to be done only once at the beginning, while stage interaction is along with every file that goes through the pipeline framework. Because these two kinds of overhead are tiny, we test every kind of overhead for several hundreds or thousands times, and take the average value for an overhead. The overhead of pipeline framework is given in Table 4 and the unit of time is millisecond.

Table 4. Overhead of Pipeline Framework

Kind of Overhead	Overhead (ms)
Pipeline initialization	13.060
Stage interactions	0.044

The pipeline framework works in nodes, thereby we make the tests in single-node environment.

As shown in Table 5, three programs are used by employing: 1) CPU and pipeline framework, 2) GPU and pipeline framework, and 3) CPU-GPU hy-

[21] OpenCV SURF CPU. http://docs.opencv.org/doc/tutorials/features2d/feature_homography/feature_homography.html, Oct. 2015.

[22] Corel-1k. http://wang.ist.psu.edu/docs/related/, Oct. 2015.

Table 5. Overhead of Pipeline Framework in Processing Time

Image Dataset	Image Number	Overhead of Pipeline Framework (ms)	CPU and Pipeline		GPU and Pipeline		Hybrid and Pipeline	
			Processing Time (ms)	Overhead Percentage	Processing Time (ms)	Overhead Percentage	Processing Time (ms)	Overhead Percentage
256×384 pixels	10^0	13.10	1 115	1.17	1 514	0.87	/	/
	10^1	13.50	3 156	0.43	1 667	0.81	/	/
	10^2	17.46	23 990	0.07	3 510	0.50	/	/
	10^3	57.06	240 801	0.02	12 381	0.46	12 470	0.46
	10^4	453.06	2 361 035	0.02	97 913	0.46	96 089	0.47
400×400 pixels	10^0	13.10	1 263	1.04	899	1.46	/	/
	10^1	13.50	4 231	0.32	1 496	0.90	/	/
	10^2	17.46	41 833	0.04	4 104	0.43	/	/
	10^3	57.06	349 085	0.02	17 001	0.34	18 139	0.31
	10^4	453.06	3 457 976	0.01	140 950	0.32	150 811	0.30
800×800 pixels	10^0	13.10	2 335	0.56	1 542	0.85	/	/
	10^1	13.50	12 824	0.11	1 718	0.79	/	/
	10^2	17.46	115 806	0.02	6 537	0.27	/	/
	10^3	57.06	1 242 379	≈ 0	44 019	0.13	48 023	0.12
	10^4	453.06	12 304 540	≈ 0	397 668	0.11	387 218	0.12

brid computing and pipeline framework. All the programs have the same purpose, which is to extract LBP and SURF features of image files simultaneously. Three image datasets (256×384 pixels, 400×400 pixels and 800×800 pixels) with different image numbers (1/10/100/1 000/10 000) are utilized. Because hybrid pipeline framework makes no sense for processing a small number of image files (1/10/100), the corresponding test data is empty. Each process time is an average value over five runs. The unit of processing time and overhead is millisecond. For all the programs with different image datasets, as the number of images increases, the overhead percentages of pipeline framework decrease and tend to be smooth. Consequently, the impact of pipeline framework is less with more image files.

6 Related Work

There already exist many kinds of image processing algorithms, such as LBP[8], SURF[9], SIF[10], and Daisy[11], and their variations are still emerging. Some of them are based on GPU resource. In addition, there are a couple of studies focusing on image processing in distributed systems.

The related work on the acceleration of image processing algorithms on GPU and image processing on distributed systems is given in this section.

6.1 Acceleration of Image Processing Algorithms on GPU

With the computing power of hundreds or thousands of stream processor cores, GPU is widely used in the acceleration of computation-intensive applications[15], formerly in high performances computing and recently extended to big data analysis. Many studies have demonstrated the power of GPUs in image processing domain, such as biomedical image analysis[16], image denoising[17], and image classification[18].

To simplify programming of GPU, some novel or existing parallel programming language extensions are used, such as CUDA㉓, Brook+㉔ and OpenCL㉕. CUDA is the most widely used GPU programming model, and it defines a suitable C language expansion pack for developers. A number of classical image processing algorithms have been implemented on GPU with CUDA, e.g., SUFR[19], SiftGPU[20], and fast-hog[21].

6.2 Image Processing on Distributed Systems

MapReduce is a programming model for processing and generating large datasets with a parallel, distributed algorithm on a cluster[3]. In recent years, it

㉓NVIDIA CUDA programming guide 5.0. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Oct. 2015.

㉔AMD Brook+ presentation. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/AMD-Brookplus.pdf, Oct. 2015.

㉕OpenCL—The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl, Oct. 2015.

has been widely used in considerable domains, such as big data processing[22] and data mining[23]. There are several classical MapReduce implementations, in which Hadoop is an open source implementation for cluster computing[5] developed by Apache Software, and Phoenix is an implementation for shared-memory systems with multi-core chips[24].

Recently, there are a couple of studies focusing on image processing in distributed systems. Moise *et al.*[25] made use of the MapReduce paradigm for image similarity search, which provides good practices and recommendations to image processing in Hadoop. Mills *et al.*[26] paid attention to large-scale feature matching in image processing. Moreover, they implemented the large-scale feature matching on distributed platform and GPU. Teodoro *et al.*[27] proposed a practical GPU-CPU hybrid system to make efficient collaborative use of CPUs and GPUs on a parallel system to accelerate large-scale image analysis. Teodoro *et al.*[28] also utilized the system proposed in [27] to analyze large-scale microscopy images for brain cancer studies. Hua *et al.*[29] proposed a near real-time scheme, called FAST. Sixty million images were analyzed by FAST, which could demonstrate the efficiency and efficacy of this methodology, and Liu *et al.*[30] made similar work.

There are also a series of systems that implement MapReduce on GPUs[31-35] for general purpose applications. Mars[31] is a GPU-based MapReduce system that utilizes GPU's power for MapReduce applications. Mars also employs Hadoop streaming technology[26] to integrate their framework into Hadoop. MapCG provides a framework that offers source code level portability between CPU and GPU[32]. By using MapCG's APIs, developers can write programs that execute on both CPU and GPU automatically. Lit[33] is a MapReduce-like framework based on CPU/GPU cluster, which provides an annotation approach to generating CUDA codes from Java codes in Hadoop. MGMR[34] is a MapReduce framework that utilizes multiple GPUs to manage large-scale data. Moreover, an upgrade version of MGMR, MGMR++, has been proposed to eliminate GPU memory limitation in old version[35]. Both systems were tested on one or two servers, and each server was equipped with two GPUs.

I/O limitations from distributed file systems to memory and from GPU to CPU have been concerned. Wittek and Darányi[36] accelerated text mining workloads in MapReduce-based distributed GPU environment, which focuses on the limitations of device memory and I/O problem in CPU-GPU hybrid systems. The solution is that I/O-bound operations are run on the CPU, while computation-intensive tasks are executed on the GPU.

Compared with the related work, the system proposed in this paper focuses on processing massive images on distributed platform with GPU accelerators, and can be easily extended or customized by integrating existing implementations of various kinds of image processing algorithms. In addition, the system employs a pipeline-based framework to process image files in parallel with transparent prefetching by using simplified programming interface.

7 Conclusions

With the demands of the rapidly growing of massive image processing in recent years, this paper proposed a distributed image processing system named SEIP to support efficient image processing on distributed platforms.

The system is built on Hadoop distributed platform with GPU accelerators, and employs extensible in-node architecture to support the integration of existing implementations of various kinds of image processing algorithms. In addition, the system uses a pipeline-based framework to simplify in-node parallel programming in application layer while improving the efficiency of massive image file processing. A demonstration application, which extracts LBP and SURF features of massive images, and then clusters and stores the image features, was also developed. The system is evaluated in a small cluster with GPU accelerators, and the evaluation results show the usability and efficiency of SEIP.

The system will be improved continually, and our future work will focus on more flexible pipeline framework with load-balancing, not only among stages, but also between CPUs-GPUs.

References

[1] Tanenbaum A S, Van Steen M. Distributed Systems: Principles and Paradigms. Upper Saddle River, NJ: Prentice Hall, 2007, pp.7-8.

[2] Fleischmann A. Distributed Systems: Software Design and Implementation. Springer-Verlag Berlin Heidelberg, 2012, pp.4-5.

[3] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 2008, 51(1): 107-113.

[26] Hadoop Streaming. http://hadoop.apache.org/docs/r1.2.1/streaming.html, Oct. 2015.

[4] Zaharia M, Chowdhury M, Franklin M J *et al*. Spark: Cluster computing with working sets. In *Proc. the 2nd USENIX Conference on Hot Topics in Cloud Computing*, Jun. 2010.

[5] White T. Hadoop: The Definitive Guide (1st edition). O'Reilly Media, Jun. 2009.

[6] Zaharia M, Chowdhury M, Das T *et al*. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. the 9th USENIX Conference on Networked Systems Design and Implementation*, Apr. 2012, pp.15-28.

[7] Ojala T, Pietikainen M, Harwood D. Performance evaluation of texture measures with classification based on Kullback discrimination of distributions. In *Proc. the 12th International Conference on Pattern Recognition* (*ICPR*), Oct. 1994, Volume 1, pp.582-585.

[8] Ojala T, Pietikainen M, Mäenpää T. Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2002, 24(7): 971-987.

[9] Bay H, Tuytelaars T, Van Gool L. SURF: Speeded-up robust features. In *Proc. the 9th ECCV*, May 2006, pp.404-417.

[10] Ng P C, Henikoff S. SIFT: Predicting amino acid changes that affect protein function. *Nucleic Acids Research*, 2003, 31(13): 3812-3814.

[11] Tola E, Lepetit V, Fua P. DAISY: An efficient dense descriptor applied to wide-baseline stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2010, 32(5): 815-830.

[12] Juan L, Gwun O. A comparison of SIFT, PCA-SIFT and SURF. *International Journal of Image Processing* (*IJIP*), 2009, 3(4): 143-152.

[13] Lewis J, Alghamdi M, Assaf M A *et al*. An automatic prefetching and caching system. In *Proc. the 29th IEEE International on Performance Computing and Communications Conference* (*IPCCC*), Dec. 2010, pp.180-187.

[14] Shvachko K, Kuang H, Radia S *et al*. The Hadoop distributed file system. In *Proc. the 26th IEEE Symposium on Mass Storage Systems and Technologies* (*MSST*), May 2010.

[15] Lindholm E, Nickolls J, Oberman S *et al*. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 2008, 28(2): 39-55.

[16] Hartley T D R, Catalyurek U V, Ruiz A *et al*. Author's retrospective for biomedical image analysis on a cooperative cluster of gpus and multicores. In *Proc. the 25th ACM International Conference on Supercomputing Anniversary Volume*, Jun. 2014, pp.82-84.

[17] McGaffin M G, Fessler J. Edge-preserving image denoising via group coordinate descent on the GPU. *IEEE Transactions on Image Processing*, 2015, 24(4): 1273-1281.

[18] Zhu L, Jin H, Zheng R *et al*. Effective naive Bayes nearest neighbor based image classification on GPU. *Journal of Supercomputing*, 2014, 68(2): 820-848.

[19] Cornelis N, van Gool L. Fast scale invariant feature detection and matching on programmable graphics hardware. In *Proc. IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops* (*CVPRW*), June 2008, pp.1-8.

[20] Wu C. SiftGPU: A GPU implementation of scale invariant feature transform (SIFT). http://cs.unc.edu/∼ccwu/siftgpu, Oct. 2015.

[21] Prisacariu V, Reid I. fastHOG — A real-time GPU implementation of HOG. Technical Report 2310/09, Department of Engineering Science, University of Oxford, January 2012.

[22] Jiang D, Chen G, Ooi B C *et al*. epiC: An extensible and scalable system for processing big data. *Proceedings of the VLDB Endowment*, 2014, 7(7): 541-552.

[23] Zhang X, Yang L T, Liu C *et al*. A scalable two-phase top-down specialization approach for data anonymization using MapReduce on cloud. *IEEE Transactions on Parallel and Distributed Systems*, 2014, 25(2): 363-373.

[24] Ranger C, Raghuraman R, Penmetsa A *et al*. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proc. the 13th IEEE International Symposium on High Performance Computer Architecture* (*HPCA*), Feb. 2007, pp.13-24.

[25] Moise D, Shestakov D, Gudmundsson G *et al*. Terabyte-scale image similarity search: Experience and best practice. In *Proc. IEEE International Conference on Big Data*, Oct. 2013, pp.674-682.

[26] Mills S, Eyers D, Leung K C *et al*. Large-scale feature matching with distributed and heterogeneous computing. In *Proc. the 28th IEEE International Conference of Image and Vision Computing New Zealand* (*IVCNZ*), Nov. 2013, pp.208-213.

[27] Teodoro G, Kurç T M, Pan T *et al*. Accelerating large scale image analyses on parallel, CPU-GPU equipped systems. In *Proc. the 26th IEEE International on Parallel and Distributed Processing Symposium* (*IPDPS*), May 2012, pp.1093-1104.

[28] Teodoro G, Pan T F, Kurç T M *et al*. High-throughput analysis of large microscopy image datasets on CPU-GPU cluster platforms. In *Proc. the 27th IEEE International on Parallel and Distributed Processing Symposium* (*IPDPS*), May 2013, pp.103-114.

[29] Hua Y, Jiang H, Feng D. FAST: Near real-time searchable data analytics for the cloud. In *Proc. the International Conference for High Performance Computing, Networking, Storage and Analysis* (*SC*), Nov. 2014, pp.754-765.

[30] Liu J, Huang Z, Cheng H *et al*. Presenting diverse location views with real-time near-duplicate photo elimination. In *Proc. the 29th IEEE International Conference on Data Engineering* (*ICDE*), Apr. 2013, pp.505-516.

[31] Fang W, He B, Luo Q *et al*. Mars: Accelerating MapReduce with graphics processors. *IEEE Transactions on Parallel and Distributed Systems*, 2011, 22(4): 608-620.

[32] Hong C, Chen D, Chen W *et al*. MapCG: Writing parallel program portable between CPU and GPU. In *Proc. the 19th ACM International Conference on Parallel Architectures and Compilation Techniques* (*PACT*), Sept. 2010, pp.217-226.

[33] Zhai Y, Mbarushimana E, Li W *et al*. Lit: A high performance massive data computing framework based on CPU/GPU cluster. In *Proc. IEEE International Conference on Cluster Computing* (*CLUSTER*), Sept. 2013.

[34] Jiang H, Chen Y, Qiao Z *et al*. Accelerating MapReduce framework on multi-GPU systems. *Cluster Computing*, 2014, 17(2): 293-301.

[35] Jiang H, Chen Y, Qiao Z *et al.* Scaling up MapReduce-based big data processing on multi-GPU systems. *Cluster Computing*, 2015, 18(1): 369-383.
[36] Wittek P, Darányi S N. Accelerating text mining workloads in a MapReduce-based distributed GPU environment. *Journal of Parallel and Distributed Computing*, 2013, 73(2): 198-206.

Tao Liu received his B.E. and M.S. degrees in computer science and technology from Shandong University, Jinan, in 2007 and 2010, respectively. Currently, he is a Ph.D. candidate in the School of Computer Science and Engineering, Beihang University, Beijing. He is a member of Sino-German Joint Software Institute at Beihang University. His research interests include parallel computing and high performance computing.

Yi Liu received his B.S., M.S. and Ph.D. degrees in computer science from Xi'an Jiaotong University, Xi'an, in 1990, 1993 and 2000, respectively. Currently, he is a professor of Beihang University, and vice director of Sino-German Joint Software Institute at Beihang University. His research interests include computer architecture and high performance computing.

Qin Li received his B.E. and M.S. degrees in computer science and technology from Beihang University, Beijing, in 2012 and 2015, respectively. His research interests include distributed system and high performance computing.

Xiang-Rong Wang received her B.E. degree in computer science and technology from Xi'an Jiaotong University, Xi'an, in 2011, and M.S. degree in computer science and technology from Beihang University, Beijing, in 2014. Her research interests include image processing algorithm and high performance computing.

Fei Gao received her B.E. and M.S. degrees in computer science and technology from Beihang University, Beijing, in 2012 and 2015, respectively. Her academic interests include image processing algorithm and high performance computing.

Yan-Chao Zhu received his B.E. degree in computer science and technology from Beihang University, Beijing, in 2012. Currently, he is a Ph.D. candidate in the School of Computer Science and Engineering, Beihang University. He is a member of Sino-German Joint Software Institute at Beihang University. His research interests include distributed system and high performance computing.

De-Pei Qian graduated from Xi'an Jiaotong University in 1977 and from North Texas State University in 1984 with his M.S. degree. He worked at Xi'an Jiaotong University from 1977 to 2010 and joined Beihang University in 2000. He is currently a professor and the director of Sino-German Joint Software Institute at Beihang University. He has been involved in the activities of the expert group for the National High Technology Research and Development Program (the 863 Program) since 1996 and led three key projects on high-performance computing. His current research interests include high-performance computer architecture and implementation technologies, distributed systems, and multicore/many-core programming support. He has published over 300 papers.