Parallel Implementation of the Recursive Least Square for Hyperspectral Image Compression on GPUs

Changguo Li
College of Fundamental Education, Sichuan Normal University
Chengdu, 610068, China
[e-mail: 389224879@qq.com]
*Corresponding author: Changguo Li

Received November 3, 2016; revised March 15, 2017; accepted March 25, 2017; published July 31, 2017

Abstract

Compression is a very important technique for remotely sensed hyperspectral images. The lossless compression based on the recursive least square (RLS), which eliminates hyperspectral images' redundancy using both spatial and spectral correlations, is an extremely powerful tool for this purpose, but the relatively high computational complexity limits its application to time-critical scenarios. In order to improve the computational efficiency of the algorithm, we optimize its serial version and develop a new parallel implementation on graphics processing units (GPUs). Namely, an optimized recursive least square based on optimal number of prediction bands is introduced firstly. Then we use this approach as a case study to illustrate the advantages and potential challenges of applying GPU parallel optimization principles to the considered problem. The proposed parallel method properly exploits the low-level architecture of GPUs and has been carried out using the compute unified device architecture (CUDA). The GPU parallel implementation is compared with the serial implementation on CPU. Experimental results indicate remarkable acceleration factors and real-time performance, while retaining exactly the same bit rate with regard to the serial version of the compressor.

Keywords: Graphics Processing Units (GPUs), compute Unified Device Architecture (CUDA), recursive least square (RLS), parallel implementation, hyperspectral compression

This work was supported by the National Natural Science Foundation of China (Grant No. 61373162), the Research Foundation of the Sichuan Department of Education (Grant No. 15ZB0044), and the Research Foundation of Sichuan Normal University (Grant No. 2015KYQD312).

1. Introduction

Hyperspectral image compression is an active research topic in remote sensing [1]. It is generally known that hyperspectral instruments acquire images in hundreds of narrow and continuous spectral bands. Moreover, the data volume of hyperspectral images has been drastically increased with the growing scientific and technological demands in spatial and spectral resolutions, which poses a significant challenge to data transmission and storage. Therefore, there is an increasing need for highly performing image compression techniques. Typically, hyperspectral image compression techniques are classified into three modalities: lossless, lossy, and near-lossless. However, the last two techniques are unacceptable in many of the corresponding applications such as target detection, classification, object identification, and automatic feature extraction. As a result, only lossless compression allows for reconstructing the original image perfectly.

Among these three compression methods, lossless compression has received a lot of interest [2-5]. Taking into consideration that the Adaptive filter can automatically adjust its parameters and achieve the optimal filtering when the input signal and noise statistical characteristics are unknown or change, as a technique for the efficient compression of the original hyperspectral image with less statistical properties, the adaptive filter has been widely used [4-6]. In [6], a novel algorithm for lossless compression of hyperspectral imagery based on the recursive least square (RLS) was proposed. This compressor calculates the local difference between the local mean of four neighbor of the current pixel and the current pixel, and the local differences of the pixels which co-locate with the current pixel in previous bands form the input vector of the RLS. The bit rate of this algorithm is comparable or superior to that exhibited by many other state-of-the-art techniques. However, its computational complexity was shown to be relatively high, thus limiting its application in time-critical scenarios. The reason is not only the extremely high dimensionality of hyperspectral data, but also that the RLS filter refers to multiple loop iterations of variables for each pixel. This results in a computational complexity that is even higher than fast lossless (FL)-based compression [4].

Recent advances in high-performance computing have opened new avenues to overcome the aforementioned computational challenges [7-10]. These high-performance computing technologies such as Beowulf clusters and distributed computers, multicore central processing units (CPUs), field programmable gate arrays (FPGAs), and graphics processing units (GPUs), can be used to accelerate hyperspectral image processing algorithms so as to make them suitable for time-critical scenarios. In the above mentioned technologies, GPUs have recently emerged as a commodity platform for many compute-intensive, massively parallel, and data-intensive computations. GPU-based parallel computing offers a tremendous potential to bridge the gap toward real-time compression of hyperspectral images. In [11], a spectral image data compression method called Linear Prediction with Constant Coefficients (LP-CC) using NVIDIA's CUDA parallel computing architecture was implemented, which achieves a speedup of 86 compared to a single threaded CPU version. In [12], two most time consuming stages of linear prediction and vector quantization were chosen for GPU-based implementation. By exploiting the data parallel characteristics of these two stages, a spatial division design showed a speedup of 72x. In [13], the GPU implementation of an algorithm for onboard lossy hyperspectral image compression was described, and an architecture that allows to accelerate the compression task by parallelizing it on the GPU was proposed. In [14], a partitioning error-resilient entropy coding (P-EREC) algorithm, which splits variable-length blocks into groups and then every group is coded using the EREC separately, was proposed. Each GPU thread processed one group so as to make the EREC coarse-grained parallel. 32x to 123x speedup over the original C code of EREC was achieved. In [15] and [16], the CUDA toolkit based on GPU was used to design the paprallel algorithms of the all phase biorthogonal transform with JPEG scheme (APBT-JPEG) and the all phase discrete sine biorthogonal transform with JPEG scheme (APDSBT-JPEG), and their maximum speedup ratios all reached more than 100 times. However, to the best of our knowledge, and despite the importance of RLS filter methods in the hyperspectral compression, there are no available GPU implementations for this category of algorithms in the literature.

In this paper, we develop a new parallel RLS method for hyperspectral image compression on GPUs. First, we introduce an optimized RLS model based on optimal number of prediction bands. The method promotes the bit rate by spreading the spectral information from the current pixel to its neighbors until achieving a global stable state on the whole image. Then this optimized RLS method is used as a case study to illustrate the advantages and potential challenges of utilizing GPU parallel computing principles to improve the computation speed of the proposed approach. The proposed implementation accelerates intensive computations and operations involving large data sets on the GPU by utilizing NVidia's compute unified device architecture (CUDA), executing the rest of the operations (mostly related with control) on the CPU. The performance of the proposed GPU-based parallel implementation is assessed using real hyperspectral images and compared with the CPU-based serial implementation. The remainder of this paper is organized as follows. Section II briefly describes the hyperspectral images compression using RLS filter. Section III presents its parallel implementation. Experimental results are reported in Section IV. Conclusions with some remarks and hints at plausible future research lines are given in Section V.

2. Optimized Recursive Least Square Method Based on Optimal Number of Prediction Bands

2.1 Recursive Least Square (RLS)

As an adaptive filtering algorithm, the basic idea of RLS is that given the least square estimation of the filter weight vector at time n-1, the iterative method is used to calculate the least squares estimation of the filter weight vector at time n. For the above reason, the applications of RLS have drawn wide attention in recent years. In [6], it has been proved to be an extremely powerful compression tool for hyperspectral image, which has strong correlations on both spectral and spatial dimensions, and leads to the state-of-the-art performance. To define the problem in mathematical terms in compression stage, let represent the current pixel, where x and y are the coordinates of the current pixel in the current band, W and H are the image's width and height, and $t = Wy + x$. For the first band, the intraband estimate of pixel $s_z(t)$ ($\tilde{s}_z(t)$) is given as follows:

$$\tilde{s}_z(t) = (s_z^W(t) + s_z^{NW}(t) + s_z^N(t) + s_z^{NE}(t)) / 4 \tag{1}$$

$$d_z(t) = s_z(t) - \tilde{s}_z(t) \tag{2}$$

where $s_z^W(t)$, $s_z^{NW}(t)$, $s_z^N(t)$, *and* $s_z^{NE}(t)$ denote four neighboring pixels, respectively. For the other bands, the RLS filter is adopted to conduct the interband prediction by the following model:

$$e_z(t) = d_z(t) - \left\lfloor \boldsymbol{d}_z(t)\boldsymbol{w}^T(t-1) \right\rfloor \tag{3}$$

where the input vector $\boldsymbol{d}_z(t) = \left[d_{z-1}(t), d_{z-2}(t), \cdots, d_{z-p}(t)\right]$, the weight vector $\boldsymbol{w}(t) = \left[w_1(t), \cdots w_p(t)\right]$, p is the number of prediction bands (prediction bands are previous bands of the current band, which are used to predict the current band), w(0)=[0], and t=1. The gain vector k(t), the inverse correlation matrix P(t), and the weight vector w(t) are updated as follows:

$$k^T(t) = \frac{P(t-1)\boldsymbol{d}_z^T(t)}{1 + \boldsymbol{d}_z(t)P(t-1)\boldsymbol{d}_z^T(t)} \tag{4}$$

$$P(t) = P(t-1) - k^T(t)\boldsymbol{d}_z(t)P(t-1) \tag{5}$$

$$\boldsymbol{w}(t) = \boldsymbol{w}(t-1) + k(t)e_z(t) \tag{6}$$

where $P(0) = \delta I_P$, $\delta = 0.0001$, I_p is the p-order identity matrix. After the prediction residual is calculated, it is entropy-coded using an adaptive arithmetic coder (AAC). Then t=t+1, and the next pixel is executed the same interband prediction. When t is larger than W×H, the procedure of current band prediction is ended.

2.2 Optimized RLS Based on Optimal Number of Prediction Bands

In hyperspectral images, spectral correlations are usually stronger than spatial correlations. It is expected that compression methods based on spectral correlation can work well. Previous works indicate that by taking advantage of the spectral correlation, bit rate can be significantly improved [2-5], and on the basis, prediction bands are further explored to optimize the compression results [17-19].

Table 1. Portion of compression results (bits in per pixel) as a function of the number of prediction bands

Number of bands	**5**	**10**	**15**	**20**	**25**	**30**	**35**	**40**	**45**	**50**
YSC-0	3.82	3.77	3.74	3.74	3.74	3.70	3.70	3.70	3.72	3.75
YSC-3	3.68	3.64	3.62	3.61	3.61	3.58	3.58	3.58	3.61	3.65
YSC-10	3.31	3.26	3.22	3.23	3.23	3.18	3.18	3.18	3.19	3.23
YSC-11	3.54	3.47	3.43	3.43	3.43	3.38	3.39	3.39	3.40	3.45
YSC-18	3.72	3.67	3.65	3.65	3.65	3.62	3.62	3.62	3.64	3.67
Average-YSC	3.61	3.56	3.53	3.53	3.53	3.49	3.49	3.49	3.51	3.55
YSU-0	6.05	5.99	5.95	5.95	5.95	5.91	5.91	5.91	5.93	5.96
YSU-3	5.88	5.82	5.79	5.80	5.80	5.75	5.75	5.75	5.76	5.78
YSU-10	5.51	5.48	5.45	5.44	5.44	5.41	5.40	5.40	5.43	5.47
YSU-11	5.74	5.67	5.63	5.62	5.62	5.58	5.58	5.58	5.61	5.65
YSU-18	5.94	5.90	5.86	5.85	5.85	5.82	5.82	5.82	5.83	5.86
Average-YSU	5.82	5.77	5.73	5.73	5.73	5.69	5.69	5.69	5.71	5.74
MU-10	2.69	2.64	2.61	2.61	2.61	2.57	2.57	2.57	2.58	2.60
HU-1	2.57	2.52	2.50	2.50	2.50	2.47	2.47	2.47	2.50	2.53
Average-MU&HU	2.63	2.58	2.56	2.56	2.56	2.52	2.52	2.52	2.54	2.57

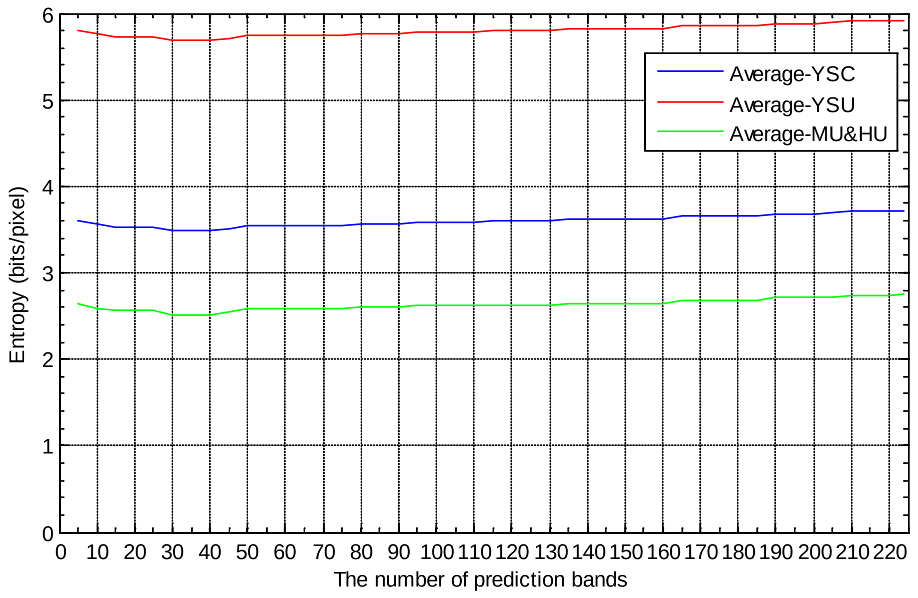


Fig. 1. Compression results using different number of prediction bands.

Since the spatial location of the current pixel is known, this aspect can be included in the RLS model to retain the spectral information of the current pixel and spread them to neighboring pixels. Thus, the bit rate of the RLS model can be further improved by spreading the spectral information from the current pixel to its neighbors until achieving a global stable state on the whole image. In order to obtain the optimal number of prediction bands for RLS, we varied the number of prediction bands from 5 to 224 using a stride value of 5 in the first test. For test data, we used NASA's AVIRIS'06 images (details of the dataset will be given in Section IV). The results from those tests are quantitatively shown in **Table 1** and **Fig. 1**. In this table, the number of prediction bands is shown in row 1. Rows 2-16 show results using twelve scene images. For this test purpose, we are only interested in finding the optimal value for the number of prediction bands p. As can be seen from **Table 1** and **Fig. 1**, the average compression bits-per-pixel reaches its minimum at 30 prediction bands. Thus, we take 30 as the optimal number of prediction bands for compression calculation in this paper, and then can obtain the lowest bit rate, which means that the RLS model can achieve the optimal compression results by spreading the spectral information from the current band to its optimal prediction bands. To sum up, the optimized RLS model based on optimal prediction bands (referred to hereinafter as RLS-OPB), which exploits both spatial and spectral information in the compression stage, is described in **Fig. 2**.

```
Serial Algorithm of RLS-OPB (referred to hereinafter as RLS-OPB-S)
Input: HSI images $R^{Z\times W\times H}$, number of bands Z, width of image W, and height of image H, pixel
sample $s_z(t) \in R^{Z\times W\times H}$;
Initialization: Set z=2, t=1, $\delta = 0.0001$, p=30, initialize Ip equals the p-order identity matrix, the
prediction error E=R, w(0)=[0], P(0)=$\delta I_p$;
Do:
  Do:
    Step 1. Calculate $d_z(t)$ and $d_{z-i}(t)$, form the input vector
    $\mathbf{d}_z(t) = [d_{z-1}(t), d_{z-2}(t), \ldots, d_{z-p}(t)]$, where i=1,…,p;
    Step 2. $e_z(t) = d_z(t) - \lfloor \mathbf{d}_z(t)\mathbf{w}^T(t-1) \rfloor$;
    Step 3. $k^T(t) = \frac{P(t-1)\mathbf{d}_z^T(t)}{1 + \mathbf{d}_z(t)P(t-1)\mathbf{d}_z^T(t)}$;
    Step 4. $P(t) = P(t-1) - k^T(t)\mathbf{d}_z(t)P(t-1)$;
    Step 5. $\mathbf{w}(t) = \mathbf{w}(t-1) + k(t)e_z(t)$;
    Step 6. t=t+1;
  While $t \leq W \times H$
  Step 7. z=z+1;
While $z \leq Z$
Step 8. the prediction error E is entropy-coded using AAC;
Output: the compressed bit-stream of E.
End
```

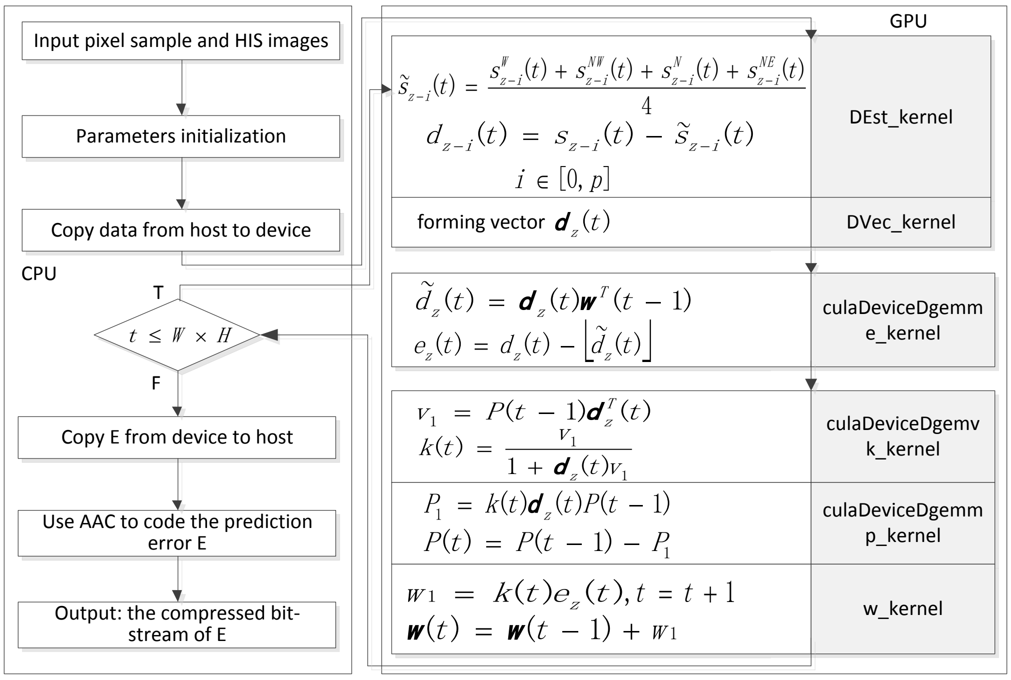
Fig. 2. Serial algorithm of RLS-OPB (referred to hereinafter as RLS-OPB-S).

3. Algorithm Optimization for RLS-OPB on GPUs

The computing system consists of a host that is a traditional CPU and devotes more resources to caching or control flow operations, while GPUs can be regarded as computer devices or coprocessors. GPUs are single-instruction multiple-data (SIMD) parallel devices. Therefore, we can take data-parallel computing of computationally intensive portions of the algorithm or application on GPUs [10]. 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 [20] and CULA [21]. However, the latter usually achieves better performance than the former for matrix multiplication [22]. Thus, CULA is chosen to realize the main matrix operations in this paper.

With the aforementioned issues in mind, to further optimize the RLS-OPB algorithm on GPUs, it is now necessary to deeply analyze the serial algorithm of RLS-OPB-S. The inputs to the algorithm are the hyperspectral images $R^{Z \times W \times H}$, the inverse correlation matrix P(0), and the weight vector w(0). Because of referring to matrix multiplication, matrix subtraction, and matrix addition, every read-write and arithmetic type operation of the algorithm is very time consuming for the higher dimensional data (e.g., the images $R^{Z \times W \times H}$, P(t), and w(t)). Meanwhile, some iteration steps such as the gain vector k(t), the inverse correlation matrix P(t), and the weight vector w(t) increases the computational burden even more. On the basis of the foregoing, the key stage of RLS-OPB-S is the loop iterative solution of vector k(t), matrix P(t) and vector w(t). Since there are dependencies among these iteration steps, it is hard to parallelize the algorithm as a whole.

In order to improve the computing performance of RLS-OPB, a GPU-based parallel hyperspectral compression algorithm (RLS-OPB-P) has been developed according to the following optimization principles: 1) at the beginning, we reconstruct the iteration steps with tight coupling; 2) then we parallel the iteration steps with loose coupling; 3) finally, the parallelization at kernel level is realized by CUDA streams. The detailed flowchart of the parallel algorithm is given in **Fig. 3**. In the following, we describe the most relevant steps of the parallelization accomplished by the proposed RLS-OPB-P algorithm.



GPU
Input pixel sample and HIS images
Parameters initialization
Copy data from host to device
CPU
T
$t \le W \times H$
F
Copy E from device to host
Use AAC to code the prediction error E
Output: the compressed bit-stream of E
$\tilde{s}_{z-i}(t) = \frac{s^{W}_{z-i}(t) + s^{NW}_{z-i}(t) + s^{N}_{z-i}(t) + s^{NE}_{z-i}(t)}{4}$
$d_{z-i}(t) = s_{z-i}(t) - \tilde{s}_{z-i}(t)$
$i \in [0, p]$
DEst_kernel
forming vector $\boldsymbol{d}_z(t)$
DVec_kernel
$\tilde{d}_z(t) = \boldsymbol{d}_z(t)\boldsymbol{w}^T(t-1)$
$e_z(t) = d_z(t) - \lfloor \tilde{d}_z(t) \rfloor$
culaDeviceDgemm e_kernel
$v_1 = P(t-1)\boldsymbol{d}_z^T(t)$
$k(t) = \frac{v_1}{1 + \boldsymbol{d}_z(t)v_1}$
culaDeviceDgemv k_kernel
$P_1 = k(t)\boldsymbol{d}_z(t)P(t-1)$
$P(t) = P(t-1) - P_1$
culaDeviceDgemm p_kernel
$w_1 = k(t)e_z(t), t = t+1$
$\boldsymbol{w}(t) = \boldsymbol{w}(t-1) + w_1$
w_kernel

Fig. 3. GPU implementation of RLS-OPB for hyperspectral compression

For Step 1 in **Fig. 2**, the terms $\tilde{s}_{z-i}(t)$ and $d_{z-i}(t)$ can be calculated firstly, then these elements will be used to form the input vector $d_z(t)$, as this element remain unchanged in each loop. A kernel function called DEst_kernel is defined to carry out the operations $\tilde{s}_{z-i}(t) = (s^{W}_{z-i}(t) + s^{NW}_{z-i}(t) + s^{N}_{z-i}(t) + s^{NE}_{z-i}(t))/4$, and $d_{z-i}(t) = s_{z-i}(t) - \tilde{s}_{z-i}(t)$. The kernel function DVec_kernel in Fig. 3 is implemented to form the input vector $\boldsymbol{d}_z(t)$. Since the size of each hyperspectral image is W×H, we start a J×L thread grid on the GPU, and each thread takes charge of the calculation for one pixel element, which makes the parallelization be maximized.

Both the number of computing threads in every block (denoted as THREAD_SIZE in this paper) and the number of blocks in every grid (denoted as BLOCK_SIZE in this paper) are crucial issues, since they plays an important role on the processing performance. In general, the latency of accessing and the occupancy of GPU depend on the number of the active warps in each stream multiprocessor (SM). So the device occupancy should be improved as high as possible to make sure that there are more threads executing on every SM. However, the hardware capabilities (such as shared memories, registers, and so on) usually limit the number of threads concurrently executed on an SM. Meanwhile, higher occupancy does not mean better performance [23]. Therefore, we chose the THREAD_SIZE and the BLOCK_SIZE according to the computing capabilities of NVIDIA Kepler GTX690 (details of the GPU will be given in Section IV). Namely, they are set to 32×32 and ((J+32-1)/32)×((L+32-1)/32), respectively, in order to make the GPU perform the best.

For Step 2 in Fig. 2, because the main operation $\boldsymbol{d}_z(t)\boldsymbol{w}^T(t-1)$ is related with matrix multiplication, function culaDeviceDgemm in CULA is first invoked to compute it on the GPU. Then the rest of the calculation for $e_z(t)$ is implemented by a kernel function e_kernel, which conducts the mapping between thread and pixel, and each thread is responsible for the calculation related to one pixel. To minimize the latency of global memory access, $e_z(t)$ is stored on GPU global memory, and the temporary variable $\tilde{d}_z(t)$ is stored on shared memory. As a consequence, the iterative steps of tight coupling are combined together and the parallelization of the function level inside the iteration is realized by these optimizations.

For steps 3 and 4, we decompose the calculation of k(t) and P(t) into $v_1 = P(t-1)\boldsymbol{d}_z^T(t)$ and $k(t) = v_1/(1+\boldsymbol{d}_z(t)v_1)$, $P_1 = k(t)\boldsymbol{d}_z(t)P(t-1)$ and $P(t) = P(t-1) - P_1$, respectively. The operations $k(t) = v_1/(1+\boldsymbol{d}_z(t)v_1)$ and $P(t) = P(t-1) - P_1$ are implemented by their corresponding kernel functions: k_kernel and p_kernel, in which the computational data are transferred to shared memory to process. We allocate k(t) and P(t) on GPU global memory, and organize their grid size according to the size of $d_z(t)$ and P(t). The rest of the computations are matrix-vector and matrix multiplications, which can be efficiently performed by the function culaDeviceDgemv and culaDeviceDgemm, respectively.

For step 5, bearing in mind that $k_1(t)$ and w(t) are of the same size, we can allocate them on GPU global memory, and encapsulate them together in a kernel function w_kernel, where a grid is created according to the size of w(t), and each thread is responsible for the calculation related to one element.

Finally, the results are copied from device (GPU) to the host (CPU) for analyzing the loop termination condition. Once the stopping condition is satisfied, we perform the memory copy of E from device to host, and the AAC is executed on the CPU.

With the aforementioned observations in mind, a detailed step-by-step algorithm description of the parallel hyperspectral compression algorithm based on RLS-OPB on GPU is summarized in Fig. 4.

```
Parallel Hyperspectral Compression Algorithm Based on RLS-OPB on GPU (RLS-OPB-P)
Input: HSI images $R^{Z\times W\times H}$, number of bands Z, width of image W, and height of image H, pixel sample $s_z(t) \in R^{Z\times W\times H}$;
Initialization: Set z=2, t=1, $\delta = 0.0001$, p=30, initialize Ip equals the p-order identity matrix, the prediction error E=R, w(0)=[0], P(0)=$\delta I_p$;
Step 1. Copy data from host to device
Do:
    Step 2. Calculate dz(t) on GPU
    Invoke DEst_kernel to compute $\tilde{s}_{z-i}(t) = (s^W_{z-i}(t) + s^{NW}_{z-i}(t) + s^N_{z-i}(t) + s^{NE}_{z-i}(t))/4$ and $d_{z-i}(t) = s_{z-i}(t) - \tilde{s}_{z-i}(t)$
    Invoke DVec_kernel to form dz(t)
    Step 3. Calculate ez(t) on GPU
    Invoke culaDeviceDgemm to compute $\tilde{d}_z(t) = \mathbf{d}_z(t)\mathbf{w}^T(t-1)$
    Invoke e_kernel to compute $e_z(t) = d_z(t) - \lfloor \tilde{d}_z(t) \rfloor$
    Step 4. Calculate k(t) on GPU
    Invoke culaDeviceDgemv to compute $v_1 = P(t-1)\mathbf{d}_z^T(t)$
    Invoke k_kernel to compute $k(t) = v_1 / (1 + \mathbf{d}_z(t)v_1)$
    Step 5. Calculate P(t) on GPU
    Invoke culaDeviceDgemm to compute $P_1 = k(t)\mathbf{d}_z(t)P(t-1)$
    Invoke p_kernel to compute $P(t) = P(t-1) - P_1$
    Step 6. Invoke w_kernel to calculate $w_1 = k(t)e_z(t)$ and $w(t) = w(t-1) + w_1$ on GPU
    Step 7. t=t+1
While $t \le W \times H$
Step 8. Copy E from device to host
Step 9. E is entropy-coded using AAC on CPU;
Output: the compressed bit-stream of E.
End
```

Fig. 4. Parallel algorithm of RLS-OPB (referred to hereinafter as RLS-OPB-P).

4. Experimental Results

The proposed compression algorithm is assessed using the hyperspectral images produced by an airborne visible/infrared imaging spectrometer (AVIRIS) [24], publicly available for download. All images include 224 bands, each of which is 16 bits per pixel (bpp), except for Hawaii and Maine, which is 12 bpp. Uncalibrated images are stored as unsigned integers, whereas calibrated images are stored as signed integers. The details of these images are given in **Table 2**.

Table 2. Detalls of experimental hyperspectral images used in our experiments

Name	Acronym	Size(W×H)	Formation	Scenes
Yellow Stone Calibrated	YSC	677×512	Signed 16 bit	0,3,10,11,18
Yellow Stone Uncalibrated	YSU	680×512	Unsigned 16 bit	0,3,10,11,18
Maine Uncalibrated	MU	680×512	Unsigned 12 bit	10
Hawaii Uncalibrated	HU	614×512	Unsigned 12 bit	1

The experimental platform used in our tests is the NVIDIA Kepler GTX 690, which consists of 8 stream multiprocesors (SM), and each SM has 192 stream processors (SP). The GPU is connected to one Intel Core i7-4790K CPU at 4.0 GHz with 4 cores and 16-GB RAM. The version of OpenMP and CUDA are 2.0 and 6.0, respectively. In order to demonstrate the performance improvements between the parallel implementations on multicore CPU platform and our considered GPU platform, a multicore implementation of RLS-OPB (RLS-OPB-M) has been developed following the design principles in [10] and using OpenMP Application Program Interface (API), which is adopted to explicitly address multithreaded and shared-memory parallelism. The corresponding serial version (RLS-OPB-S) is executed on one core of the Intel Core i7-4790K CPU, and the multicore version (RLS-OPB-M) is run on the four available cores of the Intel Core i7-4790K CPU. Meanwhile, Motivated by the GPU hardware characteristics, the parallel version (RLS-OPB-P) refers to the parallel implementation adopting the single GTX 690 by default, if not otherwise specified.

The first experiment was carried out to test the bit rate of the three implementations on the complete AVIRIS images. The compression results are quantitatively shown in **Table 3**, in which results for RLS are from [6]. The fact is that the proposed RLS-OPB-S, RLS-OPB-M, and RLS-OPB-P obtain exactly the same bit rate in terms of bits per pixel (BPP), and outperform the results obtained by RLS. At this point, it is important to emphasize that both the multicore and GPU parallel versions obtain exactly the same results as the serial implementation. This means that the three versions achieve exactly the same bit rate and the only difference among them is the computing performance.

Table 3. Compression results (bpp) for the complete AVIRIS images

Image	**RLS**	**RLS-OPB-S**	**RLS-OPB-M**	**RLS-OPB-P**	**RLS-PB-S/ RLS**
YSC-0	3.79	3.70	3.70	3.70	0.98
YSC-3	3.65	3.58	3.58	3.58	0.98
YSC-10	3.29	3.18	3.18	3.18	0.97
YSC-11	3.50	3.38	3.38	3.38	0.97
YSC-18	3.69	3.62	3.62	3.62	0.98
Average	3.58	3.49	3.49	3.49	0.98
YSU-0	6.01	5.91	5.91	5.91	0.98
YSU-3	5.85	5.75	5.75	5.75	0.98
YSU-10	5.49	5.41	5.41	5.41	0.99
YSU-11	5.70	5.58	5.58	5.58	0.98
YSU-18	5.91	5.82	5.82	5.82	0.98
Average	5.79	5.69	5.69	5.69	0.98
MU-10	2.66	2.57	2.57	2.57	0.97
HU-1	2.53	2.47	2.47	2.47	0.98
Average	2.60	2.52	2.52	2.52	0.98

Table 4. Execution times and acceleration factors obtained for the AVIRIS images

Image	**RLS-OPB-S Time (s)**	**RLS-OPB-M**		**RLS-OPB-P**	
		Time (s)	**Acceleration factor (X)**	**Time (s)**	**Acceleration factor (X)**
YSC-0	131.23	54.00	2.43	2.85	46.11
YSC-3	130.62	52.04	2.51	2.82	46.35
YSC-10	130.54	54.62	2.39	2.83	46.06
YSC-11	131.87	52.75	2.50	2.85	46.28
YSC-18	130.86	52.98	2.47	2.83	46.19
YSU-0	132.64	54.14	2.45	2.87	46.15
YSU-3	133.20	54.15	2.46	2.88	46.17
YSU-10	132.25	52.26	2.58	2.85	46.43
YSU-11	131.89	53.39	2.47	2.85	46.20
YSU-18	132.74	52.47	2.53	2.86	46.39
MU-10	132.15	52.44	2.52	2.85	46.37
HU-1	123.24	47.58	2.59	2.65	46.48

Table 5. Transfer times (s) of RLS-OPB-P

Image	**CPUTo GPU**	**GPUTo CPU**	**Total I/O**	**Total Execution**	**I/O Percentage**
YSC-0	0.42	0.33	0.75	2.85	26.23%
YSC-3	0.43	0.35	0.78	2.82	27.66%
YSC-10	0.42	0.32	0.74	2.83	26.15%
YSC-11	0.42	0.34	0.76	2.85	26.67%
YSC-18	0.43	0.31	0.74	2.83	25.14%
YSU-0	0.44	0.35	0.79	2.87	27.53%
YSU-3	0.44	0.36	0.80	2.88	27.78%
YSU-10	0.45	0.32	0.77	2.85	27.02%
YSU-11	0.44	0.33	0.77	2.85	27.02%
YSU-18	0.46	0.35	0.81	2.86	30.42%
MU-10	0.45	0.35	0.80	2.85	30.14%
HU-1	0.41	0.32	0.73	2.65	27.55%

Table 4 reports the obtained results in terms of computation times and speedups measured after comparing the parallel implementations of RLS-OPB (RLS-OPB-M and RLS-OPB-P) with the equivalent serial version for the considered images. The results show that the parallel implementation RLS-OPB-P achieves a remarkable acceleration factor of more than 46 relative to the serial version RLS-OPB-S. This is because the parallel version benefits from the efficient exploration of GPU parallel capacities and the utilization of the highly efficient GPU-accelerated linear algebra libraries of CUDA. Furthermore, the proposed parallel implementation of RLS-OPB-P can be completed in less than 3 s, including the loading times and the data transfer times from CPU and GPU and vice versa, as shown in **Table 5**. This represents a significant improvement with regard to both serial and multicore versions.

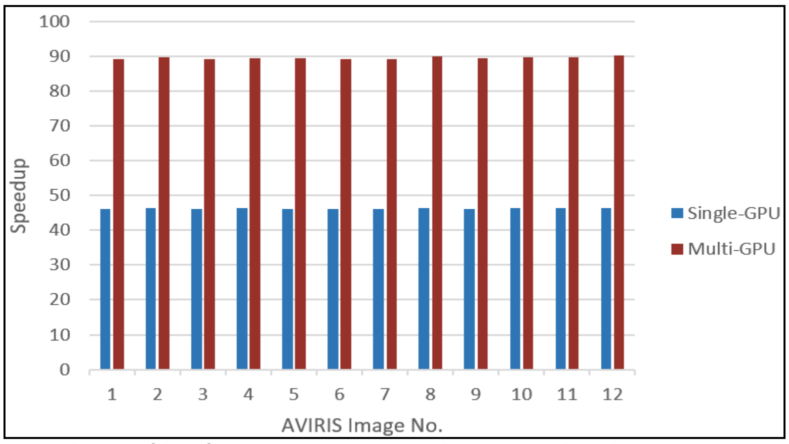


Fig. 5. The speedup profile of RLS-OPB using single-GPU and multi-GPU on 12 AVIRIS images

We now continue to pay attention to the computing performance. RLS-OPB-P, it is true, gets significant acceleration factors on AVIRIS images. Nevertheless other optimized strategies such as multi-GPU implementation and thread allocation can further boost the execution efficiency of CUDA kernels.

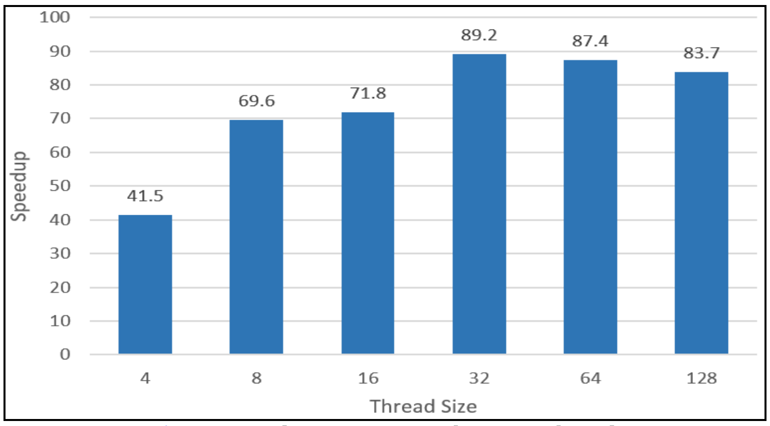


Fig. 6. Speedup comparison between thread size

GTX 690 has two GPUs. Once both GPUs are used, the execution time is further decreased. In order to compare performance between single-GPU and multi-GPU, the GPU-based parallel versions of RLS-OPB were run on the considered images, in which the proposed multi-GPU version refers to single-node desktop systems with multiple compute devices. We obtained different speedups for the AVIRIS images above on the different GPUs, as shown in Fig. 5. The results indicate that using one single GPU we have an average speedup of 46×. When there are more than one GPU available, the average compression time of the GPU-based RLS-OPB takes about 1.5 s or a speedup of 89× relative to RLS-OPB-S. The fact that using 2 GPUs does not have a total speedup near 2 can be attributed to the reason that each GPU is not assigned a job of equal workload. In spite of this, a general trend can be seen that using more GPUs does give higher speedup for RLS-OPB.

THREAD_SIZE can play an important role on the computing performance. Thread allocation determines not only the block size but also the grid size because these resources are dynamically partitioned and assigned to support their execution. Meanwhile, the latency of accessing and the occupancy of GPU depend on the number of the active warps in each SM. In order to verify the THREAD_SIZE allocation of our GPU parallel implementation. We tested the speedup for different thread sizes in processing one YSC-0 image (still using 2 GPUs). The result is comparatively reported in **Fig. 6**. As can be seen from **Fig. 6**, the THREAD_SIZE of 32×32 produces the best performance.

In order to compare performance across GPUs, we tested the speedup for images of different sizes on multiple NVIDIA GPUs. Specifically, the parallel version RLS-OPB-P was run on the Fermi-based GTX 690, which has 1536 cores; the Fermi-based GTX 570, which has 480 cores; and the Fermi-based GTX 280, which has 240 cores. The result is shown in **Fig. 7** —the Fermi-based GTX 690 produces the best performance.

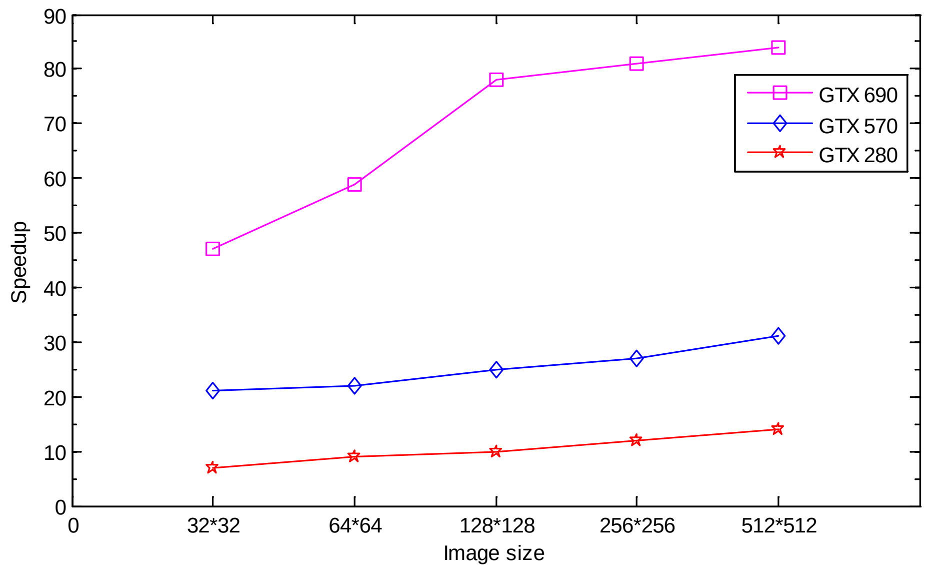


Fig. 7. Speedup for images of different sizes on the different GPUs

To conclude this section, we emphasize that (as shown in **Table 4** and **Fig. 5**) the average speedups of the single-GPU and multi-GPU GPU versions are more than 46× and 89× with regard to the serial version RLS-OPB-S, respectively. While the multicore version RLS-OPB-M obtains an average speedup of 2.3×. For the AVIRIS scene, the compression task can be completed in about 1.5 s. These are important advantages offered by GPU platforms in terms of being able to adapt computationally expensive compression problems such as those involved in RLS methods to time-critical scenarios.

5. Conclusion

In this paper, a novel parallel recursive least square compression method on GPUs has been proposed. First, an optimized recursive least square model is introduced based on optimal number of prediction bands, which improves the bit rate by spreading the spectral information from the current pixel to its neighbors until achieving a global stable state on the whole image. Further, a parallel implementation of the RLS approach for GPUs is developed using the NVIDIA CUDA. Experimental results on AVIRIS images show the effectiveness of the proposed GPU implementation, not only in terms of bit rate but also in terms of computing performance. Specially, when more than one GPU is available, the implementation achieves significant speedups compared to the serial and multicore versions, and provides an effective and efficient compression solution for AVIRIS images compression in real time. The future work will consider the I/O communications between the host and the device for further speedup.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant No. 61373162), the Research Foundation of the Sichuan Department of Education (Grant No. 15ZB0044), and the Research Foundation of Sichuan Normal University (Grant No. 2015KYQD312). The author thanks the anonymous reviewers and the editors for their valuable comments to improve the presentation of the paper.

References

[1] J. Wu, W. Kong, J. Mielikainen and B. Huang, "Lossless compression of hyperspectral imagery via clustered differential pulse code modulation with removal of local spectral outliers," *IEEE Signal Processing Letters*, vol. 22, no. 12, pp. 2194-2198, December, 2015. Article (CrossRef Link).

[2] J. Mielikainen, "Lossless compression of hyperspectral images using lookup tables," *IEEE Signal Processing Letters*, vol. 13, no. 3, pp. 157–160, March, 2006. Article (CrossRef Link).

[3] B. Huang and Y. Sriraja, "Lossless compression of hyperspectral imagery via lookup tables with predictor selection," in *Proc. of SPIE*, vol. 6365, pp. 63650L-1–63650L-8, September, 2006. Article (CrossRef Link).

[4] CCSDS, "Lossless multispectral and hyperspectral image compression," 123.0-B-1, CCSDS, 2012.

[5] C. C. Lin and Y. T. Hwang, "An efficient lossless compression scheme for hyperspectral images using two-stage prediction," *IEEE Geoscience and Remote Sensing Letters*, vol. 7, no. 3, pp. 558-562, July, 2010. Article (CrossRef Link).

[6] J. W. Song, Z. W. Zhang and X. M. Chen, "Lossless compression of hyperspectral imagery via RLS filter," *Electronics Letters*, vol. 49, no. 16, pp. 992-994, August, 2013. Article (CrossRef Link).

[7] A. Plaza, Q. Du, Y. L. Chang and R. L. King, "High performance computing for hyperspectral remote sensing," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 4, no. 3, pp. 528–544, September, 2011. Article (CrossRef Link).

[8] E. Christophe, J. Michel and J. Inglada, "Remote sensing processing: From multicore to GPU," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 4, no. 3, pp. 643–652, September, 2011. Article (CrossRef Link).

[9] C. Gonzalez, D. Mozos, J. Resano and A. Plaza, "FPGA implementation of the N-FINDR algorithm for remotely sensed hyperspectral image analysis," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 50, no. 2, pp. 374–388, February, 2012. Article (CrossRef Link).

[10] S. Bernabé, S. Sánchez, A. Plaza, S. López, J. A. Benediktsson and R. Sarmiento, "Hyperspectral unmixing on GPUs and multi-core processors: A comparison," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 6, no. 3, pp. 1386–1398, June, 2013. Article (CrossRef Link).

[11] 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," *Journal of Applied Remote Sensing*, vol. 4, no. 1, p. 041774, September, 2010. Article (CrossRef Link).

[12] S. C. Wei and B. Huang, "GPU acceleration of predictive partitioned vector quantization for ultraspectral sounder data compression," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 4, no. 3, pp. 677-682, September, 2011. Article (CrossRef Link).
[13] L. Santos, E. Magli, R. Vitulli, J. F. Lopez and R. Sarmiento, "Highly-parallel GPU architecture for lossy hyperspectral image compression," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 6, no. 2, pp. 670-681, April, 2013. Article (CrossRef Link).
[14] Y. Dai, Y. Fang, D. He and B. Huang, "Parallel design for error-resilient entropy coding algorithm on GPU," *Journal of Parallel and Distributed Computing*, vol. 73, no. 4, pp. 411-419, April, 2013. Article (CrossRef Link).
[15] C. Y. Wang, R. Y. Shan and X. Zhou, "APBT-JPEG image coding based on GPU," *KSII Transactions on Internet and Information Systems*, vol. 9, no. 4, pp. 1457-1470, April, 2015. Article (CrossRef Link).
[16] R. Y. Shan, X. Zhou, C. Y. Wang and B. C. Jiang, "All phase discrete sine biorthogonal transform and its application in JPEG-like image coding using GPU," *KSII Transactions on Internet and Information Systems*, vol. 10, no. 9, pp. 4467-4486, September, 2016. Article (CrossRef Link).
[17] C. F. Huo, R. Zhang and T. X. Peng, "Lossless compression of hyperspectral images based on searching optimal multibands for prediction," *IEEE Geoscience and Remote Sensing Letters*, vol. 6, no. 2, pp. 339-343, April, 2009. Article (CrossRef Link).
[18] J. Zhang and G. Z. Liu, "An efficient reordering prediction-based lossless compression algorithm for hyperspectral images," *IEEE Geoscience and Remote Sensing Letters*, vol. 4, no. 2, pp. 283-287, April, 2007. Article (CrossRef Link).
[19] J. Mielikainen and P. Toivanen, "Lossless compression of ultraspectral sounder data using linear prediction with constant coefficients," *IEEE Geoscience and Remote Sensing Letters*, vol. 6, no. 3, pp. 495-498, July, 2009. Article (CrossRef Link).
[20] NVIDIA Developer Zone, "CuBLAS user guide," January, 2015. [Online]. Available: http://docs.nvidia.com/cuda/cublas/index.html
[21] EM Photonics, "CULA Programmer's Guide," June, 2014. [Online]. Available: http://www.culatools.com/cula_dense_programmers_guide/
[22] 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 Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 8, no. 6, pp. 2912-2925, June, 2015. Article (CrossRef Link).
[23] F. Rob, CUDA Application Design and Development, Elsevier, Waltham, 2011. Article (CrossRef Link).
[24] Jet Propulsion Laboratory, NASA Airborne visible infrared imaging spectrometer website. [Online]. Available: http://aviris.jpl.nasa.gov

Changguo Li received his B. E. degree in applied mathematics from Mianyang Normal University, China, in 2004; his M. E. degree in computational mathematics from Sichuan Normal University, China, in 2007; his Ph. D. degree in earth exploration and information technology from Chengdu University of Technology, China, in 2015. He is currently an associate professor with the School of Fundamental Education, Sichuan Normal University, Chengdu, China. His current research interests include hyperspectral image processing, parallel computing and pattern recognition.