Real-Time Implementation of the Vertex Component Analysis Algorithm on GPUs

A. Barberis, G. Danese, F. Leporati, A. Plaza, and E. Torti

***Abstract*—In this letter, we present a new parallel implementation of the vertex component analysis (VCA) algorithm for spectral unmixing of remotely sensed hyperspectral data on commodity graphics processing units. We first developed a C serial version of the VCA algorithm and three parallel versions: one using NVIDIA's Compute Unified Device Architecture (CUDA), another using CUDA basic linear algebra subroutines library CUBLAS, and the last using the CUDA linear algebra library CULA. Experimental results, based on the analysis of hyperspectral images acquired by a variety of hyperspectral imaging sensors, show the effectiveness of our implementation, which satisfies the real-time constraints given by the data acquisition rate.**

***Index Terms*—Graphics processing units (GPUs), hyperspectral imaging, spectral unmixing, vertex component analysis (VCA).**

I. Introduction

SPECTRAL unmixing is an important technique for remote sensing data exploitation [1]. For instance, the well-known NASA Jet Propulsion Laboratory's Airborne Visible Infrared Imaging Spectrometer (AVIRIS) [2] is able to collect data in the visible and near-infrared spectra (wavelength region between 380 and 2500 nm), producing "data cubes" with a typical size of 614 lines with 512 samples and 224 spectral bands with a wavelength range of 10 nm. This amounts to approximately 140 MB per data cube. AVIRIS is mounted on an airborne facility, which can be either an Earth Resources 2 (able to fly about 20 km above the ground) or a Twin Otter (able to fly about 4 km above the ground). Typical resolutions provided by these devices are not sufficient to separate different spectrally pure constituents (*endmembers*) within a data cube, so that more of them can be found within the same pixel. Spectral measures of these "mixed" pixels are viewed as mixtures of pure constituent spectra (*endmember signatures*).

Two distinct mixture models are generally considered to *unmix* hyperspectral data: the linear and the nonlinear models [3]. In the linear model, each "mixed" pixel is viewed as a linear combination of underlying endmember signatures weighted by the correspondent *abundance fractions* (i.e., the percentage of pure material present in the pixel).

Let us denote by $\mathbf{R}$ an N-band remotely sensed data cube, so that a pixel vector at discrete spatial coordinates (i, j) can be represented as an N-dimensional array $\mathbf{r}_{ij} = [r_{ij}(1), r_{ij}(2), \ldots, r_{ij}(N)] \in \mathbb{R}^N$, where $r_{ij}(k)$ is the spectral response related to sensor channels $k = 1, 2, \ldots, N$. The pixel vector at spatial coordinates (i, j) can then be modeled as

$$\mathbf{r}_{ij} = \mathbf{M}\mathbf{a}_{ij} + \mathbf{n}_{ij} = \sum_{k=1}^{p} \mathbf{m}_{\mathrm{k}} a_{ij}^{\mathrm{k}} + \mathbf{n}_{ij} \tag{1}$$

where $\mathbf{M} \equiv [\mathbf{m}_1, \mathbf{m}_2, \ldots, \mathbf{m}_p]$ is a so-called "mixing matrix," containing p endmember signatures, $\mathbf{a}_{ij} = [a_{ij}^1, a_{ij}^2, \ldots, a_{ij}^p]^{\mathrm{T}}$ is the abundance vector (in particular, a_{ij}^k is the *abundance fraction* of kth endmember, with $k = 1, 2, \ldots, p$) and $\mathbf{n}_{ij}$ is an additive noise vector. The notation $(\cdot)^{\mathrm{T}}$ stands for vector transposed. Solving the model (i.e., "unmixing") amounts at obtaining a good estimation of the mixing matrix and the associated fractional abundances. Several approaches have been used to perform spectral unmixing based on automatic endmember identification, including techniques assuming the presence of pure signatures in the data cube and also techniques without the pure signature assumption [4]. A successful algorithm in the first category has been the vertex component analysis (VCA) [5]. However, due to the amount of hyperspectral data, high-performance computing is required, particularly for those scenarios requiring real-time response (namely, monitoring of chemical contamination, wildfire tracking, biological threat detection, etc.). In recent years, parallel computing techniques have been widely used to accelerate hyperspectral imaging algorithms. Specifically, implementations on supercomputers [6], clusters [7], [8], and specialized hardware devices, such as field-programmable gate arrays (FPGAs) [9], [10] and graphics processing units (GPUs) [11], have been made available. In particular, GPUs are quickly evolving as a standardized architecture in hyperspectral imaging due to their low cost, portability, and high computational power [11]. Nevertheless, to the best of our knowledge, the widely used VCA algorithm has not yet been implemented in parallel.

In this letter, we describe a first GPU implementation of VCA based on the NVIDIA Fermi architecture (http://www.nvidia.com/object/fermi_architecture.html) and using NVIDIA's Compute Unified Device Architecture [(CUDA), http://www.nvidia.com/object/cuda_home_new.html] with the CUDA

Manuscript received January 24, 2012; revised March 19, 2012; accepted April 26, 2012.

A. Barberis, G. Danese, F. Leporati, and E. Torti are with the Dipartimento di Ingegneria Industriale e dell'Informazione, University of Pavia, 27100 Pavia, Italy (e-mail: alessandro.barberis02@ateneopv.it; gianni.danese@unipv.it; francesco.leporati@unipv.it; emanuele.torti01@unipv.it).

A. Plaza is with the Hyperspectral Computing Laboratory, Department of Technology of Computers and Communications, University of Extremadura, 10071 Cáceres, Spain (e-mail: aplaza@unex.es).

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

Digital Object Identifier 10.1109/LGRS.2012.2200452

basic linear algebra subroutines [(CUBLAS), http://developer.nvidia.com/cublas] and with the CUDA linear algebra library [(CULA), http://www.culatools.com/]. Sections II and III describe the VCA algorithm for endmember extraction and its GPU implementation. Section IV presents the performance tests carried out by using four different hyperspectral scenes. Section V concludes this letter with some remarks and hints at plausible future research.

II. VCA Algorithm

The VCA algorithm provides a geometrical solution to the unmixing problem and is based on two main assumptions. 1) The endmembers are the vertices of a simplex, and 2) the affine transformation of a simplex is also a simplex. The algorithm assumes the presence of pure signatures in the hyperspectral data. Considering the linear mixing scenario in (1), due to physical constraints [6], abundance fractions satisfy $\mathbf{1}^{\mathrm{T}}\mathbf{a} = 1$ and are nonnegative (i.e., $0 \leq a^k \leq 1$). The set $S_x = \{\mathbf{x} \in R^N : \mathbf{x} = \mathbf{M}\mathbf{a}, \mathbf{1}^{\mathrm{T}}\mathbf{a} = 1, \mathbf{a} \geq \mathbf{0}\}$ is a simplex, since the set $\{\mathbf{a} \in R^p : \mathbf{1}^{\mathrm{T}}\mathbf{a} = 1, \mathbf{a} \geq \mathbf{0}\}$ is already a simplex. Considering now a scale factor γ (i.e., illumination variability due to surface topography) and assuming zero noise $(\mathbf{n} = \mathbf{0})$ so that $\mathbf{r}_{ij} = \mathbf{M}\gamma_{ij}\mathbf{a}_{ij}$, the modeled pixel belongs to a convex cone $C = \{\mathbf{r} \in R^N : \mathbf{r} = \mathbf{M}\gamma\mathbf{a}, \mathbf{1}^{\mathrm{T}}\mathbf{a} = 1, \mathbf{a} \geq \mathbf{0}, \gamma \geq 0\}$.

A projection of the simplex onto a suitable hyperplane $\mathbf{r}^{\mathrm{T}}\mathbf{u} = 1$ (the choice of $\mathbf{u}$ ensures that there are no observed vectors which are orthogonal to it) provides the new simplex $S_y = \{\mathbf{y} \in R^N : \mathbf{y} = \mathbf{r}/(\mathbf{r}^{\mathrm{T}}\mathbf{u}), \mathbf{r} \in C\}$. After this initial projection, the algorithm iteratively projects data onto a direction orthogonal to the subspace spanned by the endmembers already determined, and the extreme of the projection in each iteration is the new endmember identified [5]. Usually, the number of endmembers is much smaller than the number of bands, so that the data live in a subspace $p \ll N$. In order to reduce the computational complexity, VCA projects the observed data onto the subspace signal. The authors decided to use *principal component analysis* (PCA) or *singular value decomposition* (SVD) depending on a proper signal-to-noise ratio (SNR) threshold empirically selected [5]. In fact, the authors of VCA found that, when the data are noiseless, SVD followed by a projection works better, while with noisy data, PCA gives the best results. Table I shows the pseudocode of the VCA algorithm. Step 1 calculates the threshold SNRth. Step 2 tests if SNR is higher than SNRth in order to project data onto a p-dimensional or a $(p-1)$-dimensional subspace. Steps 4 and 9 give the projections obtained with SVD in the first case and with PCA in the second. Step 14 initializes the auxiliary matrix $\mathbf{A}$, where the projections of the estimated endmember signatures are stored. Steps 15–22 define the main loop of the algorithm: Here, the data that live in the previously detected subspace are projected (each time the *for* loop is executed) onto $\mathbf{f}$ relevant directions, where $\mathbf{f}$ is a vector orthonormal to the space spanned by the columns of the auxiliary matrix $\mathbf{A}$, where notation $(\cdot)^{\#}$ stands for the pseudoinverse matrix. Step 23 tests again if the SNR is higher than SNRth, to determine the matrix $\widehat{\mathbf{M}}$ which contains the endmembers identified in the N-dimensional space given by the original image.

TABLE I
VCA

INPUT p, $\mathbf{R} \equiv [\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N]$

1: $\mathrm{SNR_{th}} = 15 + \log_{10}(p)$ dB
2: **if** SNR > $\mathrm{SNR_{th}}$ **then**
3: $d := p$;
4: $\mathbf{X} := \mathbf{U}_{\mathbf{d}}^{\mathbf{T}}\mathbf{R}$ {$\mathrm{U_d}$ obtained by SVD}
5: $\mathbf{u} := \mathrm{mean}(\mathbf{X})$; {$\mathbf{u}$ is a $1 \times d$ vector}
6: $[\mathbf{Y}]_{.,j} := [\mathbf{X}]_{.,j} / ([\mathbf{X}]_{.,j}\mathbf{u})$; {projective projection}
7: **else**
8: $d := p - 1$;
9: $[\mathbf{X}]_{.,j} := \mathbf{U}_{\mathbf{d}}^{\mathbf{T}}([\mathbf{R}]_{.,j} - \bar{\mathbf{r}})$ {$\mathrm{U_d}$ obtained by PCA}
10: $\mathrm{c} := \arg\max_{j=1,\ldots,N} \| [\mathbf{X}]_{.,j} \|$;
11: $\mathbf{c} := [\mathrm{c}|\mathrm{c}| \ldots |\mathrm{c}]$; {c is a $1 \times N$ vector}
12: $\mathbf{Y} := \begin{bmatrix} \mathbf{X} \\ \mathbf{c} \end{bmatrix}$
13: **end if**
14: $\mathbf{A} := [\mathbf{e}_u|\mathbf{0}| \ldots |\mathbf{0}]$; {$\mathbf{e}_u = [0, \ldots, 0, 1]^{\mathrm{T}}$ and $\mathbf{A}$ is $p \times p$ auxiliary matrix}
15: **for** $i := 1$ to p **do**
16: $\mathbf{w} :=$ rand $(0, \mathbf{I}_p)$; {$\mathbf{w}$ is a zero-mean random Gaussian vector of covariance $\mathbf{I}_p$ }
17: $\mathbf{f} := ((\mathbf{I} - \mathbf{A}\mathbf{A}^{\#})\mathbf{w}) / (\|(\mathbf{I} - \mathbf{A}\mathbf{A}^{\#})\mathbf{w}\|)$; {$\mathbf{f}$ is a vector orthonormal to the subspace spanned by $[\mathbf{X}]_{.,1:i}$ }
18: $\mathbf{v} := \mathbf{f}^{\mathrm{T}}\mathbf{Y}$;
19: $k := \arg\max_{j=1,\ldots,N} |[\mathbf{v}]_{.,j}|$; {find the projection extreme}
20: $[\mathbf{A}]_{.,i} := [\mathbf{Y}]_{.,k}$
21: $[indice]_i := k$; {store the pixel index}
22: **end for**
23: **if** SNR > $\mathrm{SNR_{th}}$ **then**
24: $\widehat{\mathbf{M}} := \mathbf{U}_d [\mathbf{X}]_{.,indice}$; { $\widehat{\mathbf{M}}$ is a *Nxp* estimated mixing matrix}
25: **else**
26: $\widehat{\mathbf{M}} := \mathbf{U}_d [\mathbf{X}]_{.,indice} + \bar{\mathbf{r}}$; { $\widehat{\mathbf{M}}$ is a *Nxp* estimated mixing matrix}
27: **end if**

III. VCA Implementation on NVIDIA GPUs

The VCA algorithm is online as a Matlab implementation (http://www.lx.it.pt/~bioucas/code/demo_vca.zip). In order to achieve a fair benchmark in terms of execution time with respect to the GPU version, we first conceived a C implementation as a basis for the subsequent parallel implementations (the matrix multiplications needed by the algorithm are performed as row–column products). In the following, we describe some general aspects of the used Fermi GPU architecture and our implementation of VCA on it.

A. NVIDIA Fermi GPU Architecture

The NVIDIA used architecture (code-named "Fermi") has 16 streaming multiprocessors (SMs), six 64 b memory partitions, two-level cache, a host interface connecting the GPU to the CPU via a peripheral component interconnect (PCI)-Express bus, and a global scheduler (GigaThread global scheduler) that distributes blocks to SM thread schedulers. The SM is the heart of the GPU. Each one features 32/48 CUDA cores (depending on different releases), where a CUDA core is made up of an integer arithmetic logic unit and a floating point unit, both are pipelined. Sixteen load/store units allow each source and destination address to be calculated for 16 threads per block, and four special function units execute transcendental

instructions. Each SM has a set of 32-b registers and an on-chip 64-kB configurable memory, enabling the programmer to choose between 16 kB of *shared memory* and 48 kB of *cache* and *vice versa*.

Improved memory management is the real challenge when programming these devices. The GPU features an off-chip memory (divided in *global* and *local*) common to all SMs and an on-chip memory. Each SM has one set of 32-b *registers* per core, a *shared memory*, shared by all cores, and a read-only *constant* and *texture cache*, shared by all cores. The *global memory* is the biggest one on the GPU and can be read/written by the host and by all threads of a grid but with long access latencies (hundreds of clock cycles). Moreover, the *registers* are very high-speed on-chip memories, but each SM has only 32,768 registers split among all its blocks and allocated to individual threads. Finally, the GPU has a *configurable shared memory* per SM, i.e., the same fast on-chip memory used both as cache and shared memories. It is partitioned among SM thread blocks and features low access latency.

B. VCA Implementation on NVIDIA Fermi GPUs

A suitable profile of the original implementation allowed identifying intensive code parts. Apart from images of different sizes, the input parameter for the algorithm is the variable number of searched endmembers (hereinafter, "p"). At the beginning, we profiled with a small p and then increase it.

With a small p, we found out that the operations involved in SVD and PCA techniques are the bottleneck of the algorithm, taking about 60% of its execution time. Specifically, the most time-consuming part was the calculation of the covariance matrix $\mathbf{R}\mathbf{R}^{\mathrm{T}}/N$. In PCA, moreover, zero-mean data are considered, i.e., $([\mathbf{R}]_{:,j} - \overline{r})([\mathbf{R}]_{:,j} - \overline{r})^{\mathrm{T}}/N$, where $\overline{r}$ is the sample mean of $[\mathbf{R}]_{:,j}$ for $j = 1, \ldots, N$. So, we decided to implement this calculation on the GPU.

In a typical CUDA program, we need to allocate memory on the device, transfer data from the host (CPU) to the GPU, define the CUDA execution kernel mode, transfer data back from GPU to CPU memories, and free the device memory when it is no longer needed. Since a typical hyperspectral data cube can generally fit within the GPU global memory (e.g., the size of a data cube collected by AVIRIS is about 140 MB), the entire image $\mathbf{R}$ is first sent to the device memory. Then, the execution mode (i.e., the size and dimension of the processing grid and of each block) needs to be defined. Since we considered the acquired image as a 2-D structure stored in a matrix, we organized the grid and block sizes accordingly. The matrix product between $\mathbf{R}$ and its transpose results in a square matrix, and we opted for using square blocks, since performance improves if the geometry of the blocks reflects the geometry of the result. In our strategy, we associated each grid thread to an element of the result matrix $\mathbf{R}\mathbf{R}^{\mathrm{T}}$ to perform additions and multiplications associated to each element. We chose the grid size to have enough threads covering the entire surface to be processed. Since thread blocks must have the same dimension, some threads could come out from the domain; so, we added a proper control mechanism. In order to increase performance, we avoided calculations in the global memory and focused only on the shared memory.

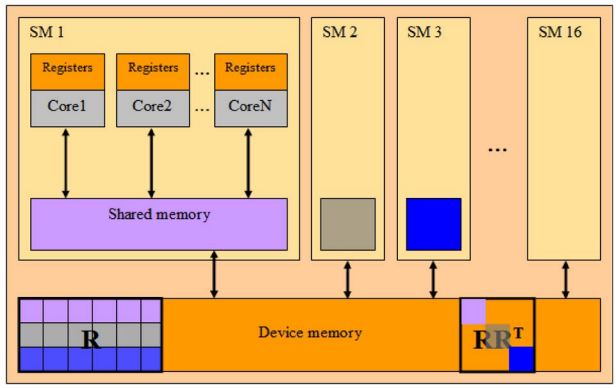

Fig. 1. Execution on GPU. In purple, gray, and blue are three of the blocks of the result matrix. The matrix $\mathbf{R}$ is divided into submatrices of the same size as the thread blocks in RR^{T}. Those blocks are marked with the same color and need the corresponding blocks of the result matrix.

During a processing loop, the input matrix $\mathbf{R}$ is divided into submatrices equal to the block size, so that each thread within the same block assigned to an SM loads an element. The process is iterated to ensure that each block of the result matrix considers all blocks of $\mathbf{R}$ necessary to the calculation of the product lines. An example of this procedure is shown in Fig. 1. The input matrix and result matrix are both allocated on the device memory. $\mathbf{R}\mathbf{R}^{\mathrm{T}}$ is divided into submatrices with thread block size, three of which are shown in Fig. 1 in purple, gray, and blue. Threads within the blocks calculate, in a loop, the corresponding elements of the result matrix. In order to perform the product, we need to consider the entire row and column of the input matrix. The matrix $\mathbf{R}$ is therefore ideally subdivided into submatrices of the same size of the thread blocks in $\mathbf{R}\mathbf{R}^{\mathrm{T}}$ (the blocks in $\mathbf{R}$ needed by the corresponding block in the result matrix are marked with the same color in Fig. 1). Each time the loop is executed, threads within a block load in shared memory a submatrix of $\mathbf{R}$ and $\mathbf{R}^{\mathrm{T}}$, calculating a partial result. The loop is performed until all elements needed by a thread block are considered. Furthermore, those threads which are out of domain (i.e., those threads that do not correspond to any element of the result matrix) are separately managed. Once the result matrix is calculated, the program control returns to CPU, which is responsible for executing the remaining instructions.

Increasing the number of endmembers (tests have been performed until values of $p = 90$ endmembers), execution time grows up as $2p^2N$. In other words, new portions of code became relevant, specifically related to the dimensionality reduction and to the VCA core. In the first case, operations involved in the projection of data onto the subspace become relevant. These are those involved in the calculation of $\mathbf{U}_{\mathbf{d}}^{\mathrm{T}}\mathbf{R}$. In the second case, the heavy core is the projective projection $\mathbf{Y}$ onto direction $\mathbf{f}$, i.e., $\mathbf{f}^{\mathrm{T}}\mathbf{Y}$. As an example, Table II reports the time needed for the Matlab processing of a real hyperspectral image (collected over the city of Pavia, Italy, and described in the next section) using 58 endmembers. However, all the found bottlenecks are again multiplications. Thus, we exploited previous implementations and developed new CUDA fast functions.

We also conceived a second parallel version of VCA using the CUBLAS library and a third only using the CULA library. CUBLAS and CULA exploit the computational power of the

TABLE II
PROFILING THE OPERATIONS INVOLVED IN THE PROCESSING OF A REAL HYPERSPECTRAL IMAGE (ROSIS PAVIA) USING MATLAB

Instructions	Time [sec]
$\mathbf{R}\mathbf{R}^{\mathrm{T}}$	1.45
$\mathbf{U}_{\mathrm{d}}{}^{\mathrm{T}}\mathbf{R}$	0.39
$\widehat{\mathbf{M}}$	1.58
$\mathbf{f}^{\mathrm{T}}\mathbf{Y}$	1.93

TABLE III
MAIN GPU FEATURES

	NVIDIA GTX 460
CUDA core	336
Clock Frequency (MHz)	1350
Memory available (MB)	993 DDR5
Memory Clock (MHz)	1800
Memory interface (bit)	256
Memory bandwidth (GB/s)	115.2

GPU to reduce the execution time of complex mathematics using a set of built-in functions. The next section provides an experimental comparison of the CUBLAS and CULA implementations (with respect to original Matlab and our C code) using a variety of hyperspectral scenes collected by different sensors. Here, we do not give results obtained with CUDA since CUBLAS and CULA versions performed better.

IV. PERFORMANCE EVALUATION

We used four images with different sizes and an NVIDIA GPU with 336 cores and performed the tests under different operating systems. Table III shows the features of the GPU used in experiments, which was connected to the PC using PCI-Express 2.0 bus, working at a frequency of 250 MHz. The PC was equipped with an Intel Core i7 as main processor (CPU), working at 2.93 GHz, with 8 GB RAM. The tests were performed on 64 b Microsoft Windows 7 and Fedora 13 operating systems, the latter with Linux kernel 2.6.33. In both cases, the CUDA development environment 4.0 (with CUBLAS libraries) and CULA R12 library were installed.

The four test images considered in our experiments are as follows: a small hyperspectral one collected by the hyperspectral digital imagery collection experiment (HYDICE) over a so-called "forest radiance" environment (169 bands $\times$ 64 lines $\times$ 64 columns, 1.32 MB), two AVIRIS scenes collected over the Cuprite mining district in Nevada ($188 \times 250 \times 191$, 17.12 MB) and the World Trade Center (WTC) region ($224 \times 512 \times 614$, 134.31 MB), and a hyperspectral image collected by the Reflective Optics Imaging Spectrographic System (ROSIS) over the city of Pavia, Italy ($102 \times 2000 \times 512$, 199.22 MB).

In Fig. 2, the Matlab processing times are given for the considered images, for $3 \leq p \leq 90$. They confirm the dependence of the algorithm complexity on the image size and on the number of endmembers. These times are obtained with a parallel execution on four cores of the i7 processor. Similarly, we run a CULA implementation with the same endmember range, both on Windows 7 and on Linux (Fedora 13) operating systems.

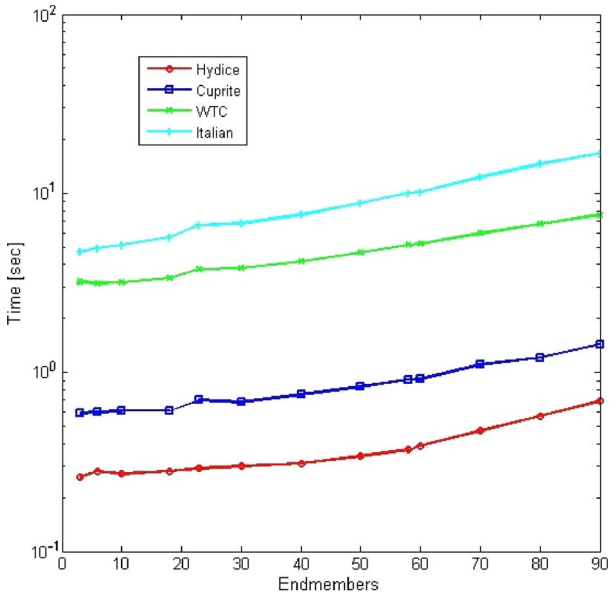

Fig. 2. Matlab execution times (log scale) on a quadcore PC for the four considered images. The VCA complexity is $2p^2N$ [5], where p is the number of endmembers and N is the number of pixels.

Execution times on Windows 7 scale from 0.36 to 0.87 s for HYDICE, from 0.72 to 1.78 s for Cuprite, from 3.28 to 8.75 s for WTC, and from 4.91 to 20.65 s for Italian. Basically, Linux acceleration sets around 20%.

Although, in these tests, we consider different numbers of endmembers, the actual estimated number for each considered scene by the hyperspectral signal identification by minimum error method [12] is given in Table IV, which also shows the processing results obtained by the GPU implementations, as well as by the original Matlab implementation, a parallel (*Matlab P*) version running on the four i7 cores and a serial (*Matlab S*) version, and our C version. Note that all the execution times reported are obtained as the mean of multiple (18) iterations. The relative standard deviations are 1.33% for Cuprite, 1.68% for HYDICE, 2% for WTC, and 1.5% for Italian. It is remarkable that Matlab serial is faster than C serial implementation. This is because it is optimized for matrix calculations, despite that our C implementation was optimized using several compilation flags such as $-$O3. By looking at the serial and parallel versions, CUBLAS and CULA implementations provide execution times that are lower in both Windows and Linux implementations than Matlab. Notice how the transfer times from the CPU and the GPU memories are greater than calculations. After profiling the GPU code with NVIDIA Visual Profiler 4.0, we found that memory copies are the most time consuming, taking about 69%, 45%, 29%, and 26% of GPU execution time for the Italian, WTC, Cuprite, and HYDICE scenes, respectively. In the first case, the high percentage is due to the high number of estimated endmembers and to the filling of the available memory. This shows that, although the algorithm scales well with the number of endmembers and image size $(2p^2N)$, increasing any of these parameters will significantly augment the memory transfers. In fact, while the achieved speedup between the CUBLAS Linux and the Matlab-P runs is close to 20% with a 58-endmember

TABLE IV
PROCESSING TIMES OBTAINED BY DIFFERENT IMPLEMENTATIONS OF THE VCA ALGORITHM WHEN PROCESSING A VARIETY OF HYPERSPECTRAL SCENES

IMAGE (endmembers)	Windows 7					Fedora 13		
	Matlab P. [sec]	Matlab S. [sec]	C Serial [sec]	CUBLAS [sec]	CULA [sec]	C Serial [sec]	CUBLAS [sec]	CULA [sec]
Hydice (18)	0.439	0.413	0.401	0.319	0.375	0.32	0.24	0.28
Cuprite (18)	0.996	1.172	3.853	0.647	0.811	3.65	0.54	0.61
WTC (23)	5.143	7.714	40.139	3.076	4.025	34.15	2.72	3.76
Italian (58)	11.567	18.234	112.695	11.31	12.511	113.26	8.84	9.99

image, the same speedup gets near to 50% for the WTC case (23 endmembers). Finally, the implicit serial structure of VCA prevents from overlapping memory copies and kernels on GPU. In any case, the achieved processing times meet the real-time processing requirement. While the AVIRIS scanning rate is 12 Hz, more recent satellite instruments such as Hyperion [13] or Environmental Mapping and Analysis Program [14] feature 220 and 230 Hz cross-line scanning rates, respectively. This means that a hyperspectral image like the AVIRIS WTC one (a typical AVIRIS data cube with 614×512 pixels and 224 spectral bands) could be collected in about 5 s. For the Italian image, the number of lines to be processed is 2000, so the real-time deadline should be set around 20 s. These requirements are fully satisfied by our parallel implementation on the considered GPU architecture.

V. CONCLUSION AND FUTURE LINES

In this letter, we have described a first real-time implementation of the VCA algorithm for endmember extraction and spectral unmixing of hyperspectral data. Our implementation has been tested on an NVIDIA Fermi GPU and is faster than the corresponding C version and the original Matlab codes (both serial and parallel). The obtained results are very encouraging to employ this technology in airborne platforms also in terms of costs (around 150 euros), really lower than the compared quadcore board; to improve performance, we suggest the use of newer NVIDIA cards which dedicate more resources to calculations while remaining quite cheap. Thus, in terms of technology evolution (particularly, memory capability), GPUs can be considered as a valuable alternative for achieving the requested computational power in remote sensing. However, GPUs still suffer some limitations for space-based processing onboard satellite instruments, such as high temperatures and energy consumption rate (i.e., NVIDIA GTX 460 works optimally under 104 °C with a maximum power consumption of 160 W). Moreover, the memory bandwidth is a well-known performance limitation, although next-generation boards will double the bus capability with a PCI 3.0 interface. At present, a possible alternative is nForce NVIDIA boards, where the bandwidth seems better, but the costs are different. These considerations, together with the GPU sensitivity to space radiations, lead us to direct our future efforts toward designing a special-purpose computing unit for implementing the VCA, possibly exploiting FPGA technology. A preliminary analysis has been already done in this direction. While the VCA elaboration seems quite straightforwardly implementable, the critical point could be represented by the dimensional reduction performed through SVD, which is very resource consuming also in performing device like Altera Stratix IV, and by the memory throughput needed to assure a real-time elaboration. Double data rate 3 synchronous dynamic RAM memories seem suitable to the aim of this study.

REFERENCES

[1] 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," *Remote Sens. Environ.*, vol. 65, no. 3, pp. 227–248, Sep. 1998.

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

[4] A. Plaza, Q. Du, J. M. Bioucas-Dias, X. Jia, and F. A. Kruse, "Foreword to the special issue on spectral unmixing of remotely sensed data," *IEEE Trans. Geosci. Remote Sens.*, vol. 49, no. 11, pp. 4103–4110, Nov. 2011.

[5] J. M. P. Nascimento and J. M. Bioucas Dias, "Vertex component analysis: A fast algorithm to unmix hyperspectral data," *IEEE Trans. Geosci. Remote Sens.*, vol. 43, no. 4, pp. 898–910, Apr. 2005.

[6] A. Plaza, D. Valencia, J. Plaza, and C.-I. Chang, "Parallel implementation of endmember extraction algorithms from hyperspectral data," *IEEE Geosci. Remote Sens. Lett.*, vol. 3, no. 3, pp. 334–338, Jul. 2006.

[7] 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.

[8] W. Luo, "Parallel VCA algorithm for hyperspectral remote sensing image in SMP cluster environment," in *Proc. CISP*, 2010, pp. 2216–2220.

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

[10] A. Gonzalez, J. Resano, D. Mozos, A. Plaza, and D. Valencia, "FPGA implementation of the PPI algorithm for remotely sensed hyperspectral image analysis," *EURASIP J. Adv. Signal Process.*, vol. 2010, no. 68, pp. 1–13, Feb. 2010.

[11] A. Plaza, Q. Du, Y.-L. Chang, and R. L. King, "High performance computing for hyperspectral remote sensing," *IEEE J. Sel. Topics Appl. Earth Obs. Remote Sens.*, vol. 4, no. 3, pp. 528–544, Sep. 2011.

[12] S. Sànchez, A. Paz, G. Martín, and A. Plaza, "Parallel unmixing of remotely sensed hyperspectral images on commodity graphics processing units," *Concurrency Comput.—Pract. Exp.*, vol. 23, no. 13, pp. 1538–1557, Sep. 2011.

[13] J. Setoain, M. Prieto, C. Tenllado, and F. Tirado, "GPU for parallel on-board hyperspectral image processing," *Int. J. High Perform. Comput. Appl.*, vol. 22, no. 4, pp. 424–437, Nov. 2008.

[14] J. M. Bioucas Dias and J. M. P. Nascimento, "Hyperspectral subspace identification," *IEEE Trans. Geosci. Remote Sens.*, vol. 46, no. 8, pp. 2435–2445, Aug. 2008.

[15] J. S. Pearlman, P. S. Barry, C. C. Segal, J. Shepanski, D. Beiso, and S. L. Carman, "Hyperion, a space-based imaging spectrometer," *IEEE Trans. Geosci. Remote Sens.*, vol. 41, no. 6, pp. 1160–1173, Jun. 2003.

[16] T. Stuffler, K. Förster, S. Hofer, M. Leipold, B. Sang, H. Kaufmann, B. Penné, A. Mueller, and C. Chlebek, "Hyperspectral imaging—An advanced instrument concept for the EnMAP mission," *Acta Astronaut.*, vol. 65, no. 7/8, pp. 1107–1112, Oct./Nov. 2009.