International Journal of Signal Processing, Image Processing and Pattern Recognition
Vol.10, No. 2 (2017), pp.1-12
http://dx.doi.org/10.14257/ijsip.2017.10.2.01

GPU Implementation of Three-Stage Prediction with Adaptive Search Threshold for Hyperspectral Image Compression

Changguo Li[1,*] and Yuke Yang[2]

[1] *College of Fundamental Education, Sichuan Normal University, Chengdu 610068, China*

[2] *Department of Laboratory and Equipment Management, Sichuan Normal University, Chengdu 610068, China*

E-mail: 389224879@qq.com

Abstract

For the huge size of hyperspectral images with hundreds of bands, compression is necessary to save storage space and transmission time. Three-stage prediction with adaptive search threshold (TSP-AST), which consists of third-order interband predictor, bi-directional pixel search, backward pixel search with adaptive search threshold, and entropy coding, has been proven to be an effective lossless compression method for hyperspectral images, but its computational complexity is very high which makes its application to time-critical scenarios quite limited. In order to improve the computational efficiency of the algorithm, a parallel implementation of the TSP-AST algorithm is presented using compute unified device architecture (CUDA) on graphics processing units (GPUs) by exploiting the data parallel characteristics of the three prediction stages. The GPU parallel implementation is compared with the serial and multicore implementations on CPUs. Experimental results based on real hyperspectral images reveal remarkable acceleration factors and real-time computing performance in our four-GPU-based implementation of the TSP-AST compression scheme, while retaining exactly the same compression performance with regard to the serial and multicore versions of the compressor.

Keywords*: Graphics Processing Unit (GPU); Compute Unified Device Architecture (CUDA); hyperspectral images; lossless compression.*

1. Introduction

The advancement of sensor technology produces remotely sensed data that have a large number of spectral bands [1]. Hyperspectral images (HSI) are widely used in a variety of fields, such as terrain classification, agricultural monitoring, and military surveillance. Fine spectral resolution can be a desired featured when it comes to detecting fingerprints in the spectral response of a scene. Such applications are enabled by the richness of data captured by hyperspectral sensors. A problem of handling such wealth of information naturally arises and calls for the use of compression methods. These methods may be divided into lossless and lossy compression techniques. Though the latter can achieve higher compression ratios than the former, it isn't able to preserve the quality of original hyperspectral images. Therefore, there is a need to compress hyperspectral images adopting lossless methods. Moreover, high compression performance is important for saving storage space and transmission time. However, HSI compression is subject to important challenges, such as high dimensionality and insufficient compression performance in practice, which pose significant challenges to HSI compression.

Among many techniques for HSI compression, lossless compression algorithms have received a lot of interest. Several lossless algorithms, such as JPEG 2000 [2], JPEG-LS

ISSN: 2005-4254 IJSIP
Copyright ⓒ 2017 SERSC

[3], LUT [4], and LAIS-LUT [5], have relatively low complexity and thus are suitable for onboard compression. Other lossless algorithms such as FL [6], C-DPCM [7], S-FMP [8], and C-DPCM-APL [9] are more time-consuming and are thus suitable for ground compression. Most of these techniques rely on the fact that a scene of hyperspectral images is a sequence of images that have two types of redundancies, namely: spatial redundancy and spectral redundancy. Spatial redundancy is the correlation among spatially adjacent pixels in the same spectral band. Spectral redundancy is the correlation among pixels that have the same spatial location but are in adjacent spectral bands. As a result, the spectral redundancy has been exploited in combination with the spatial redundancy during the compression procedure.

Taking advantage of the spatial redundancy, the spectral redundancy and the calibration-induced redundancy in hyperspectral images, in [10], we proposed a novel three-stage prediction with adaptive search threshold (TSP-AST). It takes the third-order interband predictor, the bi-directional pixel search, and the backward pixel search with adaptive search threshold as its three predictors. As shown in [10], the compression performance of this method is comparable or superior to that exhibited by many other state-of-the-art techniques. However, its computational complexity was shown to be very high, thus limiting its application in time-critical scenarios. The reason is not only the extremely high dimensionality of hyperspectral images, but also that the first, the second and the third prediction stages adopted by this compressor carry out matrix operations, bi-directional pixel search, and backward pixel search for each current pixel, respectively. This results in a computational complexity that is even higher than the above-mentioned lossless algorithms.

In recent years, co-processing on Graphic Processor Units (GPUs) has become a disruptive technology in high performance computing. Exploiting the ever increasing transistor count, a growing number of processor cores have been added to GPUs. Consequently, GPUs have hundreds of parallel processor cores for execution of tens of thousands of parallel threads. Compute Unified Device Architecture (CUDA) is an extension to the C programming language offering programming GPU's directly [11]. In general, GPU has been able to offer two to three orders of magnitude speedup over CPU for various science and engineering applications, and therefore its exploding capability has attracted more and more scientists and engineers to use it as a cost-effective high-performance computing platform. A framework for efficient implementation of GPU adaptations of image processing algorithms in remote sensing is described in [12]. A near-real-time automatic target detection algorithm has been reported in [13]. To improve the execution performance of the weather research and forecasting (WRF) model, two different kinds of high-performance computing schemes: GPU-based Stony Brook University and GPU-based Goddard shortwave radiative, are given in [14] and [15]. A new multi-GPU implementation of the minimum volume simplex analysis algorithm for hyperspectral image unmixing is reported in [16]. By porting the calculation of composite kernels to GPUs, and performing intensive computations based on NVidia's compute unified device architecture, an efficient parallel implementation of composite kernels in support vector machines for hyperspectral image classification, is described in [17]. Nevertheless, there are very few GPU implementations of hyperspectral image compression algorithms in the literature. In [18], [19], and [20], the GPU acceleration implementations of the linear prediction method using constant coefficients (LP-CC), the predictive partitioning vector quantization (PPVQ), and lossy compression for Exomars (LCE) are reported, respectively. In [21], for decreasing the high computing complexity, a parallel error-resilient entropy coding (EREC) on a GPU is given.

In this paper, we propose a novel parallel TSP-AST method for hyperspectral image compression on GPUs. First, we analyze the CPU profile of the four stages of TSP-AST on running the AVIRIS images and find out that the three prediction stages take up 97 percent of the CPU execution time. Then, we exploit the data parallel characteristics of

 Copyright ⓒ 2017 SERSC

the three prediction stages and the advantages of utilizing GPU parallel computing principles to dramatically improve the computation speed of TSP-AST. The proposed parallel implementation properly exploits the GPU architecture at the low level and makes effective use of the computational power of both CPUs and GPUs synergetic fashion. The operations relevant to large data and intensive computations are executed on the GPUs, while the operations related with control are carried out on CPUs. The parallel implementation is executed using Nvidia's compute unified device architecture (CUDA), and the results of the presented method are compared with serial and multicore CPU implementations.

This paper is organized as follows. Section II describes the effective lossless compression using TSP-AST. Section III presents its parallel implementation on GPUs using CUDA. Section IV experimentally assesses the proposed method in terms of both compression performance and computing performance. Finally, Section V gives conclusions.

2. Lossless Compression Using TSP-AST

TSP-AST based on the spatial redundancy, the spectral redundancy and the calibration-induced redundancy can obtain excellent compression performance [10]. Its prediction process in mathematical terms is described as follows. First, the third-order predictor (IP3) is adopted and has the following prediction formula:

$$\hat{f}(k,m,n) = \alpha(v - m_v) + \beta(w - m_w) + \gamma(x - m_x) + m_y \text{ or } \hat{f}(k,m,n) = CY_n + C_0 \quad (1)$$

where $\hat{f}(k,m,n)$ is the prediction of the current pixel $f(k,m,n)$ (kth band, mth row, and nth column). "x", "w", and "v" denote the pixels colocated with the current pixel in three previuos bands and "m" stands for the expectation value of a random variable. $\alpha, \beta, and\ \gamma$ are the prediction coefficients. $Y_n = [v - m_v, w - m_w, x - m_x]$, $C = [\alpha, \beta, \gamma]^T$, and $C_0 = m_y$. Prediction coefficients $\alpha, \beta, and\ \gamma$ can be derived by solving a Wiener-Hopf equation

$$\begin{bmatrix} \sigma_v^2 & \sigma_{wv} & \sigma_{xv} \\ \sigma_{wv} & \sigma_w^2 & \sigma_{xw} \\ \sigma_{xv} & \sigma_{xw} & \sigma_x^2 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} \sigma_{yv} \\ \sigma_{yw} \\ \sigma_{yx} \end{bmatrix} \text{ or AC=B} \quad (2)$$

where

$A = \begin{bmatrix} \sigma_v^2 & \sigma_{wv} & \sigma_{xv} \\ \sigma_{wv} & \sigma_w^2 & \sigma_{xw} \\ \sigma_{xv} & \sigma_{xw} & \sigma_x^2 \end{bmatrix}$ and $B = [\sigma_{yv}, \sigma_{yw}, \sigma_{yx}]^T$, the statistical parameters can be approximated as (let W=32)

$$\begin{aligned} \sigma_x^2 &= E\{x^2\} - m_x^2 \\ &= \frac{1}{W^2}(W\sum_{i=1}^{W} x_i^2 - (\sum_{i=1}^{W} x_i)^2) \end{aligned} \quad (3)$$

$$\begin{aligned} \sigma_{wy} &= E\{wy\} - m_w m_y \\ &= \frac{1}{W^2}(W\sum_{i=1}^{W} w_i y_i - \sum_{i=1}^{W} w_i \sum_{i=1}^{W} y_i) \end{aligned} \quad (4)$$

Then the bi-directional pixel search (BDPS) as the second stage is applied. In this prediction process, a prediction of the current pixel $f(k,m,n)$ using all the causal pixels, which are in the previous, current and final bands, is made. Namely, two pixels which value equal to $f(k-1,m,n)$ and $f(k_{final1},m,n)$ are searched in the previous and final bands, respectively. If equal valued pixels $f(k-1,m_1',n_1')$, $f(k_{final1},m_1'',n_1'')$, $f(k-1,m_2',n_2')$, $f(k_{final1},m_2'',n_2'')$, …, $f(k-1,m_t',n_t')$, $f(k_{final1},m_t'',n_t'')$ ($1 \le t \le M \cdot N$, where M is the number of the rows, N is the number of the columns) are found. The decision among one of the

possible prediction values $f(k, m_1', n_1')$, $f(k, m_1'', n_1'')$, $f(k, m_2', n_2')$, $f(k, m_2'', n_2'')$, …, $f(k, m_t', n_t')$, $f(k, m_t'', n_t'')$ is based on the closeness of the values to the prediction reference value P_{ref}, where the prediction obtained from the first stage prediction is treated as the prediction reference P_{ref}. Otherwise, the estimated pixel value is equal to the prediction reference value P_{ref}. The whole search is performed in the reverse scanning order. Naturally, these guarantee that the final prediction value is rarely more than the prediction reference value P_{ref}.

In the third stage, the prediction obtained from the second stage is used to replace the prediction reference P_{ref}. The causal pixels in the current band are searched, and the one with a pixel value that is closest to P_{ref} is selected as the final prediction. In this algorithm, the adaptive search thresholds are used as the upper bound in searching for the optimal threshold value for each band. The adaptive search threshold of band k is defined as

$$T_k = \frac{\sum_{s=1}^{M}\sum_{t=1}^{N}\left|\hat{f}(k, s, t) - f(k, s, t)\right|}{MN} \tag{5}$$

where $\hat{f}(k, s, t)$ is the prediction of pixel $f(k, s, t)$ obtained from previous prediction stage, and $2 \le k \le K$ (K is the total number of bands). After prediction, the difference between the predicted and original values is computed. The difference is entropy-coded using an adaptive arithmetic coder (AAC). To sum up, the serial implementation of TSP-AST (referred to hereinafter as TSP-AST-S) can be summarized in Algorithm 1.

Algorithm 1. *Serial Algorithm of TSP-AST (TSP-AST-S)*

Input: *HSI images $R^{K\times M\times N}$, number of bands K, width of image M, and height of image N, pixel sample $f(k, m, n) \in R^{K\times M\times N}$;*

Initialization: *Set k=2, m=1, n=1, initialize the number of lines in search range L, the min_error, the starting point in a line sp, the error value P_{err}, the set of prediction error $E^{K\times M\times N}$: L=10, min_error=65536, sp=0, P_{err}=0, $\forall e(k, m, n) \in E$, and e(k,m,n)=0;*

Do:

  Do:

    Do:

      Step 1. *Calculate the stage-1 prediction value of the current pixel:*

$$\sigma_x^2 = \frac{1}{W^2}\left(W\sum_{i=1}^{W} x_i^2 - \left(\sum_{i=1}^{W} x_i\right)^2\right),$$

$$\sigma_{wy} = \frac{1}{W^2}\left(W\sum_{i=1}^{W} w_i y_i - \sum_{i=1}^{W} w_i \sum_{i=1}^{W} y_i\right),$$

$$\begin{bmatrix} \sigma_v^2 & \sigma_{wv} & \sigma_{xv} \\ \sigma_{wv} & \sigma_w^2 & \sigma_{xw} \\ \sigma_{xv} & \sigma_{xw} & \sigma_x^2 \end{bmatrix}\begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} \sigma_{yv} \\ \sigma_{yw} \\ \sigma_{yx} \end{bmatrix},$$

$$\hat{f}(k, m, n) = \alpha(v - m_v) + \beta(w - m_w) + \gamma(x - m_x) + m_y;$$

      Step 2. *Search for the set of the nearest neighbor L in the previous and the final bands, set L={ $f(k-1, m_1', n_1'), \ldots, f(k_{final}, m_t'', n_t'')$ }, give the set L' where corresponding pixels have the same spatial position in current band, L'={ $f(k, m_1', n_1'), \ldots, f(k, m_t'', n_t'')$ }, and make the prediction:*

$$(p, q) = \arg\left(\min_{\substack{x=m_1,\ldots,m_t \\ y=n_1,\ldots,n_t}} \{| f(k, x, y) - P_{ref} |\}\right),$$

$\hat{f}(k, m, n) = f(k, p, q)$, *then update* $P_{ref} = \hat{f}(k, m, n)$;

      Step 3. *In the search range, the one with a pixel value that is closest to P_{ref} is selected as the final prediction:*

$$T_k = \frac{\sum_{s=1}^{M}\sum_{t=1}^{N}\left|\hat{f}(k, s, t) - f(k, s, t)\right|}{MN};$$

 Copyright ⓒ 2017 SERSC

Step 3.1. *For v=m:max{(m-L),0},*
Step 3.1.1. *If v=m, sp=n-1;*
Step 3.1.2. *Else sp=M-1;*
Step 3.1.3. *For h=sp:1,*
Compute the error value:
$P_{err} = abs(f(k,v,h) - P_{ref})$;
Step 3.1.3.1. *If* $P_{err} < \min_error$,
$\min_error = P_{err}$;
Step 3.1.3.2. *If* $\min_error \leq T_k$,
return (f(k,v,h));
Step 3.1.4. *return (f(k,v,h));*
Step 3.2. $\hat{f}(k,m,n) = f(k,v,h)$;
Step 4. *Calculate the difference between the predicted and original values:*
$e(k,m,n) = f(k,m,n) - \hat{f}(k,m,n)$;
Step 5. *n=n+1;*
While $n \leq N$;
Step 6. *m=m+1;*
While $m \leq M$;
Step 7. *k=k+1;*
While $k \leq K$
Step 8. *the prediction error is entropy-coded using AAC;*
Output: *the compressed bit-stream of* $R^{K \times M \times N}$.
End

3. GPU Implementation

Although the solution of TSP-AST can effectively compress hyperspectral images, it is quite expensive in computational terms. The theoretical analysis indicates that the calculation of the three-order interband prediction and the loop iterative solution for $\hat{f}(k,m,n)$ in Step 1 and Step 3 of Algorithm 1, respectively, are the most time consuming parts, as the former involves heavy computations with matrices, the latter includes the backward pixel search is carried out for each pixel in its local neighborhood. Here, we take the AVIRIS Cuprite hyperspectral dataset (details of the dataset will be given in Section 4) as an example to experimentally analyze computational bottlenecks of Algorithm 1. Figure 1 shows the percentage of total CPU time consumed by the different steps of Algorithm 1. Since the calculation of three-order interband prediction and the backward pixel search with adaptive search threshold take 25.68% and 65.85%, respectively, the key for optimizing Algorithm 1 is accelerating the calculation of three-order interband prediction and the backward pixel search with adaptive search threshold.

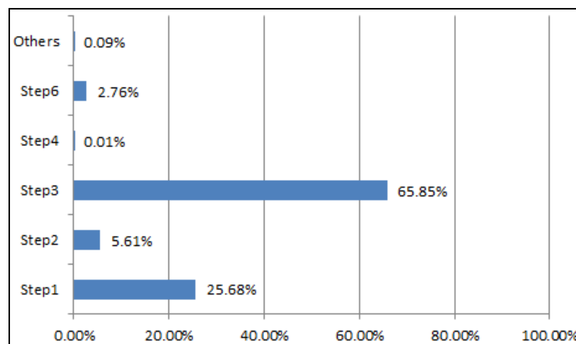

Figure 1. Percentage of Total CUP Time Consumed by Different Steps when Processing the AVIRIS Cuprite

Meanwhile, because of involving search, the bi-directional pixel search (namely the Step 2 of Algorithm 1) also leads to high computational overhead of the TSP-AST algorithm. In order to decrease the computational complexity, two LUTs for each of the

Copyright ⓒ 2017 SERSC

previous and final spectral bands are utilized and updated to speed up the search [10], which features lower computation times at the expense of extra memory usage for LUTs. Although the search processing is optimized, the bi-directional pixel search still takes more than 5% of the total execution time of Algorithm 1. In view of this, accelerating the Step 2 of Algorithm 1has become another key for optimizing Algorithm 1 not using LUTs.

Taking into consideration that the GPU is specifically optimized for computational and memory-intensive problems, while the CUP devotes more resources to caching or control flow operations, we assign the calculations related with matrix operations and pixel search on the GPU. At the same time, we allocate part of the computations operating on small data to the CPU. Furthermore, because of the quite expensive cost of input/output (I/O) communication between the host (CPU) and the device (GPU), we minimize the data transfers between the host and the device in our implementation. Namely, the data is stored in the local GPU memory as much as possible, and the storage space for intermediate variables of the iterative process is allocated in advance. According to CUDA programming paradigms, when executing a function (or kernel) on the device (GPU), one has to allocate memory on it, transfer data from the host to the GPU, and finally transfer data back to the CPU, freeing the device memory. The kernel can be either manually defined or implemented by an optimized routine, like those offered by libraries such as CUBLAS [22] and CULA [23]. However, the latter usually achieves better performance than the former for matrix multiplication [24]. Thus, CULA is chosen to realize the main matrix operations in this paper.

With the aforementioned issues in mind, a parallel implementation of TSP-AST (TSP-AST-P) has been developed, as illustrated in Figure 2. In the following, we describe the most relevant steps of the parallelization and architecture related optimizations conducted in the development of the TSP-AST-P algorithm.

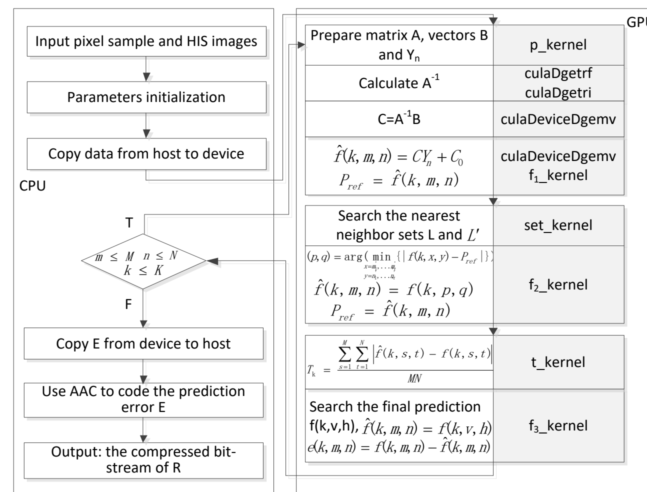

Figure 2. GPU Implementation of TSP-AST for Hyperspectral Compression

For step 1 in Algorithm 1, computation of C is most time consuming. To reduce the amount of calculation in the loop process as much as possible, the solution of C is decomposed into three steps. One is to prepare the matrix A, the vectors B and Y_n, in which A, B and Y_n are both stored on GPU global memory, and the temporary variables are stored on shared memory to minimize the latency of global memory access. A kernel function named p_kernel is defined to implement the preparation for A, B and Y_n on GPU. It launches a block with the same size of A, in which every thread calculates one element of A (i.e., $\sigma_{wy} = ((W\sum_{i=1}^{W} w_i y_i - \sum_{i=1}^{W} w_i \sum_{i=1}^{W} y_i))/W^2$). Then, matrix inversion of A is calculated. This operation is realized using functions culaDgetrf and culaDgetri, which are

 Copyright ⓒ 2017 SERSC

included in the high-efficient GPU accelerated linear algebra libraries of CULA. Finally, function culaDeviceDgemv in CULA is invoked to compute $A^{-1}B$ on the GPU. It takes full advantage of registers and shared memories of the GPU to achieve high efficiency in the multiplication of matrix and vector. After that, the remaining calculations of $\hat{f}(k, m, n)$ and P_{ref} are implemented by a kernel function f_1_kernel, in which $\hat{f}(k, m, n)$ and P_{ref} are allocated on GPU global memory, while the matrix-vector multiplication operation is realized via CULA function cluaDeviceDgemv. By these designs, the third-order interband prediction processes are reconstituted to maximize the parallel optimization.

For step 2 in Algorithm 1, the procedure searches for the nearest neighbor using all the causal pixels in the current, previous, and final bands. The main operation is related with pixel search in one causal neighbor, which can be efficiently performed by kernel functions set_kernel and f_2_kernel. The former conducts the mapping between thread and pixel on GPU, and each thread is responsible for the pixel search related to one pixel. According to the size of the causal neighbor, we start a $p_1 \times q_1$ thread grid on the GPU. The THREAD_SIZE and BLOCK_SIZE variables, respectively denoting the number of processing threads and processing blocks, are set to 32×32 and $((p_1+32-1)/32)\times((q_1+32-1)/32)$, in light of the computing capabilities of NVidia Kepler GTX690 and Tesla C2075 (details of the platforms will be given in Section 4).

For step 3, since adaptive search thresholds of hyperspectral image are also applicable to other images [10], we only compute these once for hyperspectral image, i.e., Cuprite (details of hyperspectral images will be given in Section 4). So these thresholds can be calculated in advance and be allocated on GPU global memory. The kernel function t_kernel in Fig. 2 is defined to compute these thresholds. After that, the remaining operation is search the final prediction. Since there are no dependencies among the pixels that inside the causal neighbor of the current pixel when this operation is carried out, this operation is ideally suited for GPU implementation. For the purpose of efficient parallelization, a kernel function called f_3_kernel is implemented to perform the rest operations of step 3 in Algorithm 1, where a grid is created according to the size of pixel search scope, and each thread implements the pixel search of each element. Finally, the results are copied from device (GPU) to the host (CPU) for analyzing the loop termination condition and subsequent operations.

To sum up, we provide a detailed step-by-step algorithm description of the parallel hyperspectral compression algorithm based on TSP-AST on GPUs in Algorithm 2.

Algorithm 2. *Parallel Hyperspectral Compression Algorithm Based on TSP-AST on GPU (TSP-AST-P)*

Input: *HSI images $R^{K\times M\times N}$, number of bands K, width of image M, and height of image N, pixel sample $f(k, m, n) \in R^{K\times M\times N}$;*

Initialization: *Set k=2, m=1, n=1, initialize the number of lines in search range L, the min_error, the starting point in a line sp, the error value P_{err}, the set of prediction error $E^{K\times M\times N}$: L=10, min_error=65536, sp=0, P_{err}=0, $\forall e(k, m, n) \in E$, and e(k,m,n)=0;*

Step 1. *Copy data from host to device*

Do:

  Do:

    Do:

      Step 2. *Calculate the first stage prediction on GPU*

        Invoke kernel function p_kernel to prepare maxtrix A, vectors B and Y_n

        Invoke culaDgetrf, culaDgetri and culaDeviceDgemv to compute $C=A^{-1}B$

        Invoke culaDeviceDgemv and f_1_kernel to calculate $\hat{f}(k, m, n) = CY_n + C_0$ and $P_{ref} = \hat{f}(k, m, n)$

      Step 3. *Calculate the second stage prediction on GPU*

        Invoke set_kernel to search the nearest neighbor sets L and L'

        Invoke f_2_kernel to compute

$$(p,q) = \arg(\min_{\substack{x=m_1,\dots,m_t \\ y=n_1,\dots,n_t}} \{| f(k,x,y) - P_{ref} |\}),$$

$\hat{f}(k,m,n) = f(k,p,q)$, *and* $P_{ref} = \hat{f}(k,m,n)$

Step 4. *Calculate the final prediction on GPU*

Invoke t_kernel to compute

$$T_k = \frac{\sum_{s=1}^{M}\sum_{t=1}^{N}\left|\hat{f}(k,s,t) - f(k,s,t)\right|}{MN}$$

Invoke f_3*_kernel to compute* $\hat{f}(k,m,n) = f(k,v,h)$ *and* $e(k,m,n) = f(k,m,n) - \hat{f}(k,m,n)$

Step 5. *n=n+1*

While $n \le N$

Step 6. *m=m+1*

While $m \le M$

Step 7. *k=k+1*

While $k \le K$

Step 8. *Copy E from device to host*

Step 9. *Code the E using AAC on CPU*

Output: *the compressed bit-stream of* $R^{K\times M\times N}$.

End

---

4. Experimental Results

The experimental platform used in our tests is a heterogeneous processor consisting of two CPUs and four GPUs. The hardware specifications and computing capabilities of the considered platforms are listed in Table 1. While, the tests were performed using the 64-bit Microsoft Windows 7 operating system. The version of OpenMP and CUDA are 2.0 and 6.0, respectively.

We evaluate the compression performance and computing performance of TSP-AST-P using the standard hyperspectral images, which acquired by the AVIRIS sensor in 1997. AVIRIS was devised by the JPL (jet propulsion laboratory) of NASA (National Aeronautics and Space Administration, USA), and it covers the 0.41-2.5 um spectrum range in 10 nm bands. This instrument contains four spectrometers that are flown at a 20 km altitude with a 17 m spatial resolution. The four scenes are Cuprite, Jasper Ridge, Lunar Lake, and Moffett Field, which are widely used for compression testing and the evaluation of hyperspectral images. The radiance data of the above four scenes were represented in 16 bits; each image has 512 lines, 224 bands, and 614 pixels/line.

Table 1. Hardware Specifications and Computing Capabilities of the Considered Platforms

Specification		Platform 1	Platform 2
CPU	Processor number	Intel Core i7-4790K	Intel Xeon E5-2620 v2
	Processor base frequency	4.0 GHz	2.1 GHz
	Number of cores	4	24 (2 CPUs)
	Main memory	16 GB	32 GB
GPU	Model	GTX 690	Tesla K20C
	Frequency of CUDA cores	915 MHz	0.71 GHz
	Number of CUDA cores	3072 (2 GPUs)	9984 (4 GPUs)
	Floating-point performance	5.6 Tflops	4.68 Tflops
	Dedicated memory	4 GB	20 GB
	Memory interface	512 bit	1280 bit
	Memory bandwidth	384 GB/s	832 GB/s
	CUDA compute capability	3.0	3.5

In order to demonstrate the performance improvements between the parallel implementations on multicore CPU platforms and our considered GPU platforms, a multicore implementation of TSP-AST (TSP-AST-M) has been developed using OpenMP

Copyright ⓒ 2017 SERSC

Application Program Interface (API) [25], which is adopted to explicitly address multithreaded and shared-memory parallelism. Two different platforms are used to perform the tests (see Table 1). The corresponding serial version (TSP-AST-S) is carried out on one core of the CPUs, and the multicore version (TSP-AST-M) is executed on all the cores of the CPUs. All the versions of the TSP-AST algorithm were implemented using the C++ programming language.

First of all, we test the compression performance of the three implementations. The compression results are quantitatively shown in Table 2. The results indicate that the proposed TSP-AST-S, TSP-AST-M, and TSP-AST-P obtain exactly the same compression performance in terms of bits per pixel (BPP). Here, it is worth mentioning that the serial (TSP-AST-S) and the parallel GPU (TSP-AST-P) versions obtain exactly the same compression results. The two versions can be, therefore, considered exactly equivalent in terms of BPP and only different in terms of computing performance. In the following part, we analyze the computing performance of the two versions by particularly focusing on the improvements of TSP-AST-P with regards to TSP-AST-S.

Table 2. Compression Results (in Bits per Pixel) for the Complete AVIRIS Images

Image	TSP-AST-S	TSP-AST-M	TSP-AST-P
Cuprite	3.68	3.68	3.68
Jasper Ridge	4.01	4.01	4.01
Lunar Lake	3.71	3.71	3.71
Moffett Field	4.02	4.02	4.02
Average	3.86	3.86	3.86

Figure 3 and Figure 4 show the speedup profiles of TSP-AST using 1 GPU on both platforms. For the platform 1, using 1 GPU we have a Step 1 speedup of 37×, a Step 2 speedup of 48×, a Step 3 speedup of 53×, and a total speedup of 45×. Accordingly, a Step 1 speedup of 43×, a Step 2 speedup of 55×, a Step 3 speedup of 61×, and a total speedup of 52× can be obtained on platform 2. Figure 5 and Figure 6 show the speedup profiles of TSP-AST using 2 GPUs and 4 GPUs on the 4 AVIRIS images, respectively. The average compression time of TSP-AST-P corresponds to 17 s and 4 s. The fact that using 2 GPUs and 4 GPUs does not have a total speedup near 2 and 4 respectively can be attributed to the reason that each GPU is not assigned a job of equal workload. Nevertheless, when there are more than one GPU available, a general trend can be seen that using more GPUs does give higher speedup for TSP-AST.

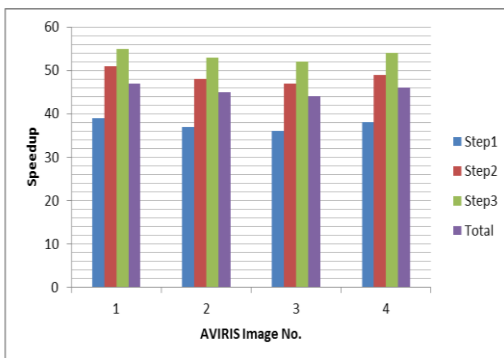

Figure 3. The Speedup Profile of TSP-AST Using 1 GPU on Platform 1

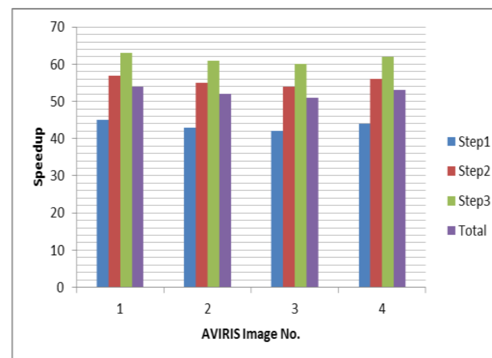

Figure 4. The Speedup Profile of TSP-AST Using 1 GPU on Platform 2

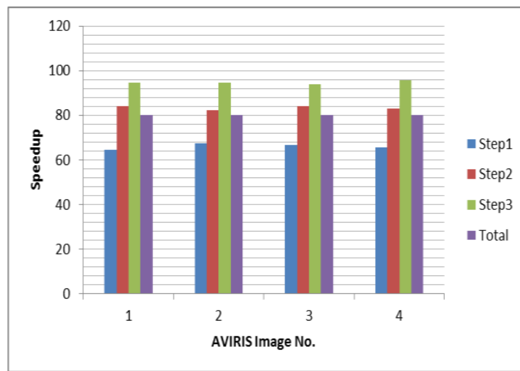

Figure 5. The Speedup Profile of TSP-AST Using 2 GPU on Platform 1

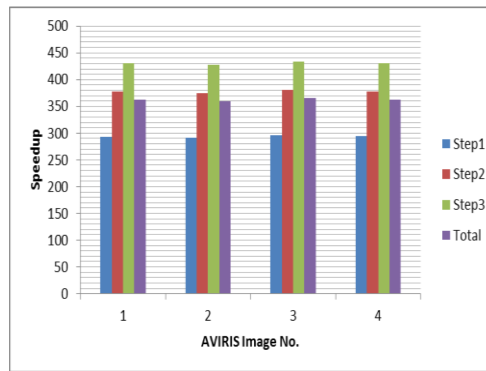

Figure 6. The Speedup Profile of TSP-AST Using 4 GPU on Platform 2

Table 3 reports the obtained results in terms of computation times and speedups measured after comparing the parallel implementations of TSP-AST (TSP-AST-M and TSP-AST-P) with the equivalent serial version for the four considered hyperspectral images. The TSP-AST-P achieved significant speedups, with regard to the serial version TSP-AST-S and also with regard to TSP-AST-M, on both platforms. The data transfers between host and device were obtained after profiling the implementation using the CUDA visual profiler tool distributed by Nvidia [26]. In all cases, less than 34 percent of the total solver time is consumed by data transfers. Taking into consideration that the AVIRIS scanning rate is 12 Hz [27], more satellite hyperspectral sensors such as Hyperion feature 220 Hz cross-line scanning rates. This means that a hyperspectral sensor data like the AVIRIS Cuprite scene could be collected in about 5s. Therefore, in the case of using the platform 2, TSP-AST-P can compress the image faster than the data being acquired, thus achieving real-time computing performance.

Table 3. Execution Time of the Serial (TSP-AST-S), Multicores (TSP-AST-M), and GPU (TSP-AST-P) Implementations

Platform 1			Cuprite	Jasper Ridge	Lunar Lake	Moffett Field
Times (sec)	TSP-AST-S		1357.41	1370.76	1406.63	1361.95
	TSP-AST-M		637.32	640.54	651.21	639.47
	TSP-AST-P	Total	16.96	17.11	17.54	17.01
		IO(Host to Device)	0.9356 (5.52%)	0.9364 (5.47%)	0.9361 (5.34%)	0.9397 (5.52%)
		IO(Device to Host)	0.4068 (2.40%)	0.4075 (2.38%)	0.4072 (2.32%)	0.4098 (2.41%)
Speed Up(X)	TSP-AST-M/TSP-AST-P		37.58	37.44	37.13	37.59
	TSP-AST-S/TSP-AST-P		80.04	80.11	80.20	80.07
Platform 2			Cuprite	Jasper Ridge	Lunar Lake	Moffett Field
	TSP-AST-S		1351.32	1364.19	1394.86	1358.64
	TSP-AST-M		259.37	263.84	275.73	261.78
	TSP-AST-P	Total	3.73	3.78	3.82	3.75
		IO(Host to Device)	0.9071 (24.32%)	0.9178 24.28%)	0.9298 (24.34%)	0.9154 (24.41%)
		IO(Device to Host)	0.3526 (9.45%)	0.3597 (9.52%)	0.3503 (9.17%)	0.3497 (9.33%)
Speed Up(X)	TSP-AST-M/TSP-AST-P		69.54	69.80	72.18	69.81
	TSP-AST-S/TSP-AST-P		362.28	360.90	365.15	362.30

5. Conclusion

While the three-stage prediction with adaptive search threshold method (TSP-AST) achieves excellent compression performance, it is relatively slow due to several matrix operations, bi-directional pixel search, and backward pixel search with adaptive search

Copyright ⓒ 2017 SERSC

threshold. All these operations must be performed on each pixel that to be predicted. In this paper, a new parallel implementation of the TSP-AST algorithm for hyperspectral image compression has been presented using the CUDA. Experimental results show the effectiveness of the proposed GPU implementation, not only in terms of compression performance but also in terms of computing performance. The implementation achieved significant speedups compared to the serial and multicore versions, which are encouraging to provide this effective and efficient compression solution for hyperspectral compression in real time.

Acknowledgments

This work is supported by the Research Foundation of the Sichuan Department of Education (15ZB0044), and the Research Foundation of Sichuan Normal University (2015KYQD312).

References

[1] P. M. Mather, "Computer processing of remotely-sensed images," 2nd ed. Chichester, U. K.: Wiley, **(1999)**.
[2] D. Taubman and M. Marcellin, "JPEG2000: Standard for interactive imaging," Proc. IEEE, vol. 90, **(2002)**, pp. 1336–1357.
[3] M. Weinberger, G. Seroussi, and G. Sapiro, "The LOCO-I lossless image compression algorithm: Principles and standardization into JPEG-LS," IEEE Trans. Image Process., vol. 9, no. 8, **(2000)**, pp. 1309–1324, 2000.
[4] J. Mielikainen, "Lossless compression of hyperspectral images using lookup tables," IEEE Signal Process. Lett., vol. 13, no. 3, **(2006)**, pp. 157–160.
[5] B. Huang and Y. Sriraja, "Lossless compression of hyperspectral imagery via lookup tables with predictor selection," in Proc. SPIE, vol. 6365, **(2006)**, pp. 63650L-1–63650L-8.
[6] A. B. Kiely and M. A. Klimesh, "Exploiting calibration-induced artifacts in lossless compression of hyperspectral imagery," IEEE Trans. Geosci. Remote Sensing, vol. 47, no. 8, **(2009)**, pp. 2672–2678.
[7] J. Mielikainen and P. Toivanen, "Clustered DPCM for the lossless compression of hyperspectral images," IEEE Trans. Geosci. Remote Sensing, vol. 41, no. 12, **(2003)**, pp. 2943–2946.
[8] B. Aiazzi, L. Alparone, S. Baronti, and C. Lastri, "Crisp and fuzzy adaptive spectral predictions for lossless and near- lossless compression of hyperspectral imagery," IEEE Trans. Geosci. Remote Sensing Lett., vol. 4, no. 4, **(2007)**, pp. 532–536.
[9] J. Mielikainen and B. Huang, "Lossless compression of hyperspectral images using clustered linear prediction with adaptive prediction length," IEEE Trans. Geosci. Remote Sensing Lett., vol. 9, no. 6, **(2012)**, pp. 1118–1121.
[10] C. G. Li, and K. Guo, "Lossless compression of hyperspectral images using three-stage prediction with adaptive search threshold," international journal of signal processing, image processing and pattern recognition, vol. 7, no. 3, **(2014)**, pp. 305-316.
[11] "Nvidia Cuda Programming Guide," ver. 7.5, **(2015)**.
[12] E. Christophe, J. Michel, and J. Inglada, "Remote sensing processing: from multicore to GPU," IEEE J. Sel. Topics Appl. Earch Observ. Remote Sens., vol. 4, no. 3, **(2011)**, pp. 643-652.
[13] S. Bernabe, S. Lopez, A. Plaza, and R. Sarmiento, "GPU implementation of an automatic target detection and classification algorithm for hyperspectral image analysis," IEEE Trans. Geosci. Remote Sens. Lett., vol. 10, no. 2, **(2013)**, pp. 221-225.
[14] J. Mielikainen, B. Huang, H. –L. A. Huang, and M. D. Goldberg, "GPU implementation of stony brook university 5-class cloud microphysics scheme in the WRF," IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens., vol. 5, no. 2, **(2012)**, pp. 625-632.
[15] J. Mielikainen, B. Huang, H. –L. A. Huang, and M. D. Goldberg, "GPU acceleration of the updated goddard shortwave radiation scheme in the weather research and forecasting (WRF) model," IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens., vol. 5, no. 2, **(2012)**, pp. 555-562.
[16] A. Agathos, L. Jun, D. Petcu, and A. Plaza, "Multi-GPU implementation of the minimum volume simplex analysis algorithm for hyperspectral unmixing," IEEE J. Sel. Topics Appl. Earch Observ. Remote Sens., vol. 7, no. 6, **(2014)**, pp. 2281-2296.
[17] Z. B. Wu, J. F. Liu, A. Plaza, J. Li, and Z. H. Wei, "GPU implementation of composite kernels for hyperspectral image classification," IEEE Trans. Geosci. Remote Sens. Lett., vol. 12, no. 9, **(2015)**, pp. 1972-1977.
[18] J. Mielikainen, R. Honkanen, B. Huang, P. Toivanen, and C. Lee, "Constant coefficients linear prediction for lossless compression of ultraspectral sounder data using a graphics processing unit," J. Appl. Remote Sens., vol. 4, no. 1, **(2010)**, pp. 041774.

Copyright ⓒ 2017 SERSC

[19] S. –C. Wei and B. Huang, "GPU acceleration of predictive partitioned vector quantization for ultraspectral sounder data compression," IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens., vol. 4, no. 3, **(2011)**, pp. 677-682.
[20] L. Santos, E. Magli, R. Vitulli, J. F. Lopez, and R. Sarmiento, "Highly-parallel GPU architecture for lossy hyperspectral image compression," IEEE J. Sel. Topics Appl. Earch Observ. Remote Sens., vol. 6, no. 2, **(2013)**, pp. 670-681.
[21] Y. Dai, Y. Fang, D. He, and B. Huang, "Parallel design for error-resilient entropy coding algorithm on GPU," J. Parallel Distrib. Comput. vol. 73, **(2013)**, pp. 411-419.
[22] NVIDIA Developer Zone. **(2014)**. cuBLAS User Guide [Online]. Available: http://docs.nvidia.com/cuda/cublas/index.html
[23] EM Photonics. **(2014)**. CULA Programmer's Guide [Online]. Available: http://www.culatools.com/cula_dense_programmers_guide/
[24] Z. B. Wu, Q. C. Wang, A. Plaza, J. Li, J. J. Liu, and Z. H. Wei, "Parallel implementation of sparse representation classifiers for hyperspectral imagery on GPUs", IEEE J. Sel. Topics Appl. Earch Observ. Remote Sens., vol. 8, no. 6, **(2015)**, pp. 2912-2925.
[25] B. Chapman, G. Jost, and R. Pas, Using OpenMO: Portable shared memory parallel programming. Cambridge, MA, USA: MIT Press, **(2007)**.
[26] NVIDIA Developer Zone. Profiler User's Guide [Online]. Available: http://www.nvidia.com/cuda/profiler-users-guide/#axzz3K7S7Wk7G, **(2014)**.
[27] X. Wu, B. Huang, A. Plaza, Y. Li, and C. Wu, "Real-time implementation of the pixel purity index algorithm for endmenber identification on GPUs," IEEE Geoscience and Remote Sensing Letters, vol. 11, no. 5, **(2013)**, pp. 955-959.

 Copyright ⓒ 2017 SERSC