Real-Time Implementation of the Pixel Purity Index Algorithm for Endmember Identification on GPUs

Xianyun Wu, Bormin Huang, Antonio Plaza, Yunsong Li, and Chengke Wu

***Abstract*—Spectral unmixing amounts to automatically finding the signatures of pure spectral components (called *endmembers* in the hyperspectral imaging literature) and their associated abundance fractions in each pixel of the hyperspectral image. Many algorithms have been proposed to automatically find spectral endmembers in hyperspectral data sets. Perhaps one of the most popular ones is the pixel purity index (PPI), which is available in the ENVI software from Exelis Visual Information Solutions. This algorithm identifies the endmembers as the pixels with maxima projection values after projections onto a large randomly generated set of random vectors (called *skewers*). Although the algorithm has been widely used in the spectral unmixing community, it is highly time consuming as its precision asymptotically increases. Due to its high computational complexity, the PPI algorithm has been recently implemented in several high-performance computing architectures, including commodity clusters, heterogeneous and distributed systems, field programmable gate arrays, and graphics processing units (GPUs). In this letter, we present an improved GPU implementation of the PPI algorithm, which provides real-time performance for the first time in the literature.**

***Index Terms*—Endmember extraction, graphics processing units (GPUs), hyperspectral imaging, pixel purity index (PPI), real-time processing, spectral unmixing.**

I. Introduction

HYPERSPECTRAL imaging instruments are capable of collecting hundreds of images, corresponding to different wavelength channels, for the same area on the surface of the Earth [1]. The concept of hyperspectral imaging originated at NASA's Jet Propulsion Laboratory in Pasadena, CA, USA, which developed instruments such as the Airborne Visible/Infrared Imaging Spectrometer (AVIRIS). This instrument is able to record the visible and near-infrared spectrum of the reflected light of an area 2–12 km and several kilometers long using 224 spectral bands [2]. A hyperspectral data cube typically comprises several gigabytes per flight. Hyperspectral data have been used in many applications [3], [4], and one of the main problems involved in the analysis of hyperspectral data cubes is the presence of mixed pixels, which arise when the spatial resolution of the sensor is not high enough to separate spectrally distinct materials. Due to the often limited spatial resolution of hyperspectral images and other aspects such as intimate mixtures [5], a single pixel in the scene can comprise the response of different spectrally pure materials. To accurately characterize these substances, spectral unmixing algorithms are generally designed to unmix a pixel into a combination of different pure substances. Nowadays, many popular spectral unmixing algorithms assume linear interaction between different pure substances (*endmembers*), which results in the possibility of expressing a mixed pixel as a linear combination of the endmembers weighted by a set of abundance fractions [3]–[5]. In this context, spectral unmixing allows subpixel analysis, which is crucial in many remote sensing applications.

In recent years, many algorithms have been developed for the purpose of endmember extraction from hyperspectral scenes using different concepts [6]. The pixel purity index (PPI) [7], [8] has been widely used due to its publicity and availability in Exelis Visual Information Solutions ENVI software.[1] The overall nature of this algorithm is supervised, although it has been also used in supervised fashion as preprocessing to other endmember extraction algorithms. First, a pixel purity score is calculated for each point in the image cube by generating k random and unitary n-dimensional vectors called *skewers*. All the pixels in the original n-dimensional space comprised by the input hyperspectral data (with n spectral bands) are then projected onto the skewers, and the ones falling at the extremes of each skewer are tallied. After many repeated projections to different random skewers, those pixels selected a number of times above a certain cutoff threshold t are declared "pure" and loaded into an interactive "n-dimensional visualization tool" (available as a built-in companion piece in ENVI software) and manually rotated until a desired number of endmembers, i.e., p, are visually identified as extreme pixels in the n-dimensional data cloud.

Due to the high computational complexity of the PPI algorithm, several parallel implementations have been discussed in platforms such as clusters [9], [10] and heterogeneous distributed platforms [11], [12]. However, these platforms are difficult to be adapted to on-board processing scenarios, which can be greatly beneficial in applications such as wild land fire tracking, biological threat detection, monitoring of oil spills, and other types of chemical contamination [13]. In this regard, low-weight hardware accelerators such as graphics processing units (GPUs) [12], [15] and field programmable gate arrays [16], [17] have been also explored for accelerating the PPI algorithm.

Manuscript received October 22, 2012; revised February 15, 2013; accepted March 15, 2013.

X. Wu, Y. Li, and C. Wu are with the State Key Laboratory of Integrated Service Networks, Xidian University, Xi'an 710071, China.

B. Huang is with the Space Science and Engineering Center, University of Wisconsin-Madison, Madison, WI 53706 USA (e-mail: bormin@ssec.wisc.edu).

A. Plaza is with the Hyperspectral Computing Laboratory, Department of Technology of Computers and Communications, Escuela Politécnica de Cáceres, University of Extremadura, 10003 Cáceres, Spain.

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

Digital Object Identifier 10.1109/LGRS.2013.2283214

[1] http://www.exelisvis.com/language/en-us/productsservices/envi.aspx

In [15], a GPU-based implementation of the PPI algorithm was presented, which achieved a significant speedup of $196.35\times$ with regard to our CPU-based serial version of the algorithm. However, this implementation was not fast enough to perform real-time processing of hyperspectral image data. In this letter, we develop an improved GPU implementation of the PPI algorithm using the Compute Unified Device Architecture (CUDA) from NVIDIA, the main GPU vendor worldwide, and implemented on both NVIDIA Tesla C1060 and NVIDIA Fermi GTX590, achieving significant improvements when compared with previous GPU-based implementations of the PPI algorithm [12], [15]. The remainder of this letter is organized as follows. Section II outlines the PPI algorithm. Section III describes the proposed strategy for parallel implementation of the PPI algorithm on GPUs. Section IV presents an experimental assessment of the accuracy and parallel performance of the proposed GPU implementation. Section V concludes with some remarks.

II. PPI Algorithm

The PPI algorithm [7] searches for a set of vertices of a convex hull in a given data set, which are supposed to be the purest spectral signatures present in the data. The PPI algorithm described here is based on the limited published results and our own interpretation [8]. Nevertheless, except a final manual supervision step (included in ENVI's PPI), which is replaced by an automatic step in our implementation, both our approximation and the PPI in ENVI produce very similar results. The inputs to the algorithm are a hyperspectral data cube $\mathbf{F}$ with n dimensions; a number of random skewers to be generated during the process, i.e., k; and a cutoff threshold value, i.e., t_v, used to select as final endmembers only those pixels that have been selected as extreme pixels at least t_v times throughout the PPI process. The PPI algorithm used in our implementation is given by the following steps.

1) *Skewer generation*. Produce a set of k randomly generated unit vectors $\{\mathbf{skewer}_j\}_{j=1}^{k}$.
2) *Extreme projections*. For each $\mathbf{skewer}_j$, $j = \{1, 2, \ldots, K\}$, all pixel vectors $\mathbf{f}_i$ in the original data set $\mathbf{F}$ are projected onto $\mathbf{skewer}_j$ via dot products of $\mathbf{f}_i \cdot \mathbf{skewer}_j$ to find sample vectors at its extreme (maximum and minimum) projections, thus forming an extrema set for $\mathbf{skewer}_j$, which is denoted by $S_{\text{extrema}}(\mathbf{skewer}_j)$. Despite the fact that different skewers generate different extrema sets, it is very likely that some sample vectors may appear in more than one extrema set. To account for this, we define an indicator function $I_S(x)$ to denote membership of an element $\mathbf{x}$ to a particular set as follows:
$$I_s(x) = \begin{cases} 1, & \text{if } x \in S \\ 0, & \text{if } x \notin S. \end{cases} \quad (1)$$
3) *Calculation of PPI scores.* Using the preceding indicator function, we calculate the PPI score associated to each pixel vector $\mathbf{f}_i$ (i.e., the number of times that a given pixel has been selected as extreme in step 2) using the following equation:
$$N_{\text{PPI}}(\mathbf{f}_i) = \sum_{j=1}^{K} I_{S_{\text{extrema}}}(\mathbf{skewer}_j)(\mathbf{f}_i). \quad (2)$$

```
// Create pseudo-random number generator
cuRand_CALL(cuRandCreateGenerator(&gen,
cuRand_RNG_PSEUDO_DEFAULT));
// Set seed
cuRand_CALL(cuRandSetPseudoRandomGeneratorSeed(gen,1234U
LL));
// Generate k floats on device
cuRand_CALL(cuRandGenerateUniform(gen,k_skewer,iSkewerNo
*iBand));
```

Fig. 1. CUDA code used for skewer generation in the GPU.

4) *Endmember selection*. Find the pixel vectors with scores of $N_{\text{PPI}}(\mathbf{f}_i)$, which are above t_v, and label them as spectral endmembers. This step replaces a manual endmember selection step conducted in supervised fashion in ENVI software. Optional postprocessing based on removing potentially redundant endmembers may be also applied.

The most time-consuming stage of the PPI algorithm is stage 2 (extreme projections). However, the PPI algorithm is very well suited for parallel implementation since the computations of skewer projections are independent and can be simultaneously performed, leading to many ways of parallelization. In the following section, we present an optimized implementation of the PPI for GPU platforms, which outperforms the very significant speedups reported in [15] and achieves real-time performance.

III. GPU Implementation of the PPI Algorithm

Before describing our GPU implementation of the PPI algorithm, it is first important to describe our data partitioning strategy. Here, we split the image into multiple spatial domain partitions made up of entire pixel vectors, a partitioning strategy explored in [9]–[12] and shown to achieve good results for the parallelization of hyperspectral imaging algorithms. In [15], the core of the parallel implementation was a CUDA kernel in which each thread performed the projection of all pixel vectors onto a skewer for accelerating the extreme projection step. In the following subsections, we describe the different steps of our improved GPU implementation.

A. Skewer Generation

The skewers are n-dimensional vectors in which the components are randomly generated. Since the number of skewers is generally in the order of $k = 10^4$ or more, the total number of random numbers needed, i.e., $k \times n$, is generally very high, and an efficient strategy for random number generation in the GPU is needed. For this purpose, we used the cuRand function provided by CUDA to generate the random values needed to simulate the skewers directly in the GPU. It is important that the skewers are generated in the GPU instead of in the CPU in order to avoid their transmission to the GPU, which would be time consuming due to the generally very large number of skewers involved. Fig. 1 reports the CUDA code used for skewer generation. As shown in Fig. 1, the skewers are generated using cuRand and stored in the global memory of the GPU, in a structure called k_skewer.

B. Extreme Projections

After the skewer generation process has been completed in the GPU, the next step is the projection of all pixel vectors $\mathbf{f}_i$ in the original data set $\mathbf{F}$ onto each $\mathbf{skewer}_j$ via dot products of

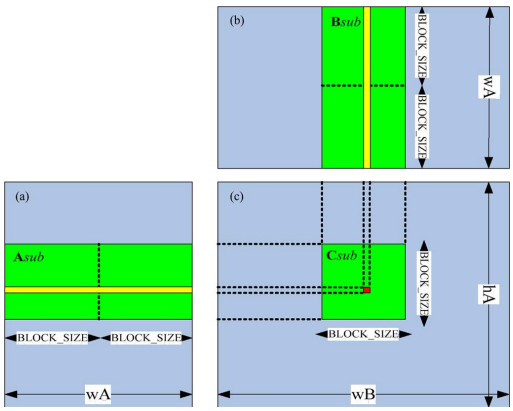

Fig. 2. Matrix multiplication operation in cuBLAS.

```
cublasStatus_t stat ;
cublasHandle_t handle ;
stat = cublasCreate(&handle ) ; //create cublas handle
if( stat != CUBLAS_STATUS_SUCCESS ) {
   printf( "CUBLAS initialize failed \n" ) ;
   return EXIT_FAILURE ;}
stat = cublasSgemm( handle, CUBLAS_OP_T, CUBLAS_OP_T, p, k,
n, &alpha, p_image, p, k_skewer, k, &beta, n_image_ret, n);
if( stat != CUBLAS_STATUS_SUCCESS ) {
   printf ( " cublasSgemm failed " ) ;
   cublasDestroy ( handle ) ;
   return EXIT_FAILURE ;}
stat = cublasDestroy ( handle ) ; //destroy cublas handle
```

Fig. 3. CUDA code used for calculating the skewer projections on the GPU.

$f_i \cdot \mathbf{skewer}_j$ to find sample vectors at its extreme (maximum and minimum) projections. The computations of skewer projections are independent and can be simultaneously performed in the GPU using multiple threads. This independence was exploited in previous work [15]. However, this operation can be expressed as a matrix multiplication by reshaping the hyperspectral image $\mathbf{F}$ as a matrix with dimensions $n \times p$, where n is the number of bands, and p is the number of pixels. Similarly, the skewers are stored in a $k \times n$ matrix, where k is the number of skewers. Hence, a matrix multiplication can be conducted in order to calculate the skewer projections. To do so, the `cuBLAS` library[2] provided by NVIDIA CUDA includes a highly effective matrix multiplication function called `cublasSgemm`, which has been used in this work in order to implement in parallel the skewers projection step of the PPI algorithm.

Fig. 2 describes the cuBLAS implementation of the matrix multiplication operation. As shown in Fig. 3, the task of computing the product $\mathbf{C}$ of two matrices $\mathbf{A}$ and $\mathbf{B}$ can be split among multiple thread blocks (which may be executed in different iterations if the local memory is not enough to store all intermediate results) as follows: each thread block computes one subblock $\mathbf{C}$sub for $\mathbf{C}$, and each thread within the thread block computes several elements of $\mathbf{C}$sub. $\mathbf{C}$sub is the product of two rectangular matrices: the submatrix of $\mathbf{A}$ that has the same row range of $\mathbf{C}$sub and the submatrix of $\mathbf{B}$ that has the same column range of $\mathbf{C}$sub. To meet the GPU constraints, these two rectangular matrices are divided into subblocks (called $\mathbf{A}$sub and $\mathbf{B}$sub), and $\mathbf{C}$sub is computed as the sum of the products of these subblocks. These subblocks in $\mathbf{A}$ and $\mathbf{B}$ do not need to be square. Fig. 3 reports the CUDA code used for skewer projections. As shown in Fig. 3, the result of the skewers projection step is stored in a structure called n_image_ret.

[2]https://developer.nvidia.com/cublas

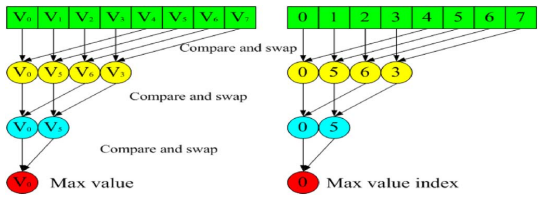

Fig. 4. Endmember identification by reduction process.

```
for(iDim=128;iDim>0;iDim=iDim/2){
  if(ix<iDim){
    if(pixel[ix]<pixel[ix+iDim]){
    //find and swap max value to first half
     temp      = pixel[ix];
     temp_index= index[ix];
     pixel[ix] = pixel[ix+iDim];
     index[ix] = index[ix+iDim];
     pixel[ix+iDim] = temp;
     index[ix+iDim] = temp_index;
 }}
  __syncthreads();
}
```

Fig. 5. CUDA code for the endmember identification by reduction step.

C. Endmember Identification by Reduction

The next step in our parallel implementation is to find the index corresponding to the positions in the matrix with maximum values after the multiplication of the pixels by the skewers has been completed. In [15], this could be easily done since each thread was perfectly identified. However, in our implementation, we use a reduction operation in order to find the maximum and minimum values from projection results in an effective manner. Global memory accesses on the GPU usually take several hundred system clocks, but shared memory devices are much faster. Thus, the projection values are first loaded to the shared memory, whereas the indices are also loaded into the corresponding shared memory. The comparison between projection values and indices is illustrated in Fig. 4. After the process shown in Fig. 4 is repeated, the maximum value and its associated index will be separately stored in the first position. In order to find the minimum value and its associated index, we can use the same algorithm. Using the extreme index values, we finally conform an image containing the PPI scores. Fig. 5 provides the code for this operation.

In our implementation, we use a kernel function to calculate the maximum/minimum values of projection results of all skewers. This function launches 256 threads in each CUDA block by default. This is motivated by the characteristics of the hardware architectures that we are using. Each time, we can perform reduction of 256 values and find the extreme index. In other words, we can perform the reduction for 512 pixels if we compare them first before they are loaded from global memory to shared memory, which will increase the global memory access bandwidth and decrease the execution time. Each projection result has 122 500 (350×350) samples, where 350×350 are the spatial dimensions of the hyperspectral data set that we have considered in experiments. Thus, we need to perform 240 comparisons since each time, we can only compare 512 pixels. After we find the extreme index in each iteration, all the extreme indexes need to be compared one more time to find the final extreme index.

IV. Experimental Results

In our experiments, we have used a real hyperspectral scene collected by the AVIRIS instrument over the Cuprite mining

TABLE I
HARDWARE SPECIFICATIONS OF THE TWO GPUS USED IN OUR EXPERIMENTS

Model	Tesla C1060	Fermi GTX590
Total cores	240 cores (30 MP x 8) cores	512 cores (16 MP x 32) cores
Global memory	~4GB	~6GB
Shared memory per MP	16KB	16/48 KB (configurable)
L1 cache	0	48/16 KB (configurable)
L2 cache	0	768 KB
Registers per MP	16384	32768
Clock rate	1.30GHz	1.22 GHz

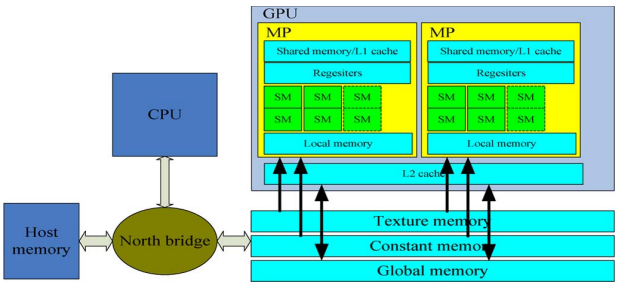

Fig. 6. Memory architecture of Fermi GTX590.

district in Nevada.[3] The portion used in the experiments corresponds to a 350 × 350 pixel subset of the sector labeled as f970619t01p02 r02 sc03.a.rfl. The scene comprises 224 spectral bands between 0.4 and 2.5 μm, with a nominal spectral resolution of 10 nm. Prior to the analysis, bands 1–3, 107–114, 153–169, and 221–224 were removed due to water absorption and low signal-to-noise ratio in those bands. The number of skewers used in experiments was set to $k = 15\,360$. The execution time measured after processing the AVIRIS Cuprite scene using our implemented CPU version of the PPI algorithm on an Intel Core i7 920 CPU (using just one of the available cores) was 3454.55 s [15]. In order to test the computational performance of our proposed parallel method, we used two different NVIDIA GPUs: Tesla C1060 and Fermi GTX590. For illustrative purposes, Table I describes the hardware specifications of the two GPUs used in our experiments.

A. Implementation Without Using Shared Memory

In the Fermi architecture (see Fig. 6), the GPU is composed of a set of multiprocessors (MPs), each with 32 cores and two levels of cache. The first level (L1) is available to each MP, and the second level (L2) can be shared by all MPs. Both levels are used to cache accesses to local or global memory, including temporary register spills. The cache behavior (e.g., whether reads are cached in both L1 and L2 or in L2 only) can be partially configured on a per-access basis using modifiers to the load or store instruction. The same on-chip memory is used for both L1 and the shared memory; it can be configured as 48 kB of shared memory with 16 kB of L1 cache (default setting) or as 16 kB of shared memory with 48 kB of L1 cache using cudaFuncSetCacheConfig or cuFuncSetCache configuration functions. This feature can increase the performance significantly. Since L1/L2 cache is available on the Fermi architecture, we simply compute each skewer projection with one GPU thread and access to global memory directly. With this method, we obtained 601.31× speedup, which is a very significant improvement with regard to the results presented in [17] (see Fig. 7 for a comparison of the speedups achieved with regard to our CPU implementation and the commercial software version distributed by ENVI).

[3]Available online at http://aviris.jpl.nasa.gov.

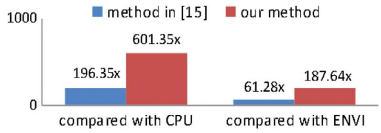

Fig. 7. Speedup results obtained on Fermi GTX 590 with regard to a CPU implementation of the PPI and with regard to the commercial version of the algorithm available in ENVI (without using shared memory).

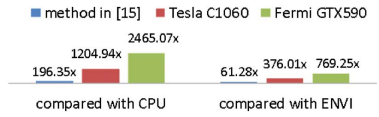

Fig. 8. Speedup results obtained by the GPU implementation in Tesla C1060 and Fermi GTX 590 (using shared memory in both cases) with regard to our CPU implementation and the commercial version of the algorithm available in ENVI.

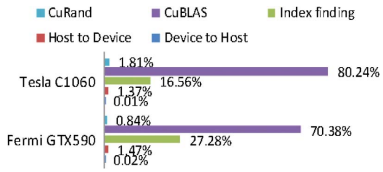

Fig. 9. Percentage of the total GPU time consumed by different kernels on Tesla C1060 and Fermi GTX590 (using shared memory in both cases).

B. Implementation Using Shared Memory

It should be noted that, although L1/L2 cache can significantly improve performance, we can further accelerate our efficiency using the shared memory. As explained in the previous section, shared memory arrays should be used as much as possible in order to minimize the use of global memory arrays. With shared memory accesses in our new method, the PPI execution time on Tesla C1060 decreases to 2.866 s, which satisfies the real-time processing requirement (AVIRIS is a push-broom instrument that needs approximately 2.985 s to collect the Cuprite scene [18]), and the speedup in this case is up to 1204.94× when compared with the CPU-based serial implementation, which was developed in C language and compiled with O3 flag for optimization. When we run our codes on Fermi GTX590 GPU, an impressive speedup of 2465.07× was achieved with regard to the CPU-based serial implementation. The speedup is up to 769.25× compared with the commercial ENVI software implementation. Fig. 8 summarizes the obtained results, which are state of the art for the PPI.

For illustrative purposes, Fig. 9 shows the percentage of the total execution time consumed by each kernel invoked. These values were obtained after profiling the implementation using the CUDA visual profiler tool distributed by NVIDIA. In the figure, the percentage of time for data movements is from the host (CPU) to the device (GPU) and from the device to the host. It should be noted that the execution time from the host to the device includes the cuBLAS initialization time. As shown in Fig. 9, the PPI kernel consumes most of the total GPU time (including CuBLAS and index finding), whereas the

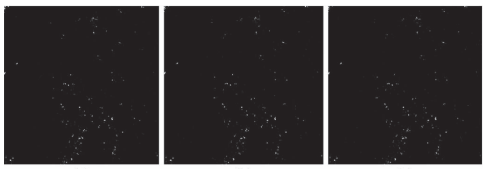

Fig. 10. Pixels selected in the AVIRIS Cuprite scene after the skewers projection step by different implementations of the PPI. (a) ENVI. (b) CPU. (c) GPU.

`cuRand` kernel comparatively occupies a much smaller fraction. Finally, the data movement operations are not significant, which indicates that most of the GPU processing time is invested in the most time-consuming operation, i.e., the calculation of skewer projections and the identification of maxima and minima projection values leading to endmember identification.

C. Analysis of PPI Score Image

Fig. 10 shows the pixels selected from the AVIRIS Cuprite image after the skewer projections step by the original ENVI implementation, our PPI algorithm implementation in C code, and our GPU-based implementation after using 15 360 skewers. As shown in Fig. 10, the three implementations provide very similar results with negligible differences in terms of pixel selections, which indicate that their calculation of the PPI scores is almost equivalent.

D. Computational Complexity Analysis

The PPI is based on regular computations (i.e., skewer pixel projections and identification of maxima and minima projection values). We have experimentally observed that the GPU implementation of the PPI is embarrassingly parallel in the sense that its computational efficiency can be linearly improved by adding more computing resources, as it scales linearly with three factors, namely, the number of pixels, the number of bands, and the number of skewers, and independently of the number of endmembers or the complexity of the considered scenes. This has been experimentally observed with several scenes other than the AVIRIS Cuprite data, including all freely available data sets in[4] and with an AVIRIS scene collected over the World Trade Center in New York after September 11, 2011.

V. Conclusion and Future Lines

In this letter, we have reported the first real-time implementation of the PPI algorithm commonly used for endmember extraction from hyperspectral images on GPUs. The L1/L2 cache levels available on Fermi architecture are exploited in order to obtain an implementation with a speedup of $601.31\times$ on the NVIDIA GTX590 GPU (using global memory accesses). When this implementation is further optimized by using shared memory devices, we achieved an impressive speedup of $1204.94\times$ on Tesla C1060 and $2465.07\times$ on Fermi GTX590. The latter speedup is $769.25\times$ when compared with the commercial ENVI software implementation of the PPI. These are state-of-the-art results that allow the PPI algorithm to perform in real time for a real hyperspectral scene for the first time in the literature. GPUs still suffer from several problems (e.g., high power consumption and energy requirements and lack of radiation-tolerant versions), which currently prevents their exploitation in satellite missions. However, GPUs represent a promising hardware accelerator subject to technological improvements that may allow its future incorporation to spaceborne remote sensing missions also. As future work, we are currently exploring the role of multicore CPU architectures in hyperspectral imaging and further developing an implementation of the PPI algorithm using OpenMP that will lead to a comparison of GPUs versus multicore CPUs in this context.

[4]http://aviris.jpl.nasa.gov/data/free_data.html

References

[1] A. F. H. Goetz, G. Vane, J. E. Solomon, and B. N. Rock, "Imaging spectrometry for Earth remote sensing," *Science*, vol. 228, no. 4704, pp. 1147–1153, Jun. 1985.

[2] R. O. Green, M. L. Eastwood, C. M. Sarture, T. G. Chrien, M. Aronsson, B. J. Chippendale, J. A. Faust, B. E. Pavri, C. J. Chovit, M. Solis, M. R Olah, and O. Williams, "Imaging spectroscopy and the Airborne Visible/Infrared Imaging Spectrometer (AVIRIS)," *Remote Sens. Environ.*, vol. 65, no. 3, pp. 227–248, Sep. 1998.

[3] A. Plaza, J. A. Benediktsson, J. Boardman, J. Brazile, L. Bruzzone, G. Camps-Valls, J. Chanussot, M. Fauvel, P. Gamba, J. Gualtieri, M. Marconcini, J. C. Tilton, and G. Trianni, "Recent advances in techniques for hyperspectral image processing," *Remote Sens. Environ.*, vol. 113, no. S1, pp. S110–S122, Sep. 2009.

[4] A. Plaza, P. Martinez, R. Perez, and J. Plaza, "A quantitative and comparative analysis of end member extraction algorithms from hyperspectral data," *IEEE Trans. Geosci. Remote Sens.*, vol. 42, no. 3, pp. 650–663, Mar. 2004.

[5] N. Keshava and J. F. Mustard, "Spectral unmixing," *IEEE Signal Process. Mag.*, vol. 19, no. 1, pp. 44–57, Jan. 2002.

[6] J. M. Bioucas-Dias, A. Plaza, N. Dobigeon, M. Parente, Q. Du, P. Gader, and J. Chanussot, "Hyperspectral unmixing overview: Geometrical, statistical and sparse regression-based approaches," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 5, no. 2, pp. 354–379, Apr. 2012.

[7] J. W. Boardman, F. A. Kruse, and R. O. Green, "Mapping target signatures via partial unmixing of AVIRIS data," in *Proc. JPL Airborne Earth Sci. Workshop*, 1995, pp. 23–26.

[8] C.-I. Chang and A. Plaza, "A fast iterative algorithm for implementation of pixel purity index," *IEEE Geosci. Remote Sens. Lett.*, vol. 3, no. 1, pp. 63–67, Jan. 2006.

[9] A. Plaza, D. Valencia, J. Plaza, and P. Martinez, "Commodity cluster-based parallel processing of hyperspectral imagery," *J. Parallel Distrib. Comput.*, vol. 66, no. 3, pp. 345–358, Mar. 2006.

[10] A. Plaza and C.-I. Chang, "Clusters versus FPGA for parallel processing of hyperspectral imagery," *Int. J. High Perform. Comput. Appl.*, vol. 22, no. 4, pp. 366–385, Nov. 2008.

[11] D. Valencia, A. Lastovetsky, M. O'Flynn, A. Plaza, and J. Plaza, "Parallel processing of remotely sensed hyperspectral images on heterogeneous networks of workstations using HeteroMPI," *Int. J. High Perform. Comput. Appl.*, vol. 22, no. 4, pp. 386–407, Nov. 2008.

[12] A. Plaza, J. Plaza, and H. Vegas, "Improving the performance of hyperspectral image and signal processing algorithms using parallel, distributed and specialized hardware-based systems," *J. Signal Process. Syst.*, vol. 61, no. 3, pp. 293–315, Dec. 2010.

[13] A. Plaza and C.-I. Chang, *High Performance Computing in Remote Sensing*. Boca Raton, FL, USA: CRC Press, 2007.

[14] H. Chang and C.-I. Chang, "Field programmable gate arrays (FPGA) for pixel purity index using blocks of skewers for endmember extraction in hyperspectral imagery," *Int. J. High Perform. Comput. Appl.*, vol. 22, no. 4, pp. 408–423, Nov. 2008.

[15] S. Sánchez and A. Plaza, "GPU implementation of the pixel purity index algorithm for hyperspectral image analysis," in *Proc. IEEE Int. Conf. Cluster Comput.*, Heraklion, Crete, Sep. 2010, vol. 1, pp. 1–7.

[16] C. Gonzalez, J. Resano, D. Mozos, A. Plaza, and D. Valencia, "FPGA implementation of the pixel purity index algorithm for remotely sensed hyperspectral image analysis," *EURASIP J. Adv. Signal Process.*, vol. 2010, no. 1, pp. 969 806-1–969 806-13, Jun. 2010.

[17] C. Gonzalez, D. Mozos, J. Resano, and A. Plaza, "FPGA for computing the pixel purity index algorithm on hyperspectral images," in *Proc. ERSA*, Las Vegas, NV, USA, 2010, p. 293.

[18] R. N. Clark, K. E. Livo, and R. F. Kokaly, "Geometric correction of AVIRIS imagery using on-board navigation and engineering data," in *Proc. 7th Annu. JPL Airborne Earth Sci. Workshop*, R. O. Green, Ed., Jan. 12–14, 1998, pp. 57–65.