Further Optimizations of the GPU-based Pixel Purity Index Algorithm for Hyperspectral Unmixing

Xianyun Wu[1], Bormin Huang[2*], Antonio Plaza[3], Yunsong Li[1], and Chengke Wu[1]

[1]State Key Laboratory of Integrated Service Networks, Xidian University, Xi'an 710071, China
[2]Space Science and Engineering Center, University of Wisconsin-Madison, Madison, WI 54706, USA
[3]Department of Technology of Computers and Communications, University of Extremadura, 10071 Cáceres, Spain

ABSTRACT

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), available in the ENVI software from Exelis Visual Information Solutions. Although the algorithm has been widely used in the spectral unmixing community, it is highly time consuming as its precision increases asymptotically. 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 (FPGAs) 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.

Keywords: Hyperspectral unmixing, endmember extraction, pixel purity index (PPI), graphics processing units (GPUs)

1. INTRODUCTION

In recent years, many algorithms have been developed for the purpose of endmember extraction from hyperspectral scenes using different concepts [1-6]. The pixel purity index (PPI) [7, 8] has been widely used due to its publicity and availability in Exelis Visual Information Solutions ENVI software. The overall nature of this algorithm is supervised, although it has also been used in supervised fashion as a pre-processing 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 cut-off 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, 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 field programmable gate arrays [14] and graphics processing units (GPUs) [12] have also been explored for accelerating the PPI algorithm.

In [15], a GPU-based implementation of the PPI algorithm was presented which achieved a significant speedup of 196.35x with regards to an optimized serial version of the algorithm. However, this implementation was not fast enough to perform real-time processing of hyperspectral image data. In this paper, we develop an improved GPU implementation of the PPI algorithm using the compute device unified architecture (CUDA) from NVidia, the main GPU vendor worldwide. and implemented on both NVIDIA Tesla C1060 and NVIDIA Fermi GTX590, achieving significant

* Corresponding author, phone: 608-265-2231; e-mail: bormin@ssec.wisc.edu

High-Performance Computing in Remote Sensing II, edited by Bormin Huang, Antonio J. Plaza,
Proc. of SPIE Vol. 8539, 85390D · © 2012 SPIE · CCC code: 0277-786/12/$18
doi: 10.1117/12.979310

improvements when compared with previous GPU-based implementations of the PPI algorithm [12, 15]. The remainder of this letter is organized as follows. Section 2 outlines the PPI algorithm. Section 3 describes the proposed strategy for parallel implementation of the PPI algorithm on GPUs. Section 4 presents an experimental assessment of the accuracy and parallel performance of the proposed GPU implementation. Section 5 concludes with some remarks.

2. THE PPI ALGORITHM

The pixel purity index (PPI) algorithm [7] searches for a set of vertices of a convex hull in a given dataset, which are supposed to be the purest spectral signatures present in the data. The PPI algorithm described in this section 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 **F** with *n* dimensions; a number of random skewers to be generated during the process, *k*; and a cut-off threshold value, 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 **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 $S_{extrema}(\mathbf{skewer}_j)$. 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 **x** to a particular set as follows:
$$I_S(x) = \begin{cases} 1 & if \quad x \in S \\ 0 & if \quad x \notin S \end{cases} \tag{1}$$
3. *Calculation of PPI scores.* Using the indicator function above, we calculate the PPI score associated to each pixel vector $\mathbf{f}_i$ (i.e., the number of times that given pixel has been selected as extreme in step 2) using the following equation:
$$N_{PPI}(\mathbf{f}_i) = \sum_{j=1}^{K} I_{S_{extrema}(\mathbf{skewer}_j)}(\mathbf{f}_i) \tag{2}$$
4. *Endmember selection*. Find the pixel vectors with scores of $N_{PPI}(\mathbf{f}_i)$ 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. An optional post-processing 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 performed simultaneously, 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.

3. GPU IMPLEMENTATION

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 generated randomly. Since the number of skewers is generally in the order of $k = 10^4$ or more, the total number of random numbers needed, $k \times n$, 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 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 by Fig. 1, the skewers are generated using cuRand and stored in the global memory of the GPU, in a structure called k_skewer.

```
// Create pseudo-random number generator
cuRand_CALL(cuRandCreateGenerator(&gen,
cuRand_RNG_PSEUDO_DEFAULT));
// Set seed
cuRand_CALL(cuRandSetPseudoRandomGeneratorSeed(gen,12
34ULL));
// Generate k floats on device
cuRand_CALL(cuRandGenerateUniform(gen,k_skewer,iSkewer
No*iBand));
```

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

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 **F** onto each $\mathbf{skewer}_j$ via dot products of $\mathbf{f}_i \cdot \mathbf{skewer}_j$ to find sample vectors at its extreme (maximum and minimum) projections. The computation of skewer projections are independent and can be performed simultaneously 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 **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[†] 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.

Figure 2 reports the CUDA code used for skewer projections. As shown by Fig. 2, the result of the skewers projection step is stored in a structure called n_image_ret.

```
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
```

Figure 2. CUDA code used for calculating the skewer projections on the GPU.

[†] http://www.nvidia.com/cuda

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 done easily 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 memories are much faster. So, the projection values are first loaded to the shared memory while the indices are also loaded into the corresponding shared memory. The comparison between projection values and the indices is illustrated in Fig. 3. After the process shown in Fig. 3 is repeated, the maximum value and its associated index will be separately stored in the first position.

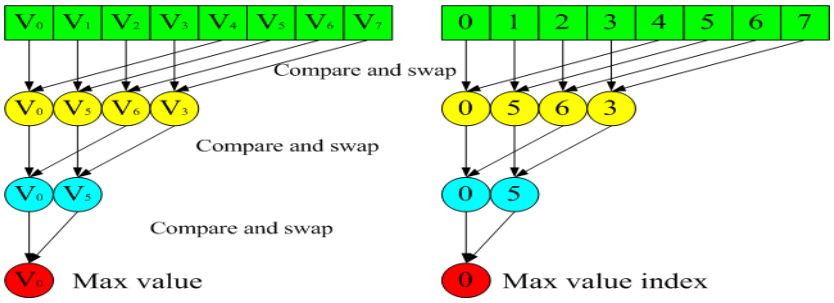

Figure 3. Endmember identification by reduction process.

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.

In our implementation, we launch 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 122500 (350 x 350) samples, where 350 x 350 are the spatial dimensions of the hyperspectral data set that we have considered in experiments. So, 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.

4. EXPERIMENTAL RESULTS

In our experiments we have used a real hyperspectral scene collected by the AVIRIS instrument over the Cuprite mining district in Nevada[‡]. The portion used in 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 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 SNR in those bands. The number of skewers used in experiments was set to k = 15360. The execution time measured after processing the AVIRIS Cuprite scene using an optimized CPU version of the PPI algorithm on an Intel Core i7 920 CPU (using just one of the available cores) was 3454.55 seconds [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 1 describes the hardware specifications of the two GPUs used in our experiments.

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

Table 1. 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

In the Fermi architecture, the GPU is composed by a set of multiprocessors (MPs), each with 32 cores and two levels of cache. The first level (L1) is available to each multiprocessor, and the second level (L2) can be shared by all multiprocessors [18]. 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 601.31x speedup which is a very significant improvement with regard to the results presented in [15]. Fig. 4 shows a comparison of the speedups achieved with regards to our optimized CPU implementation and the commercial software version distributed in ENVI.

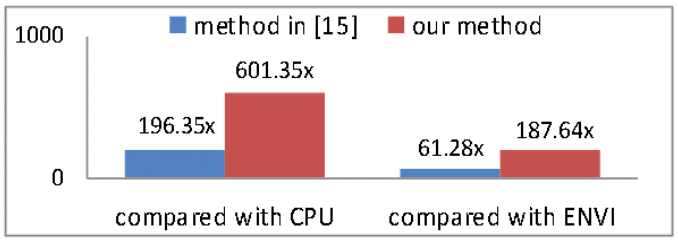

Figure 4. Speedup results obtained on the Fermi GTX 590 with regard to a CPU implementation of PPI and with regard to the commercial version of the algorithm available in ENVI (without using shared memory).

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 the Tesla C1060 decreases to 2.866 seconds which satisfies the real-time processing requirement (AVIRIS is a push broom instrument that needs approximately 2.985 seconds to collect the Cuprite scene) and the speedup in this case is up to 1204.94x when compared with the optimized serial implementation, developed in C language and compiled with O3 flag for optimization. When we run our codes on the Fermi GTX590 GPU, an impressive speedup of 2465.07x was achieved with regard to the optimized serial implementation. The speedup is up to 769.25x compared with the commercial ENVI software implementation. Fig. 8 summarizes the obtained results, which are state-of-the-art for the PPI.

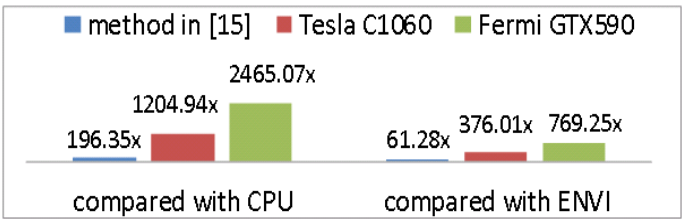

Figure 8. Speedup results obtained on the Fermi GTX 590 with regard to a CPU implementation of PPI and with regard to the commercial version of the algorithm available in ENVI (without using shared memory).

5. CONCLUSIONS

In this paper, 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 601.31x speedup on the NVidia GTX590 GPU (using global memory accesses). When this implementation is further optimized by using shared memories, we achieved an impressive speedup of 1204.94x on the Tesla C1060 and of 2465.07x speedup on the Fermi GTX590. The latter speedup is 769.25x 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.

REFERENCES

[1]. A. F. H. Goetz, G. Vane, J. E. Solomon, and B. N. Rock, "Imaging spectrometry for Earth remote sensing," Science, vol. 228, pp. 1147– 1153, 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 et al., "Imaging spectroscopy and the airborne visible/infrared imaging spectrometer (AVIRIS)," Remote Sensing of Environment, vol. 65, no. 3, pp. 227–248, 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 Sensing of Environment, vol. 113, pp. 110–122, 2009.

[4]. A. Plaza, P. Martinez, R. Perez, and J. Plaza, "A quantitative and comparative analysis of endmember extraction algorithms from hyperspectral data," IEEE Transactions on Geoscience and Remote Sensing, vol. 42, pp. 650–663, 2004.

[5]. N. Keshava and J. F. Mustard, "Spectral unmixing," IEEE Signal Processing Magazine, vol. 19, no. 1, pp. 44–57, 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 Journal of Selected Topics in Applied Earth Observations and Remote Sensing, vol. 5, no. 2, pp. 354-379, April 2012.

[7]. J. W. Boardman, F. A. Kruse, and R. O. Green, "Mapping Target Signatures Via Partial Unmixing of Aviris Data," Proc. JPL Airborne Earth Sci. Workshop, pp. 23–26, 1995.

[8]. C.-I Chang and A. Plaza, "A fast iterative algorithm for implementation of Pixel Purity Index," IEEE Geoscience and Remote Sensing Letters, vol. 3, pp. 63–67, 2006.

[9]. A. Plaza, D. Valencia, J. Plaza, and P. Martinez, "Commodity cluster-based parallel processing of hyperspectral imagery," Journal of Parallel and Distributed Computing, vol. 66, pp. 345–358, 2006.

[10]. A. Plaza and C.-I Chang, "Clusters versus FPGA for parallel processing of hyperspectral imagery," International Journal of High Performance Computing Applications, vol. 22, no. 4, pp. 366-385, November 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," International Journal of High Performance Computing Applications, vol. 22, no. 4, pp. 386-407, November 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," Journal of Signal Processing Systems, vol. 50, pp. 293-315, December 2010.

[13]. A. Plaza and C.-I. Chang, High Performance Computing in Remote Sensing. Taylor & Francis: Boca Raton, FL, 2007.

[14]. M. Hsueh and C.-I Chang, "Field programmable gate arrays (FPGA) for pixel purity index using blocks of skewers for endmember extraction in hyperspectral Imagery," International Journal of High Performance Computing Applications, vol. 22, no. 4, pp. 408-423, November 2008.

[15]. S. Sánchez and A. Plaza, "GPU implementation of the pixel purity index algorithm for hyperspectral image analysis," Proceedings of the IEEE International Conference on Cluster Computing, vol. 1, pp.1-7, Sept. 2010.