

Parallel Acceleration of SAM Algorithm and Performance Analysis

Haicheng Qu, *Member, IEEE*, Junping Zhang, *Member, IEEE*, Zhouhan Lin, and Hao Chen

Abstract—Advances in sensor and computer technology are revolutionizing the way that remote sensing data with hundreds or even thousands of channels for the same area on the surface of the earth is collected, managed and analyzed. In this paper, the classical Spectral Angle Mapper (SAM) algorithm, which is fit for parallel and distributed computing, is implemented by using Graphic Processing Units (GPU) and distributed cluster respectively to accelerate the computations. A quantitative performance comparison between Compute Unified Device Architecture (CUDA) and MATLAB platform is given by analyzing result of different parallel architectures' implementation of the same SAM algorithm. Especially for the property of GPU, this paper studied the balance between resource acquirement of each thread and the number of active blocks, and the impact of computational complexity on speedup. In addition, page-locked memory and stream are also introduced to make CPU and GPU work collaboratively. Moreover, we improved the SAM algorithm, in which several training samples are instead of a single one. Experimental results on hyperspectral data have shown that recognition result of the improved SAM algorithm is better than that only using single spectrum. On the other hand, the GPU parallel implementation achieves a higher speedup comparing with the multithread CPU counterpart. And the asynchronous transfer function of CUDA covers the data transmission latency effectively, thus improves the devices' resource occupancy significantly.

Index Terms—High-performance computing, SAM, GPU, distributed computing.

I. INTRODUCTION

IN recent years, several efforts have been directed towards the incorporation of high-performance computing (HPC) models to remote sensing missions. A relevant example of the use of HPC techniques (such as parallel and distributed computing) is hyperspectral remote sensing, in which an imaging spectrometer collects hundreds or even thousands of measurements (at multiple wavelength channels) for the same area on the surface of the earth [1]–[3]. Antonio *et al.* pioneered to research on the utilization of HPC infrastructure in hyperspectral imaging applications involving clusters, heterogeneous networks of computers, and specialized hardware devices, such

as field programmable gate arrays (FPGAs) and GPUs [4]–[6]. Many algorithms such as unmixing using the pixel purity index (PPI) or N-Finder algorithm to extract endmembers and principal component analysis (PCA) have been implemented on the platform of Compute Unified Device Architecture (CUDA) with remarkable speedups [1]. Yang *et al.* [7], [8] implemented SAM algorithm based on CUDA using a specific pixel value as reference spectra without transferring image data asynchronously. Lu and Park *et al.* [9]–[11] researched the key factors in design and evaluation of image processing algorithms on the massive parallel GPU using CUDA and proposed metrics to show the suitability in predicting the effectiveness of an application for parallel implementation. Now, the GPU architecture with small size, low cost and low power dissipation is drawing more attention than ever towards onboard and real-time analysis of remote sensing data; and distributed architecture such as grid computing and its evolution, cloud computing, currently represent a tool of choice for efficient distribution and management of very high-dimensional data sets in remote sensing and other fields [2]. Then, how to map traditional and classical algorithms to both architectures to achieve high performance has become one of the research interests recently.

SAM algorithm is a classical identification or classification method for hyperspectral data, which is suitable for parallel and distributed computing without any mutual influences between different pixels in computing spectral angle [8].

A plurality of spectral angles between spectrum of each pixel and training samples of the same class are calculated in the first. If the maximum spectral angle is larger than a certain threshold, this pixel is regarded as a target pixel. In the actual operation, we use morphological erosion when extracting training samples so that the mixed pixels are omitted. Similarly, morphological opening operation is performed after recognition to eliminate noise. Here, it is taken as an example to illustrate the parallel implementation based on GPU and distributed clusters.

This paper is organized as follows. The key technique and adopted approaches are explained in detail in Section II. The experimental results and discussion are reported in Section III. Finally, the conclusions are given in Section IV.

II. KEY TECHNIQUE AND METHODOLOGY

This section mainly introduces the classical spectral angle matching algorithm and its improved implementation version. Based on the MATLAB and CUDA platform, we designed the algorithm in three different approaches: stand-alone, multi-computer parallel and distributed computing. These are part of our attempts to apply high-performance computing into the field of remote sensing.

Manuscript received September 28, 2012; revised December 04, 2012; accepted January 04, 2013. Date of publication January 25, 2013; date of current version June 17, 2013. This work was supported in part by the National Natural Science Foundation of China (No. 61271348, 61172144). (*Corresponding author: J. Zhang.*)

H. Qu is with the Department of Information Engineering, Harbin Institute of Technology, Harbin 150001, and with the College of Software, Liaoning Technical University, Hu Lu Dao 125105, China (e-mail: haichengqu@gmail.com).

J. Zhang, Z. Lin, and H. Chen are with the Department of Information Engineering, Harbin Institute of Technology, Harbin 150001, China (e-mail: zhangjp@hit.edu.cn).

Color versions of one or more of the figures in this paper are available online at <http://ieeexplore.ieee.org>.

Digital Object Identifier 10.1109/JSTARS.2013.2239261

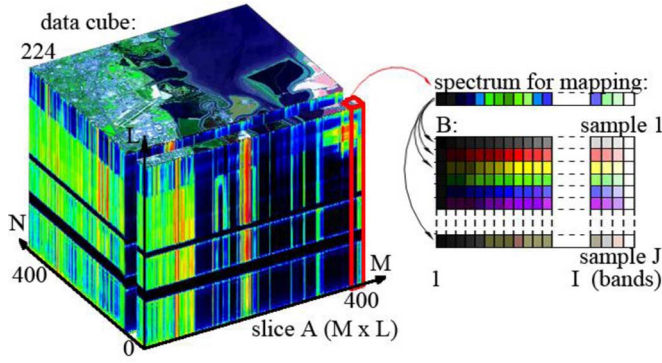


Fig. 1. Ketch map for matrix operation.

A. SAM Algorithm and Its Improvement for Target Detection

SAM algorithm is an identification method that permits rapid mapping by calculating the spectral similarity between the image spectra and reference reflectance spectra. The reference spectra can either be taken from laboratory or field measurements or extracted directly from the image. SAM measures the spectral similarity by calculating the angle between the two spectra which are treated as vectors in n -dimensional space. The main advantage of the SAM algorithm is that it's an easy and rapid method for mapping the spectral similarity of image spectra to reference spectra through some paralleled techniques such as multiple threads or multiple cores. It is also a very practical identification method because it restrains the influence of shading effects to accentuate the target reflectance characteristics.

In order to mitigate the spectral mixture problem for target detection, we select some reference spectra extracted from the reference target (such as plane) whose edges are eroded by mathematical morphology methods which can avoid mixed pixels to compute spectral angle with each pixel. These reference target spectrums come from special spectral database or the current hyperspectral image itself by manual selection. In this paper, we use the latter for the sake of simplification. The algorithm was implemented in the way of serialization and parallelization on the platform of MATLAB and CUDA.

B. Sequential Implementation of SAM

Spectral angle calculation is the most elapsed time segment of the whole algorithm. As a result, the parallel methods based on multi-core and multi threads are applied to accelerate the processing [12].

The original serial implementation method is as follows:

- 1) Record all bands values through "triple for loop" for each pixel.
- 2) Calculate spectral angle between current pixel and reference spectra.
- 3) Combine computational results.

In order to use the special advantage of matrix operation for MATLAB, the improved implementation scheme is proposed based on matrix operation. The sketch map of the scheme is shown in Fig. 1.

From Fig. 1, the source hyperspectral data is shown as a three-dimensional matrix Data ($M \times N \times L$, M : lines of source data,

N : columns of source data, L : bands of source data). Matrix A ($M \times L$) comes from a slice of source data and Matrix B stands for training samples coming from source data ($I \times J$, I :bands of reference spectra, J :number of samples). The matching process will be conducted by using the following formula:

$$\alpha = \cos^{-1} \left[\frac{\sum_{i=1}^{nb} t_i r_i}{\sqrt{\left(\sum_{i=1}^{nb} t_i^2 \right)} \sqrt{\left(\sum_{i=1}^{nb} r_i^2 \right)}} \right] \quad (1)$$

Where nb represents the number of bands, t_i and r_i represent test spectrum and reference spectrum, respectively.

Algorithm 1:

For iRow = 1 ~ M

For iColumn = 1 ~ N

SpectraForCompare \leftarrow Spectrum of pixel (iRow, iColumn)

Calculate α between SpectraForCompare and each training sample by (1)

Find max α

Compare to the threshold

Record result

End

End

C. Parallel Implementation of GPU

The improved SAM algorithm is implemented in three different ways through GPU accelerating processes. The first way is based on the platform of MATLAB2010b which supports GPU Array operation through calling the special function such as `gpuArray()`; the second one is achieved with the support of MATLAB Jacket Engine (Jacket Engine is developed specifically to accelerate MATLAB codes by introducing GPU computation. Through its high-level interface, the complexity of the underlying hardware is transparent to users). The third way is GPU-based implementation which is developed using CUDA [13].

The flow chart of CUDA implementation is shown in Fig. 2.

The key step of the flow chart is asynchronous execution between transferring data from host to device and GPU computation which can improve resource utilization rate through hiding overhead of loading data [14], [15]. So the whole procedures are shown as follow:

- 1) Load image data to page-locked memory.
- 2) Initialize GPU device.
- 3) Create streams and initialization (manage coincidences).
- 4) Create events (monitor devices progress and record the exact execution time).
- 5) Load m kernels, each kernel deals with specific data of itself (n slices of data are processed by a kernel each time).

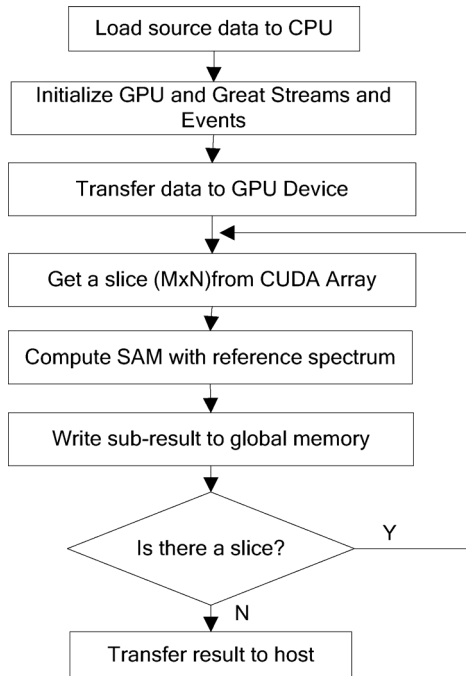


Fig. 2. Flow chart of CUDA implementation.

A slice of data serves as a cross session of the hyperspectral data cube).

- 6) Load p times 'cudaMemcpyAsync ()' asynchronously (p equals the number of streams).

Now, some acceleration strategies are applied to improve computational performance of GPU. The SIMD architecture of GPU provides a fine granularity, and underlies the tight relationship between the number of threads invoked and the computation. These threads are identified in the two-layer hierarchy with several built-in variables. The more threads invoked, the greater the parallel granularity. But as a consequence, more threads would need more resources, and cause performance degradation [17]–[19]. In Section 3, there is a concrete description that how to configure threads scheduler.

After threads optimization, a page-locked host memory technique is used to increase the data transfer speed between host memory and device memory. In general, cudaMalloc () function is used to allocate memory on the GPU. However, the CUDA runtime offers its own mechanism for allocating host memory: cudaHostAlloc () called pinned memory or page-locked memory. Page-locked buffers have an important property: the operating system (OS) guarantees that it will never page this memory out to disk, which ensures its residency in physical memory. The corollary to this is that it becomes safe for the OS to allow an application access to the physical address of the memory, since the buffer will not be evicted or relocated. So we further optimize the algorithm by introducing this kind of memory; the thread assignment is the optimum what we found above. In our GPU program, we restrict their use to memory which is used as a source or destination in calls to cudaMemcpy() such as loading hyperspectral images and reference samples from host memory to device memory and freeing them when they are no longer needed rather than

waiting until application shutdown to release the memory. Page-locked memory can improve the data transfer speed by almost 2 times. This is because page-locked memory does not follow the dynamic paging of OS and can be accessed by the data transfer engine of graphics device independently [20].

At last, covering the data transmission latency is the third acceleration strategies through the asynchronous transfer function of CUDA. The stream function of CUDA platform allows us to occupy both the computation engine and the data transfer engine simultaneously. The whole task is divided into several pieces, each for a single stream. Within the stream, all the steps are executed sequentially, but the executing sequence between streams are generally random. While one stream is computing, another can begin its data transfer step at the same time. So that the data transfer latency can be covered.

D. Distributed Implementation Based on MATLAB 2010b Cluster

In distributed computing, the whole task is divided into n separated pieces, and the master node arranges them to be done by different terminals in the network. Then, after all of the task pieces are finished, the master node sums up the partial results into a single one [13].

Distributed algorithm implementation is under the platform of MATLAB2010b based on Parallel Computing Toolbox. The source Data is divided into n subsets ($M \times (N/n) \times L$, N is a multiple of n), every subset is distributed to different worker-node to compute. The final computational result is produced through combining all the results of sub-Data on the master-node. The algorithm flow is the same as others. The description of distributed algorithm implementation is shown in Fig. 3.

Client and Job Manager are distributed onto the master node, which is responsible for task assignment, scheduling, and result presentation. Node 1 . . . Node N is child node, which are responsible for finishing tasks and returning partial results.

III. EXPERIMENTAL RESULTS AND ANALYSIS

A. Experimental Environment

1) *Experimental Data*: The source hyperspectral Data (400: lines of source data, 400: columns of source data, 224: bands of source data) is a low altitude AVIRIS image whose resolution is 3.5 m and radiation wavelengths is from 0.4–2.5 μm . The truth map is extracted from the original image.

2) *Software Environment*: Operating System: Windows XP/Ubuntu 10.10

Development platform: CUDA 4.0/MATLAB 2010b/Jacket 1.8

3) *Hardware Environment*: GPU: NVIDIA GTX 560 Compute Capability: 2.1 (Tesla S2050)

Cluster node: (Pentium Dual-Core E5300 @2.6 GHz 2 GB Memory) X4

B. Experimental Results

Different parallel architectures are fit for different application environments. Even though under the same parallel architecture, different speed-up ratios can be achieved by using different parameters [16]. In order to obtain comparable data, sev-

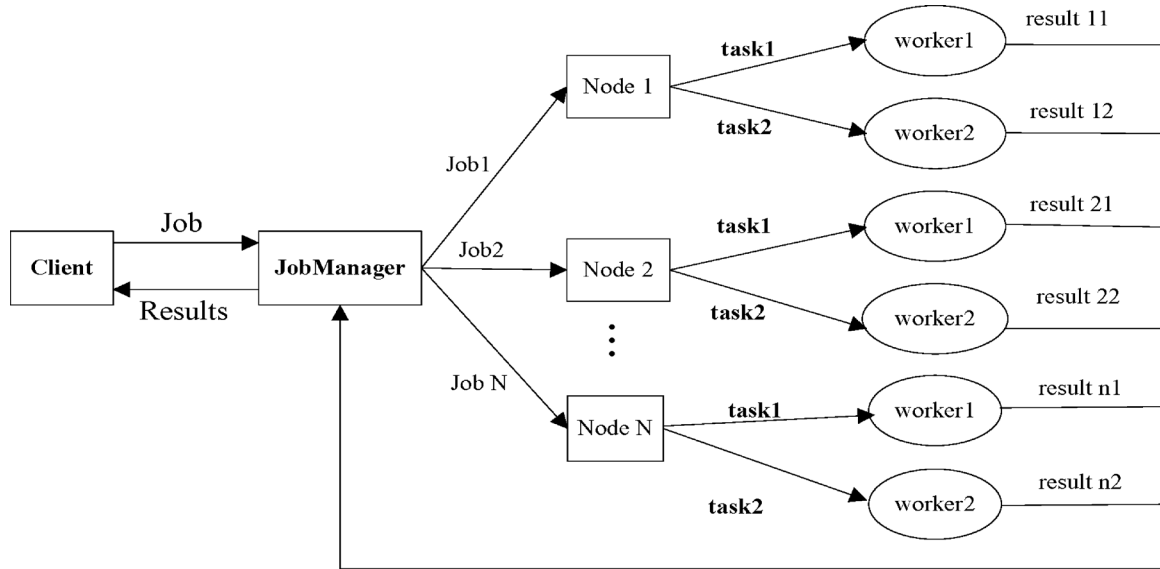


Fig. 3. Distributed architecture of MATLAB.

TABLE I
TIME FOR DIFFERENT PARALLELED TECHNIQUES (S)

Items	Trans Time t_1	Compute Time t_2	Show Time t_3	Total Time t_4
Mat(1)	1.23	216.17	0.31	217.71
Mat(2)	1.21	3.80	0.32	5.34
Mat(3)	1.21	5.07	0.30	6.58
Mat(4)	1.32	1.04	0.32	2.68

eral group experiments have been conducted. The results are given from four aspects:

1) *Parallel Performance Analysis Under the Platform of MATLAB*: Five experiments for different paralleled techniques have been done. The results are shown in Table I.

Four time indicators are figured through averaging five experimental results. Transmission time and core computational time are key evaluated parameters. The designations of four experiments are shown as follow:

Mat (1): original method implementation on the platform of MATLAB 2010b

Mat (2): improved method implementation on the platform of MATLAB 2010b

Mat (3): improved method implementation on the platform of MATLAB 2010b with GPU

Mat (4): improved method implementation on the platform of MATLAB 2010b with Jacket GPU accelerating

t_1 : time needed for loading data from hard disk to host memory;

t_2 : time needed for SAM computation;

t_3 : time needed for summing up and presenting results.

From the chart above we can find that the one exploiting Jacket Engine has reached the highest speedup, the total time is only 2.68 s, in which 1.04 s are computation time.

2) *Performance Analysis Under the Platform of CUDA*: To find the optimum thread assignment, we consider the executing time as a function of its two parameters: block size and grid

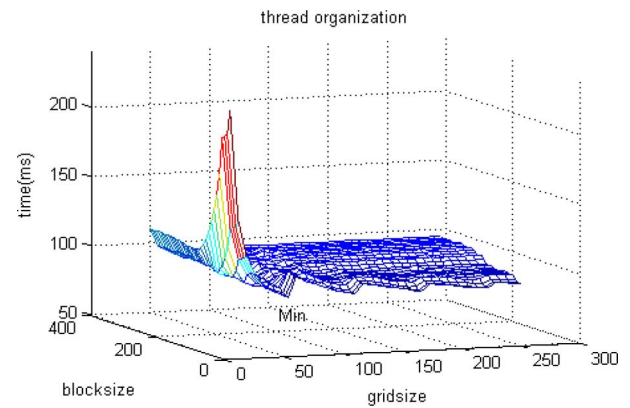


Fig. 4. Schedule of threads configuration.

size. The following experiment is to find the minimum of this function. Since the NVidia GTX560 device has 7 Multiprocessors with 48 cores each, it is reasonable to set the grid size as a multiple of 7. Note that the warp size are 32, we can reduce the amount of tests by setting block size a multiple of 8. So we get the value of this 2-D function, indicate the relationship between thread assignment and computation performance. The minimum point is found when grid size is 140, and block size is 8.

So in our threads schedule, the grid size and block size are 140 and 8 respectively. This means that there are 1120 threads activated in computing simultaneously.

To make the efficient of page-locked memory clear, we set the number of training sample as 1. The experimental results are given in Table II.

Time₁ : time needed for SAM computation;

Time₂ : total time (for GPU: computation times add data transmission time between host-memory and device-memory);

A: CPU running time based on C code

B-G: GPU running time

TABLE II
COMPUTATIONAL TIME FOR CPU AND GPU (MS)

Item	A	B	C	D	E	F	G
Time ₁	400.90	29.44	22.99	29.46	22.96	---	---
Time ₂	400.90	99.62	93.21	81.70	75.08	54.87	34.70

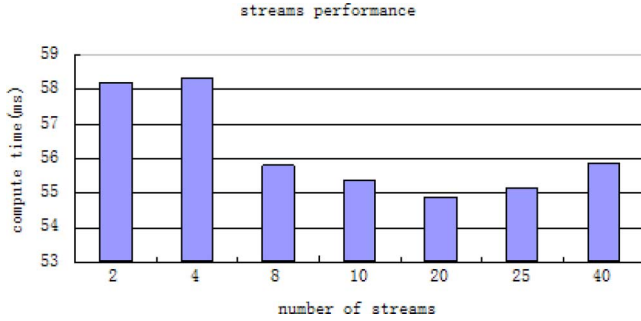


Fig. 5. Influence of streams.

B: program in which thread assignment is not optimized

C: after optimization of thread assignment

D: page-locked memory is introduced

E: page-locked memory is introduced and thread assignment optimized

F: stream and page-locked memory are all introduced and thread assignment optimized

G: the executing time of F on Tesla S2050

It is obvious to draw the conclusion from the chart above that the thread allocation and the CPU and GPU asynchronous collaborative work mode have an efficient impact on the performance of the program. Due to the increased data transfer bandwidth and number of computing cores, Tesla reaches a higher acceleration performance than GTX 560.

The efficiency of this multi-stream execution mode is related to the number of streams: generally, the more streams, the better performance. However, with the number of streams increase, the computation time increase. The performance reduces by reason of large system resource acquirement of myriads of streams. Fig. 5 shows the relation between computation time and number of streams. In the experiment, the best number of streams is set to 20.

From the hardware architecture of GPU, the main objective is to achieve parallel computation oriented the throughput of data using a large number of threads. Now we focus on this issue by the following experiment: test the speedups while thread assignment and the number of streams are fixed and only the size of training samples is different. The result is shown in Fig. 6.

The horizontal axis in Fig. 6 represents the number of training samples, each spectrum calculates spectral angle with every training samples. It is equivalent to an increase of computation. The speedup in the vertical axis is drawn when comparing with the C language.

The experiments in this section demonstrate that the stream-based algorithm in the CUDA platform accelerates SAM process the most, yielding an optimized executing time of 54.87 ms. Thread assignment and stream number impact

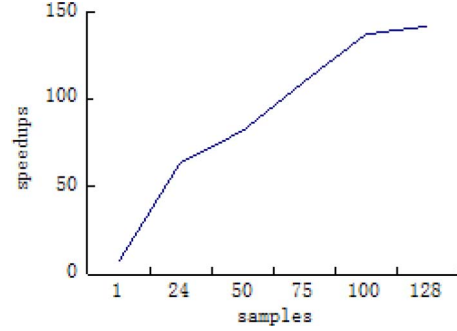


Fig. 6. The relationship between speedup and computation.

speedup a lot. Graphics devices are more suited to compute-intensive applications.

3) *Performance Subject to Different Task Size*: This experiment mainly analyzes the relationship between performances, data transfer and calculation amount for large amount of remote sensing data on MATLAB Distributed Computing platform. The remote sensing image is divided into blocks (2, 4 or 8 blocks), each one corresponding to a task; then the master (or dispatch) node distribute these tasks to the appropriate node (a dual-core CPU is deemed as two nodes); each node returns its result back to the dispatch node after the completion of its portion. Eventually the dispatch node generates the final results. Task allocation is designed in three ways: (1) file dependency: the master node distribute instructions and data to the child node; (2) path dependency: each child node gets its data from a unified data sharing pool, e.g., a shared disk in the LAN, the master node distribute only the control instructions; (3) mixed dependency: master node distribute control data and image data to the child node, the child nodes return processing results to the master node. In order to make the result more obvious, the algorithm is implemented by triple loop structure.

Under distributed environment mode, the balance of computation time is shown in Fig. 7.

As can be seen from Fig. 7, the program is the most efficient while 6 nodes (6-core) involved in the calculation and the image data is loaded from the shared disk. If the number of nodes continues to grow, the total amount of data to load will be even greater, thus reducing the performance.

4) *Impact of I/O*: Testing the impact of I/O on the performance, the number of nodes keeps constant. A four-node network is built in the experiment. And we designed five experiments in which the amount of I/O decreases monotonically. Under distributed environment mode, the total computation time is shown in Table III.

“A” stands for four nodes to compute with file dependence based on original method; “B” stands for four nodes to compute with path dependence based on original method; “C” stands for four nodes to compute with remote access data based on new method; “D” stands for four nodes to compute with local access data based on new method, in which each child node loads the image data from local address; “E” stands for four nodes to compute with less return data based on new method. In addition to load the image data from the local address, each child node further processes the result so that only a binary image is needed

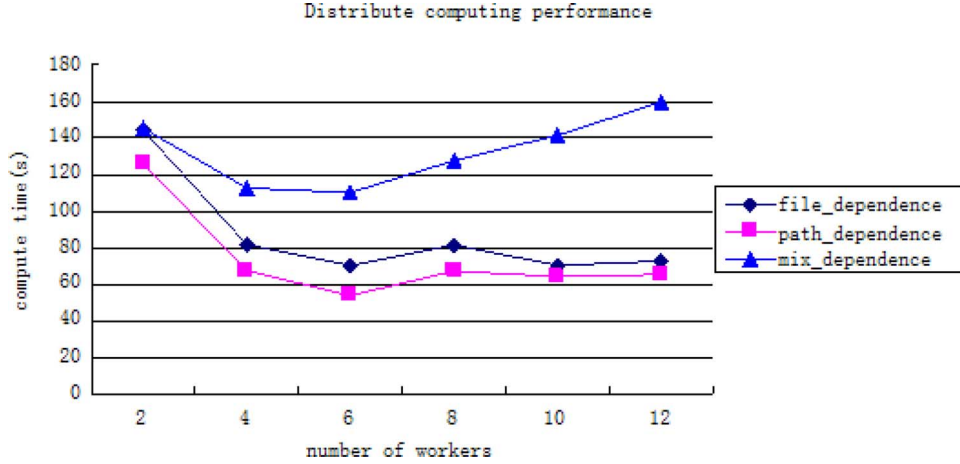


Fig. 7. Time for different size of task (s).

TABLE III
TIME FOR DIFFERENT DISTRIBUTED IMPLEMENTATION TECHNIQUES (s)

Item	A	B	C	D	E
Total time	83.60	67.37	7.83	9.31	5.71

to be transmitted to the master node. Thus we can further reduce the amount of data transfer.

It can be seen that the performance bottleneck is I/O. While the computing power of each node keeps constant, minimizing the amount of I/O will increase overall acceleration performance.

From the above experiments, it can be seen that improved matrix operation accelerates computing speed relative to original “triple for loop” code for segment of spectral angle computation on the platform of MATLAB 2010b. The key is that many fast algorithms based on multi-core and multi-thread are used to accelerate the speed of matrix operation. The method based on GPU of MATLAB 2010b platform could not achieve ideal effect, because there are many operations of data interaction from CPU-memory to GPU-device and from GPU-device to CPU-host which consume a lot of time. Another issue is that the efficiency of entailing GPU on MATLAB is lower than CUDA. In the latter platform, only 54.87 seconds are needed to finish the whole algorithm. However, the method based on Jacket accelerating of MATLAB 2010b platform which supports index of matrix speeds up by reducing time of data interaction. The computation speed is raised further by CUDA through optimum design of blocks and threads on GPU device, even though the data loading time (2.36 s) is added to the total executing time. Under the distributed environment, the whole computation time depends on computational power of worker-node and transmission delay. The shortest computation time (5.71 s) is based on four same worker-nodes through improved method and local data access. For small-scale image data processing, single computer parallel implementation is superior to multiple nodes (2.68 s–5.71 s) because of less I/O.

IV. CONCLUSION

In this paper, HPC techniques have been introduced into hyperspectral remote sensing missions. A notable acceleration per-

formance of SAM algorithm is achieved by using different parallel architectures such as GPU and distributed clusters. Encouraging experimental results show that GPU based on CUDA platform has played a role in accelerating improved SAM algorithm with a speedup of $63.9\times$ (taking 24 spectra as training samples), as the computation loads increases, the speedups are more obvious (while taking 128 spectra as training samples, the total speed up reaches $141.6\times$). On the other hand, what inhibits our further improvement on computation efficiency is the bottleneck of I/O. The algorithm which has a trait of intensive computation and less I/O is suitable for GPU to acceleration. Distributed clusters system is useful to resolve task-parallel rather than data-parallel problem. For SAM algorithm, if the source data is huge amount, distributed computing is the first candidate method; if the computation amount increase for the algorithm structure, accelerating by GPU will achieve better effect. The future work will consider how to realize parallel implementation and quantificational evaluation the acceleration performance of GPU for a general remote sensing application algorithm.

REFERENCES

- [1] A. Plaza, Q. Du, and Y.-L. Chang, “High performance computing for hyperspectral remote sensing,” *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 4, no. 3, pp. 529–543, Sep. 2011.
- [2] C. A. Lee *et al.*, “Recent developments in high performance computing for remote sensing: A review,” *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 4, no. 3, pp. 508–527, Sep. 2011.
- [3] A. Plaza *et al.*, “Recent advances in techniques for hyperspectral image processing,” *Remote Sens. Environ.*, vol. 113, pp. S110–S122, 2009.
- [4] A. Plaza *et al.*, “Improving the performance of hyperspectral image and signal processing algorithms using parallel, distributed and specialized hardware-based systems,” *J. Signal Process. Syst.*, vol. 61, pp. 293–315, 2010.
- [5] A. Plaza *et al.*, “An experimental comparison of parallel algorithms for hyperspectral analysis using heterogeneous and homogeneous networks of workstations,” *Parallel Computing*, vol. 34, pp. 92–114, 2008.
- [6] A. Plaza and C.-I. Chang, *High Performance Computing in Remote Sensing*. Boca Raton, FL, USA: Taylor & Francis, 2007.
- [7] J. Yang, Y. Zhang, and G. Dong, “Investigation of parallel method of RS image SAM algorithmic based on GPU,” *Science of Surveying and Mapping*, vol. 35, no. 3, pp. 9–11, May 2010.
- [8] X. Liu and Y. Kang, “Algorithm of spectral angle parallel classification on remote sensing image,” *Computer Science*, vol. 36, no. 9, pp. 267–269, Sep. 2009.
- [9] F. Lu and J. Song, “Survey of CPU/GPU synergetic parallel computing,” *Computer Science*, vol. 38, no. 3, pp. 5–9, Mar. 2011.

- [10] I. K. Park *et al.*, "Design and performance evaluation of image processing algorithms on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, pp. 91–104, Jan. 2011.
- [11] D. Castaño-Díez *et al.*, "Performance evaluation of image processing algorithms on the GPU," *J. Structural Biology*, vol. 164, pp. 153–160, Oct. 2008.
- [12] E. Christophe, J. Michel, and J. Inglada, "Remote sensing processing: From multicore to GPU," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 4, no. 3, pp. 643–652, Sep. 2011.
- [13] W. Fang, Q. Luo, and N. K. Govindaraju, "Mars: Accelerating MapReduce with graphics processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 4, pp. 608–620, Apr. 2011.
- [14] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Boston, MA, USA: Pearson Education, 2010.
- [15] D. B. Kirk and W.-M. W. Hwu, *Programming Massively Parallel Processors: A Hands-On Approach*. New York, NY, USA: Elsevier Science & Technology, 2010.
- [16] G. M. Striemer and A. Akoglu, "Sequence alignment with GPU: Performance and design challenges," in *IEEE Int. Symp. Parallel & Distributed Processing*, May 2009, pp. 1–10.
- [17] S. Ryoo *et al.*, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proc. 13th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, Feb. 20–23, 2008, pp. 73–82.
- [18] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1370–1380, Oct. 2008.
- [19] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *High Performance Computing—HiPC2007*, Berlin/Heidelberg, Germany, 2007, Springer, ser. Lecture Notes in Computer Science.
- [20] B. Huang *et al.*, "Development of a GPU-based high-performance radiative transfer model for the infrared atmospheric sounding interferometer (IASI)," *J. Computational Physics*, vol. 230, pp. 2207–2221, 2011.
- [21] J. Mielikainen, B. Huang, and A.-L. Huang, "GPU-accelerated multi-profile radiative transfer model for the infrared atmospheric sounding interferometer," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 4, no. 3, pp. 691–700, 2011.



Haicheng Qu (M'11) received the B.S. degree in computer science and technology from Qingdao Technological University, China, in 2005, and the M.S. degree in computer application technology from Liaoning Technical University, Huludao, China in 2008. Currently, he is pursuing the Ph.D. degree in signal and information systems at Harbin Institute of Technology, China.

His research interests include hyperspectral image processing, software architecture technology and high performance computing.



Junping Zhang (M'05) received the B.S. degree in biomedical engineering and instrument from Harbin Engineering University and Harbin Medical University, Harbin, China, in 1993, and the M.S. and Ph.D. degrees in signal and information processing from the Harbin Institute of Technology (HIT), in 1998 and 2002, respectively.

She is currently a professor with the Department of Information Engineering, School of Electronics and Information Engineering, HIT. Her research interests include hyperspectral data analysis and image processing, multisource information fusion, pattern recognition and classification.



in data or patterns.

Zhouhan Lin received the Bachelor degree from the School of Electronics and Information Engineering, Harbin Institute of Technology, Harbin, China, in 2012. Currently, he is a graduate student in the Institute of Image and Information Technology, Harbin Institute of Technology.

His research interests include pattern recognition, machine learning, remote sensing application and their parallel implementation on GPU devices. Especially, his studies currently focus on geometrical approaches to reveal implicit properties and relations



Hao Chen received the B.S., M.S., and Ph.D. degrees from the Harbin Institute of Technology, Harbin, China, in 2001, 2003, and 2008, respectively.

Since 2004, he has been with the Department of Information Engineering, School of Electronics and Information Engineering, Harbin Institute of Technology. His main research interests include image and video compression, remote sensing imaging, and remote sensing data processing.