Fast Spatial Preprocessing for Spectral Unmixing of Hyperspectral Data on Graphics Processing Units

Jaime Delgado, Gabriel Martín, Javier Plaza, *Member, IEEE*, Luis Ignacio Jiménez, *Student Member, IEEE*, and Antonio Plaza, *Fellow, IEEE*

***Abstract*—Spectral unmixing is an important technique for hyperspectral data exploitation. It amounts at finding a set of pure spectral signatures (endmembers) of the most representative materials in the scene, and estimating their abundance fractions. The integration of spatial information prior to spectral unmixing of hyperspectral data has attracted much attention in recent years. Several approaches have been developed for the purpose of guiding endmember identification algorithms to spatially representative, yet spectrally pure endmembers. In particular, the spatial preprocessing (SPP) algorithm can be used prior to most existing spectral-based endmember identification techniques, thus promoting the selection of endmembers in spatially representative areas of the scene. However, most SPP techniques are computationally expensive, which adds significant burden to the spectral unmixing process. In this paper, we present three parallel implementations of SPP that have been specifically developed for commodity graphics processing units (GPUs). We perform an evaluation of these techniques using two GPU architectures from NVidia: GeForce GTX 580 and GeForce GTX 870M, which reveals that real-time processing performance can be obtained for real hyperspectral data sets collected by the airborne visible infra-red imaging spectrometer (AVIRIS).**

***Index Terms*—Endmember identification, graphics processing units (GPUs), hyperspectral imaging, spectral unmixing, spatial preprocessing (SPP).**

I. Introduction

SPECTRAL unmixing amounts at estimating the abundance of pure spectral signatures (called endmembers) in each mixed pixel of a hyperspectral image. Mixed pixels arise due to insufficient spatial resolution and other phenomena [1], [2]. A challenging problem is how to automatically identify endmembers, as the presence of mixed pixels generally prevents the localization of pure spectral signatures in transition areas between different land-cover classes. A possible strategy to address this problem is to guide the endmember identification process to spatially homogeneous areas, expected to contain the purest signatures available in the scene [3]–[5].

For this purpose, several spatial preprocessing (SPP) methods have been used prior to endmember identification [6]–[8]. The SPP in [6] introduces the spatial information in the endmember extraction process, so that the preprocessing can be combined with classic methods for endmember identification [9]. In this way, the endmembers can be obtained based on spatial and spectral features. An extension of this concept was presented in [7], in which the use of fixed spatial neighborhoods adopted by SPP was replaced by the incorporation of regions intended to better characterize the spatial context. However, the RBSPP strongly depends on a prior region growing algorithm that makes the procedure sensitive to the selected technique for region segmentation. Also, region growing algorithms are difficult to implement in parallel due to their irregular nature. Another technique is the spatial and spectral preprocessing (SSPP) presented in [8], which integrates both spatial and spectral information at the preprocessing level. This technique also includes a clustering step that makes the parallelization difficult. Out of the techniques discussed in [6]–[8], the most amenable one for parallel implementation is the SPP in [6] due to the regularity of its computations, which can be exploited in latest-generation hardware accelerators such as commodity graphics processing units (GPUs).

Despite the availability of several techniques to accelerate the performance of spectral unmixing algorithms on GPUs [10]–[13], no efficient implementations of SPP techniques to be used prior to spectral unmixing have been presented thus far in the literature. In this paper we present, for the first time in the literature, three different parallel implementations of the SPP algorithm in [6] using GPUs (with different memory management strategies). These versions have been implemented using NVidia's compute device unified architecture (CUDA),[1] and tested on two different NVidia GPU architectures: GeForce GTX 580 and GeForce GTX870M, using two different images collected by NASA's AVIRIS over the Cuprite mining district in Nevada and the World Trade Center in New York City. We also presented an exhaustive assessment of the performance of these different versions in terms of memory stalls and cache hit rates. Our experimental validation shows that a significant reduction in the execution time can be achieved allowing the integration of this preprocessing step in a fully operational unmixing chain, which exhibits real-time performance with regards to the time that the AVIRIS sensor takes to collect the data.

This paper is structured as follows. Section II outlines the SPP technique considered in this work. Section III describes

Manuscript received April 22, 2015; revised September 11, 2015; accepted October 15, 2015. Date of publication November 24, 2015; date of current version February 09, 2016. This work was supported by the Portuguese Science and Technology Foundation under Project UID/EEA/50008/2013, Project PTDC/EEI-PRO/1470/2012, and Project SFRH/BPD/94160/2013.

J. Delgado, J. Plaza, L. I. Jiménez, and A. Plaza are with the Hyperspectral Computing Laboratory, Department of Computer Technology and Communications, University of Extremadura, Cáceres E-10071, Spain (e-mail: jplaza@unex.es; luijimenez@unex.es; aplaza@unex.es).

G. Martín is with the Instituto de Telecomunicações, Lisbon 1049-1, Portugal (e-mail: gabriel.hernandez@lx.it.pt).

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/JSTARS.2015.2495128

[1] [Online]. Available: http://www.nvidia.com/cudazone

the GPU implementations of the SPP algorithm. Section IV presents an experimental evaluation of the proposed implementation in terms of both accuracy and parallel performance. Section V concludes this paper with some remarks and hints at plausible future research lines.

II. Spatial Preprocessing

The main idea behind the SPP framework is to estimate, for each input pixel vector in the hyperspectral image, a scalar factor ρ which is intimately related to the spatial similarity between the pixel and its spatial neighbors, and then use this scalar factor to spatially weight the spectral information associated to the pixel [6]. Let $\mathbf{y}_{i,j}$ represent the pixel in spatial coordinates i, j. With this notation in mind, the scalar factor is calculated as follows:

$$\alpha(i,j) = \sum_{r=i-d}^{i+d} \sum_{s=j-d}^{j+d} \beta[r-i, s-j] \cdot \gamma[\mathbf{y}_{i,j}, \mathbf{y}_{r,s}] \quad (1)$$

where $\mathbf{y}_{i,j}$ is the pixel for which we are calculating the scalar factor, and $\mathbf{y}_{r,s}$ is the spatial neighbors. Here, d represents half of the window size, so that the full window size is $ws = 2 \cdot d + 1$. The γ function computes the spectral angle between the pixel and its neighbors, and the β function computes a weight factor based on the distance between the pixel and the neighbors. The spectral angle is computed as follows:

$$\gamma[\mathbf{y}_{i,j}, \mathbf{y}_{r,s}] = \arccos \frac{< \mathbf{y}_{i,j}, \mathbf{y}_{r,s} >}{\|\mathbf{y}_{i,j}\| \cdot \|\mathbf{y}_{r,s}\|} \quad (2)$$

where $< \cdot, \cdot >$ denotes the dot product between two vectors and $\| \cdot \|$ denotes the euclidean norm of a vector. As we can see in (3), the closest neighbors are given more relevance. Also, the β function is normalized to sum to one as follows:

$$\beta(a,b) \propto \frac{1}{a^2 + b^2}. \quad (3)$$

Once the scalar factor has been computed, every pixel is displaced to the simplex centroid depending on the scalar factor. Expressions (4) and (5) show how to displace the image pixels depending on the scalar factor, as detailed in [6]

$$\rho(i,j) = \left(1 + \sqrt[2]{\alpha(i,j)}\right)^2 \quad (4)$$

$$\mathbf{y}_{i,j}{}' = \frac{1}{\rho(i,j)}(\mathbf{y}_{i,j} - \bar{\mathbf{c}}) + \bar{\mathbf{c}} \quad (5)$$

where $\bar{\mathbf{c}}$ is the simplex centroid, computed as the average of all the image pixels; $\mathbf{y}_{i,j}{}'$ is the new displaced pixel; and $\mathbf{y}_{i,j}$ is the original pixel at the spatial coordinates i, j. Finally, n_l and n_c are the number of lines and columns of the hyperspectral image, respectively.

Fig. 1 provides a graphical representation of how the SPP technique works. This figure illustrates a toy example in which only two bands of an input hyperspectral image are represented against each other for visualization purposes. As Fig. 1 illustrates, the idea behind SPP is to center each spectral feature in the data cloud around its mean value, and then shift each feature

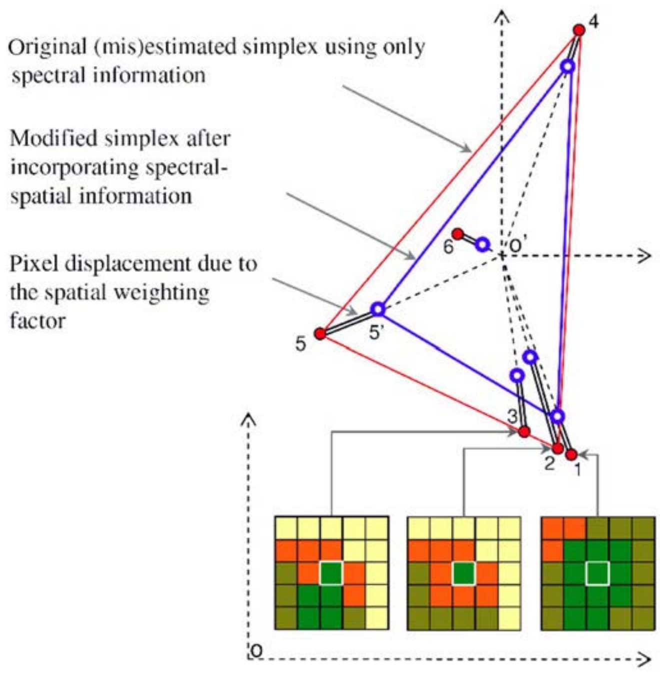
Original (mis)estimated simplex using only spectral information
Modified simplex after incorporating spectral-spatial information
Pixel displacement due to the spatial weighting factor
4
6
0'
5
5'
3
2
1
0

Fig. 1. Graphical illustration of the SPP technique.

toward the centroid of the data cloud. The shift of each spectral feature is proportional to a similarity measure calculated using both the spatial neighborhood around the pixel under consideration and the spectral information of the pixel. The correction is performed so that pixels located in spatially homogeneous areas (such as pixel 1 in Fig. 1) are expected to have a smaller displacement with regards to their original location in the data cloud than pure pixels surrounded by spectrally distinct substances (such as pixels 2 and 3 in Fig. 1). Resulting from the above operation, a modified simplex is formed (in blue in Fig. 1) with regards to the original one (in red in Fig. 1), using not only spectral but also spatial information.

III. GPU Architecture

In this section, we provide a description of the NVidia CUDA architecture, which can be seen as a set of multiprocessors (MPs), where each one is characterized by a single instruction multiple data (SIMD) architecture, i.e., in each clock cycle each processor executes the same instruction but operating on multiple data streams. Each processor has access to a local shared memory and also to local cache memories in the MP, whereas the MPs have access to the global GPU (device) memory (see Fig. 2). Unsurprisingly, the programming model for these devices is similar to the architecture lying underneath. GPUs can be abstracted in terms of a stream model, under which all data sets are represented as streams (i.e., ordered data sets). Algorithms are constructed by chaining so-called kernels, which operate on entire streams and which are executed by an MP, taking one or more streams as inputs and producing one or more streams as outputs. Thereby, data-level parallelism is exposed to hardware, and kernels can be concurrently applied without any sort of synchronization. As shown in Fig. 3, kernels can perform a kind of batch processing arranged in the

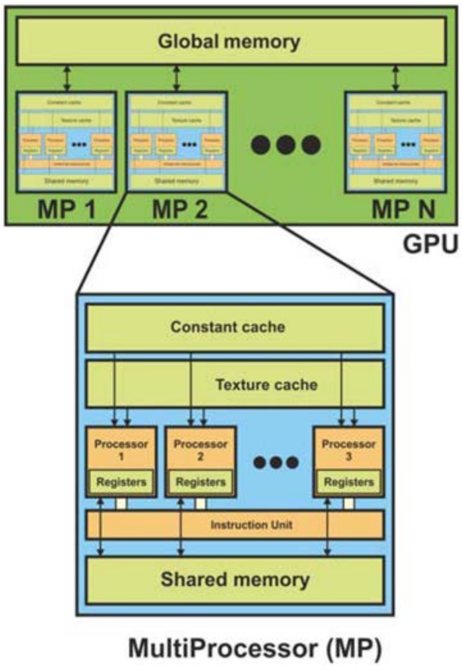

Fig. 2. GPU architecture, where the GPU can be seen as a set of MPs.

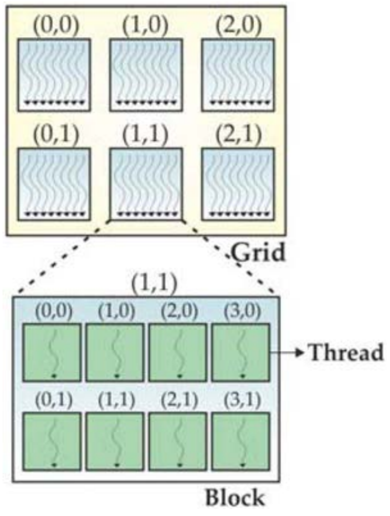

Fig. 3. GPU batch processing in the GPU: grids of blocks, where each block is composed by a group of threads.

form of a grid of blocks, where each block is composed by a group of threads that share data efficiently through the shared local memory and synchronize their execution for coordinating accesses to memory.

IV. GPU Implementations

In this section, we describe the parallel implementations of SPP in the GPU. The implementation is based on four main kernels as follows.

1) The first kernel computes the centroid of the simplex $\overline{\mathbf{c}}$ as the average of all the image pixels. Here, the number of blocks is equal to the number of bands and each block computes the average of each band using a reduction operation as described in [14].
2) The second kernel computes the euclidean norms of each image pixel. The output of this kernel will be used in the following kernel to compute the γ function in (2). Here, the number of threads T is set to the maximum value supported by the GPU. Each thread will compute the euclidean norm of a vector, thus the number of blocks B will be the number of image pixels divided by the number of threads: $B = \lceil (n_l \cdot n_c)/T \rceil$.
3) The third kernel computes the similarity factor $\alpha(i,j)$ for each pixel as given by the expression (1). In this kernel, there are as many blocks as pixels: $B = n_l \cdot n_c$, and there are as many threads as the window size: $T = ws^2 = (2d+1)^2$. Here, each thread computes the dot product between the central and the corresponding neighbor pixel in the window: $< \mathbf{y}_{i,j}, \mathbf{y}_{r,s} >$. Then, it computes $\gamma[\mathbf{y}_{i,j}, \mathbf{y}_{r,s}]$ as in (2). After that, the kernel weights this value using the β function (precomputed on the CPU). Finally, the kernel performs a reduction [14] to sum all the values inside the window. As a result, the kernel obtains $\alpha(i,j)$.
4) The fourth kernel computes the displacement to the centroid for each pixel as described in (5). In this kernel, each thread computes the displacement of one pixel. The number of threads used T is the maximum allowed by the GPU. The number of blocks used is the number of image pixels divided by the number of threads: $B = \lceil (n_l \cdot n_c)/T \rceil$.

In the following, we mainly focus on how to improve the performance of the third kernel (the most time-consuming one). Specifically, we developed different parallel implementations of the SPP that mainly differ on the adopted strategy to efficiently compute the similarity factor $\alpha(i,j)$ for each pixel and the memory management performed during this process.

In Algorithm 1, we show a pseudocode description of the third kernel. This kernel is based on the outputs provided by previous kernels. Here, $\mathbf{Y}$ denotes the hyperspectral image; Φ corresponds to the the euclidean norms of each pixel computed in the second kernel, thus $\Phi_{x,y} = \|\mathbf{y}_{x,y}\|$; β represents the scale factor described in (3); B_x, B_y for $x = 1, \ldots, n_l$ and $y = 1, \ldots, n_c$ indexes the block which will compute the α value for the central pixel of the window; T_i, T_j for $i = 1, \ldots, ws$ and $j = 1, \ldots, ws$ correspond with the relative position of the threads in the window; n_l, n_c, n_b represent the number of lines, columns, and bands of the hyperspectral image, respectively; and finally ws represents the window size. Lines 1–4 from Algorithm 1 compute the dot product between the central pixel of the window $\mathbf{Y}_{B_x,B_y}$ and the relative pixel in the window $\mathbf{Y}_{T_i,T_j}$. Line 5 computes the γ function in (2) and line 6 computes the product between β and γ in (1). Finally, lines 8–13 perform the sum of all the values for all the pixels inside the window. Here, the sum of all values is performed in parallel between all the threads belonging to each block in an accumulative way, using a reduction as described in [14]. As a result, we obtain the α value as described in (1). In Sections

Algorithm 1. Similarity factor calculation CUDA kernel

```
Input: $\mathbf{Y}, \Phi, \beta, B_x, B_y, T_i, T_j, n_l, n_c, n_b, ws$
Output: $\alpha$
    Initialisation:
 1: $\Delta_{T_i,T_j} := 0$
    Dot product:
 2: for $k = 1$ to $n_b$ do
 3:     $\Delta_{T_i,T_j} = \Delta_{T_i,T_j} + \mathbf{Y}_{B_x,B_y,k} \cdot \mathbf{Y}_{T_i,T_j,k}$
 4: end for
    Spectral angle distance:
 5: $\Delta_{T_i,T_j} = cos^{-1}(\Delta_{T_i,T_j}/(\Phi_{B_x,B_y} \cdot \Phi_{T_i,T_j})$
    Weighting:
 6: $\Delta_{T_i,T_j} = \beta_{T_i,T_j} \cdot \Delta_{T_i,T_j}$
    Sum of all values using a reduction:
 7: $l = T_i \cdot ws + T_j$
 8: for $k = \lfloor \log_2(ws \cdot ws) \rfloor - 1$ to $k = 1$ step $k = k/2$ do
 9:     if $(l < k)$ then
10:         $\Delta_l = \Delta_l + \Delta_{l+k}$
11:     end if
12: end for
    Final result:
13: $\alpha_{B_x,B_y} = \Delta_0$
14: return $\alpha_{B_x,B_y}$
```

Section IV-A, Section IV-B, and Section IV-C, detailed descriptions of the implemented versions of the kernel in Algorithm 1 are provided.

A. One Pixel Per Block

The first implementation (denoted hereinafter as 1M-SPP) processes one pixel per each block using main memory to store the required data. This kernel uses the shared memory and the GPU registers to store the Δ values in Algorithm 1, but $\mathbf{Y}$, β, and Φ matrices are stored and read from the global (video) memory of the GPU. In this case, the grid configuration is a mesh of $n_l \times n_c$ blocks, each one containing $ws \times ws$ threads. As we can see in Fig. 4, different blocks will access the neighboring pixels to process the corresponding pixels (i.e., P1 and P2). As shown in Fig. 4(b), both of them need to process their overlapping neighbors (represented in yellow color) thus the memory access to those neighbors are duplicated. If we extrapolate this issue to the case of several blocks, it results in a significant decrease of parallel performance. With the aim of optimizing the memory access to common neighboring pixels, we propose the following alternative implementations.

B. Several Pixels Per Block

The second implementation (denoted hereinafter as NM-SPP) processes several pixel per each block. The main goal of this implementation is to use the first level of cache memory in the GPU (hereinafter, L1-cache) to cache the memory accesses to $\mathbf{Y}$. It should be noticed that both L1-cache and shared memory are much faster than the local and global memories, mainly due to the fact that they are on-chip memories. In the previous version (1M-SPP), each block is in charge of processing one pixel (accessing also the pixels that are in the defined spatial window). In the GPU architecture, different blocks are executed in different MPs and, thus, they cannot use the shared memory or the L1-cache to avoid several repeated memory accesses, as illustrated in Fig. 4(b).

The main idea behind NM-SPP is to accelerate memory accesses to common neighbors using the L1-cache. The latest NVidia architectures: Kepler and Fermi include cache systems and, therefore, if we process several pixels in the same block [see red region in Fig. 4(d)], the access to the common neighbors pixels in the block [see orange region in Fig. 4(d)] will be effectively cached, thus improving the performance. For instance in Fig. 4(c), when the pixel P1 is processed the memory in blue will be accessed, thus it is likely that, when the pixel P2 (in green) is processed, the yellow memory positions are already in the L1-cache [see Fig. 4(e)]. Furthermore, the access to the β matrix can also be cached for the pixels processed inside the same block. In this case, the number of pixels processed by each block will be $P = \lfloor T_{\max}/ws^2 \rfloor$ where $T_{\max}$ is the maximum number of threads supported by the architecture; therefore, the number of blocks in NM-SPP will be $B = \lceil (n_l \cdot n_c)/P \rceil$ and each block will contain $T = ws^2 \cdot P$ threads.

C. Several Pixels Per Block Using Shared Memory

Our third implementation (denoted hereinafter as NS-SPP) again processes several pixels per each block, but this time we use shared memory to store the $\mathbf{Y}$ data positions that must be accessed by the kernel threads. As mentioned in Section IV-B, L1-cache and shared memory are much faster than global and local memories. The shared memory is allocated per thread block, so that all threads in the same block can access the same portion of shared memory. Therefore, a thread can access the shared memory loaded from global memory by other threads within the same thread block. This capability allow us to implement our algorithms using a user-managed data cache approach, so that the portions of $\mathbf{Y}$ that the threads need to access are stored in the shared memory. This means that each required data position must be accessed just once. For instance, in Fig. 4(d), the orange region denotes the image positions that are going to be accessed. If we load these data in shared memory, as we can see in Fig. 4(e), then the threads directly access the data in a memory, which is much faster than the global memory, thus significantly improving the performance.

However, in this case, the GPU cores have to perform all the computations relative to the shared memory control (i.e., precompute the index of the memory positions that must be loaded and control when the loads must be performed). Due to this requirement, and considering that both the shared and cache memory effectively share the same physical memory (with the same rate and latency of accesses), the performance of this version has an overhead with regards to the NM-SPP version, in which the L1-cache control is performed by dedicated hardware of the GPU architecture. To conclude our description, we emphasize that the grid configuration for NS-SPP is the same than for NM-SPP.

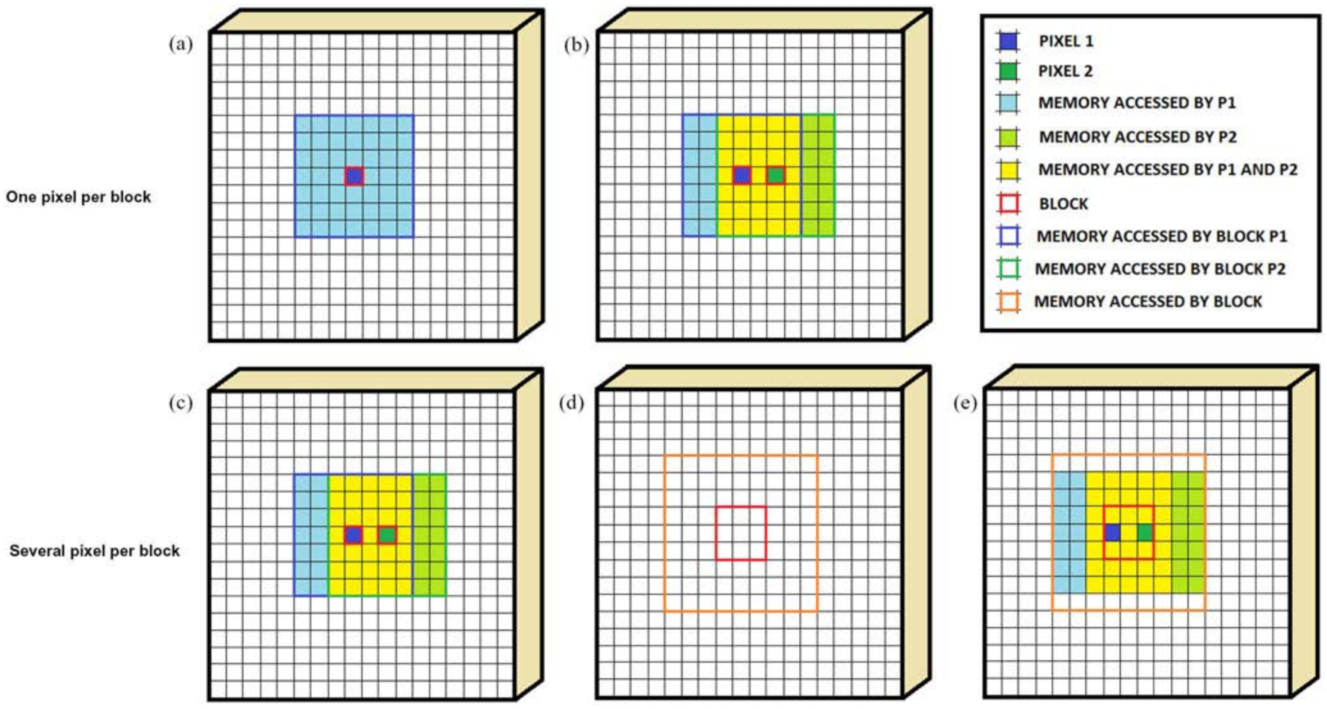

Fig. 4. Memory accessed when there is one pixel per block (a) and (b) and when there are several pixels per block (c)–(e).

V. Experimental Results

A. Hyperspectral Data Sets

In this work, we have used two different hyperspectral data sets to illustrate the performance of our newly proposed parallel implementations. The first data used in our experiments was collected by the AVIRIS instrument, operated by the NASA's Jet Propulsion Laboratory, over the Cuprite mining district in Nevada in the summer of 1997.[2] The portion used in experiments corresponds to a 350×350-pixel subset, which comprises 188 spectral bands in the range from 400 to 2500 nm and a total size of around 50 MB. Water absorption bands as well as bands with low signal-to-noise ratio (SNR) were previously removed. These data are well understood mineralogically.

The second hyperspectral image used for experiments in this work was also collected by the AVIRIS sensor over the World Trade Center area in New York City on September 16, 2001, just 5 days after the terrorist attacks that collapsed the two main towers and other buildings in the WTC complex. The full data set selected for experiments consists of 614×512 pixels, 224 spectral bands, and a total size of (approximately) 140 Mbytes. The spatial resolution is 1.7 m/pixel. Fig. 5 shows a false color composite of the two data sets selected for experiments.

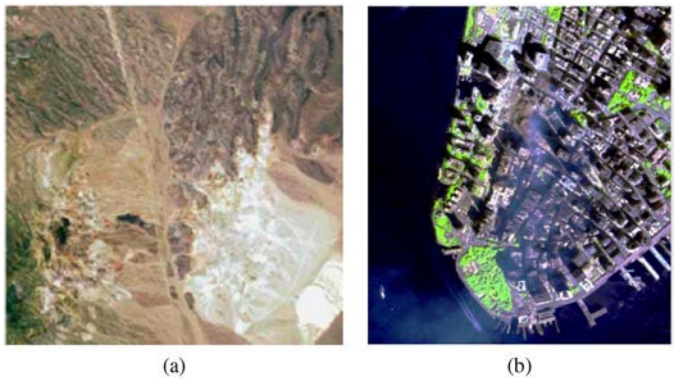

Fig. 5. False color composition of the AVIRIS hyperspectral data collected by NASA's Jet Propulsion Laboratory over (a) Cuprite mining district in Nevada and (b) World Trade Center complex in Manhattan.

B. Analysis of Algorithm Precision

In this section, we have focused on analyzing the accuracy of the parallel versions with regards to the serial implementation of the SPP algorithm. According to [15] and [16], the differences among compilers, libraries, and hardware architectures generate some residual errors which, in a cumulative process, result in a marginal difference between the serial and parallel implementations. With the aim of measuring, this error and evaluate the accuracy of the parallel implementation, we have used two different metrics.

1) Normalized root-mean-squared error (NRMSE) uses the mean value of the pixels to normalize. This metric quantifies the differences between the reference $s_i(x, y)$ (value of the pixel preprocesed by the serial version) and the estimated ones $\hat{s}_i(x, y)$ (value of the pixel preprocessed by the parallel GPU version) in one pixel of nb bands, being $\bar{s}(x, y)$ the mean value of the reference pixel. Specifically, the NRMSE expression that we have used in experiments is the following one:

[2][Online]. Available: http://aviris.jpl.nasa.gov

TABLE I
NRMSE AND MAXSDE VALUES OBTAINED AFTER COMPARING THE SERIAL IMPLEMENTATION OF SPP VERSUS THE GPU IMPLEMENTATIONS

	Window size	3	5	7	9	11	13	15
Cuprite	NRMSE ($\times 10^{-6}$)	3,28	2.85	2.61	2.45	2.32	2.22	2.14
	MaxSDE ($\times 10^{-4}$)	9.46	8.15	7.43	6.93	6.55	6.24	5.98
WTC	NRMSE ($\times 10^{-5}$)	2.69	1.60	1.23	1.04	0.91	0.83	0.76
	MaxSDE ($\times 10^{-3}$)	5.00	3.00	2.30	2.00	1.70	1.60	1.50

$$\mathit{NRMSE}(x,y) = \sqrt{\frac{\sum_{i=1}^{nb}(\hat{s}_i(x,y) - s_i(x,y))^2}{\sum_{i=1}^{nb}(s_i(x,y) - \bar{s}(x,y))^2}} \quad (6)$$

$$\text{with } \bar{s}(x,y) = \frac{1}{nb}\sum_{i=1}^{nb} s_i(x,y). \quad (7)$$

2) Maximum scaled absolute difference error (MaxSDE) uses the mean of each pixel to normalize the error [17]. This metric is computed as the ratio of the maximum absolute difference between pixels of both images [$s_i(x,y)$ and $\hat{s}_i(x,y)$] and the average absolute spectrum of that pixel using the following expression:

$$\mathit{MaxSDE}(x,y) = \max_i \left\{ \frac{nb|s_i(x,y) - \hat{s}_i(x,y)|}{\sum_{i=1}^{nb}|s_i(x,y)|} \right\}. \quad (8)$$

Values of NRMSE and MaxSDE close to zero indicate higher similarity between images and, as shown in Table I, we obtained very low error scores after comparing the serial version against the GPU implementations. It is worth noting that the difference between the serial and the GPU versions is almost nonexisting, and the output provided by the three parallel GPU versions can be considered the same.

Figs. 6 and 7 show the spatial distribution of the NRMSE and MaxSDE for the AVIRIS Cuprite and WTC data sets considering a spatial window of $ws = 5$. Again these plots indicate the differences between the results obtained by the serial and GPU implementations of SPP. As can be seen in Figs. 6 and 7, the errors are almost zero for all pixels in the considered images.

In [6], the authors state that the original SPP algorithm is used to preprocess the original hyperspectral data set prior to the selection of the endmember. Actually, SPP is applied to select the spatial coordinates of the endmembers (which are extracted over the spatial/spectral preprocesed image), whereas spectral signatures of the endmembers are finally selected from the original data set, due to the fact that SPP introduces some spectral distortion of the information. In this sense, the scores of the errors showed in Table I will be negligible if the selected endmembers are the same in the images obtained with the serial and the GPU implementations. In order to analyze this, we conducted the following experiment. We first applied the virtual dimensionality (VD) [18] algorithm to estimate the number of pure components in the two considered data sets. The estimated number of pure components was 19 for Cuprite data set and 31 for the WTC data set. Then, we used the orthogonal subspace projection (OSP) algorithm [19] in order to select the desired number of endmembers over the two preprocessed images (considering the configuration of SPP which results in a higher error in Table I, which is using $ws = 3$) and compared the obtained results. As can be seen in Fig. 8, the coordinates of the endmembers selected by ATGP algorithm over the two spatial/spectral preprocessed images are exactly the same and, therefore, we can assume that the difference between serial and GPU implementations are negligible from the viewpoint of SPP, which is the purpose of the SPP algorithm that we have implemented in parallel in this work.

C. Analysis of Parallel Performance

The GPU implementations of SPP have been tested on two different computers. 1) A desktop computer with a GPU NVidia GTX 580, which features 512 processor cores operating at 1.54 GHz, with single precision floating point performance of 1581.1 Gflops, total dedicated memory of 1536 MB, 2004 MHz memory (with 384-bit GDDR5 interface), and memory bandwidth of 192.4 GB/s.[3] The GPU is connected to an Intel core i7 920 CPU at 2.67 GHz with eight cores, which uses a motherboard Asus P6T7 WS SuperComputer. 2) A laptop computer with a GPU NVidia GTX 870M, which features 1344 processor cores operating at 967 MHz, with single precision floating point performance of 2599 Gflops, total dedicated memory of 3072 MB, 5000 MHz memory (with 192-bit GDDR5 interface), and memory bandwidth of 120 GB/s.[4] The GPU is connected to an Intel i7-4710MQ at 3.5 GHz with four cores. The serial version has been compiled using the GCC C++ compiler (version 4.8.2), whereas the GPU has been compiled using the NVCC (version 6.5.12).

As shown in Section V-B, it is important to emphasize that our GPU versions of SPP provide almost the same results as the serial version of the SPP algorithm. Hence, the only relevant difference between the serial and parallel algorithms is the time they need to complete their calculations. The serial algorithm was executed in one of the available cores of the desktop computer, and the parallel times were measured in the considered GPU platform. For each experiment, 10 runs were performed and the mean values are reported (these times were always very similar, with differences on the order of a few milliseconds only).

Considering that the NM-SPP implementation strongly depends on the available L1-cache memory, several configurations of L1-cache and shared memory were explored to achieve the best performance. On devices of compute capability 2× and 3× (which is the case for the two GPUs considered in the experiments), each MP has 64 kB of on-chip memory that can be partitioned between L1-cache and shared memory. We can select two settings: 1) 48 kB shared memory/16 kB L1-cache and 2) 16 kB shared memory/48 kB L1-cache. By default, the 48 kB shared memory setting is used. This can be configured during runtime API from the host using functions `cudaDeviceSetCacheConfig` (), which accepts one of three options: 1) `cudaFuncCachePreferNone`; 2) `cudaFuncCachePreferShared`; and 3) `cudaFuncCachePreferL1`.

[3][Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580

[4][Online]. Available: http://www.geforce.com/hardware/notebook-gpus/geforce-gtx-870m

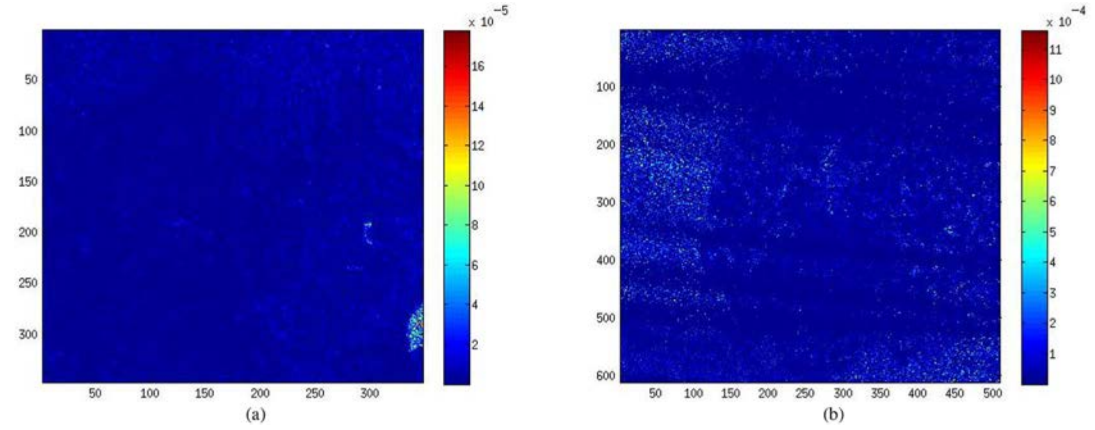

Fig. 6. Spatial distribution of the NRMSE (after comparing the serial and GPU implementations of SPP) for (a) Cuprite data set and (b) WTC data set, using a spatial window with $ws = 5$.

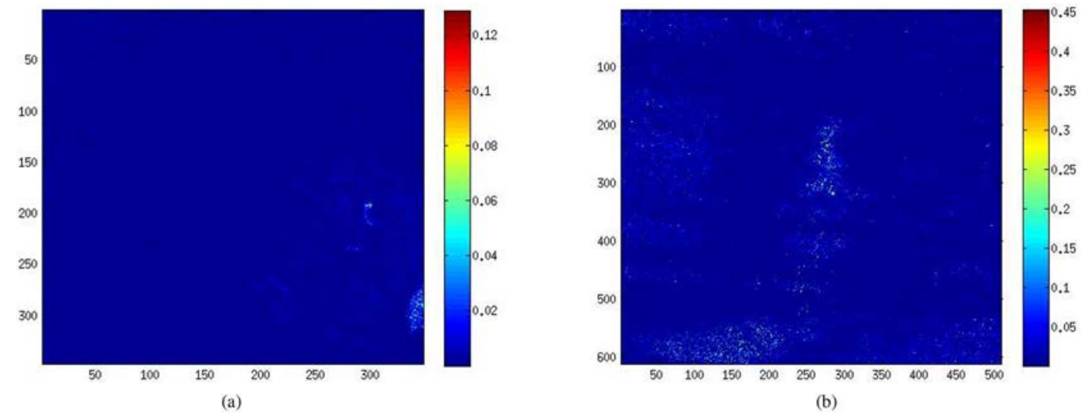

Fig. 7. Spatial distribution of the MaxSDE (after comparing the serial and GPU implementations of SPP) for (a) Cuprite data set and (b) WTC data set, using a spatial window with $ws = 5$.

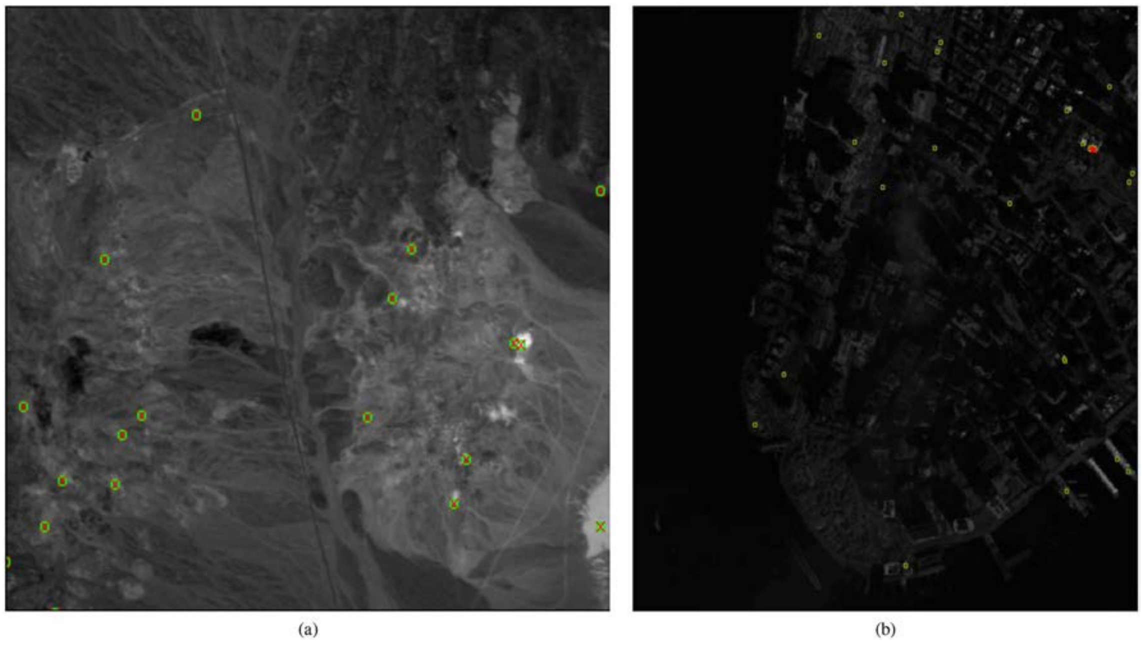

Fig. 8. Spatial coordinates of the endmembers selected by OSP for the two preprocessed images (serial in green and GPU in red) over the (a) Cuprite data set and (B) WTC data set. In all cases, the spatial coordinates of the endmembers selected by OSP from the preprocessed images obtained in serial and in parallel are exactly the same, indicating that the parallel and the serial implementations are equivalent from a functional point of view.

TABLE II
MEAN EXECUTION TIMES FOR THE PARALLEL NM-SPP IMPLEMENTATIONS CONSIDERING DIFFERENT MEMORY RATIOS BETWEEN L1-CACHE AND SHARED MEMORY, USING `cudaDeviceSetCacheConfig` (), AFTER 10 MONTE-CARLO RUNS

Window size	3	5	7	9	11	13	15
48 kB L1 GTX580	**0.35**	**0.41**	**0.49**	**0.67**	**0.89**	**1.18**	**1.75**
16 kB L1 GTX580	0.35	0.43	0.52	0.89	1.32	2.2	3.46
48 kB L1 GTX870M	**0.52**	**0.59**	**0.78**	**1.16**	**1.94**	**3.35**	**5.43**
16 kB L1 GTX870M	0.52	0.6	0.8	1.23	2.14	3.78	6.14

TABLE III
MEAN EXECUTION TIMES FOR THE PARALLEL AND SERIAL IMPLEMENTATIONS OF THE SPP ALGORITHM AFTER 10 MONTE-CARLO RUNS

Window size	3	5	7	9	11	13	15
SPP	7.86	18.63	38.87	77.72	141.80	211.27	286.97
1M-SPP GTX580	0.69	0.71	0.78	0.87	1.06	1.32	**1.62**
Speedup	11.32	26.06	50.12	89.77	133.29	159.66	**176.77**
NM-SPP GTX580	**0.35**	**0.41**	0.49	0.67	**0.89**	**1.18**	1.75
Speedup	**22.65**	**45.11**	79.23	116.01	**158.83**	**179.48**	164.03
NS-SPP GTX580	0.35	0.41	**0.48**	**0.63**	0.93	1.40	3.66
Speedup	22.26	44.97	**80.78**	**123.81**	152.71	150.44	78.45
1M-SPP GTX870M	0.98	1.03	1.06	1.29	1.6	2.07	**2.47**
Speedup	8.03	18.16	36.8	60.29	88.85	102.11	**116.28**
NM-SPP GTX870M	0.52	0.59	0.78	1.16	1.94	3.35	5.43
Speedup	15.11	31.41	49.57	67.05	72.98	63.03	52.86
NS-SPP GTX870M	**0.5**	**0.57**	**0.65**	**0.8**	**1.11**	**1.66**	5.56
Speedup	**15.87**	**32.68**	**60.07**	**97.51**	**127.29**	**127.04**	51.66

Even though L1-cache and the shared memory are located in the same on-chip hardware, there exist some differences between them. The shared memory is accessed through 32 banks, while L1-cache is accessed by cache line. With shared memory, the programme has full control over what gets stored and how (user-managed data cache), while with L1-cache, data eviction is done by the hardware according to different heuristic algorithms [20].

Table II shows the mean execution time of NM-SPP implementation considering the two possible memory settings. The best execution times for each GPU and window size are shown in bold typeface. As can be seen, increasing the size of L1-cache results in an increase of the parallel performance of the algorithm, which is able to finish execution in less than 1.8 s for all considered window sizes. This is due to the fact that more space is available to perform the data caching, thus allowing more data to be accessed from the L1-cache and, therefore, increasing the L1 hit rate, which results in less L2-cache and global (video) memory accesses. Since these memories have lower latencies, the performance is significantly improved.

Once determined that the best possible option for the NM-SPP algorithm is to reserve 48 kB of the physically available memory for the L1-cache, we continue by comparing the execution times of this version with the rest of the parallel implementations of SPP. Table III summarizes the obtained results by the C implementation and by all the three GPU implementations. An optimization has been considered for the CPU implementation, namely the inclusion of the –O3 optimization flag in the compiler. The best execution times and speedups for each GPU and window size are highlighted in bold typeface, where the speedups are calculated regarding the execution of the code over one single core of the considered architecture.

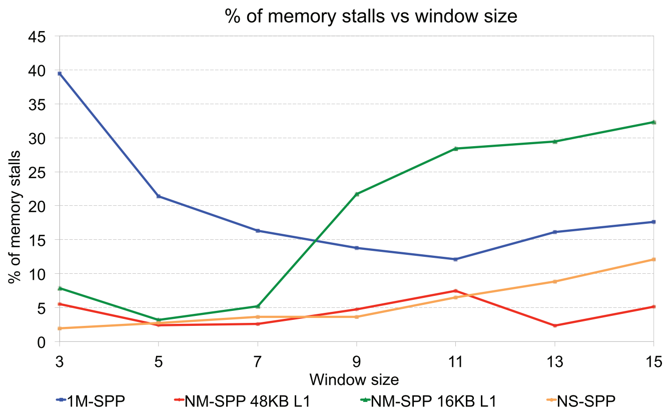

Fig. 9. Percentage of total stalls due to memory issues of parallel versions executed in GTX580.

As revealed by Table III, both NM-SPP and NS-SPP perform substantially better than 1M-SPP for reasonable window sizes. At this point, it is worth noting that selection of very large window sizes make no sense due to the fact that pixels located far away are still considered neighbors, but very low weights are assigned to the pixels near the border of the window as defined in (3). However, for very large windows sizes, 1M-SPP achieves the best results. This is due to the fact that, when the window sizes are large, the L1-cache and register capacities of the GPUs are only able to process one or very few pixels per block in the case of NM-SPP and NS-SPP, thus resulting in penalized versions of 1M-SPP. These results clearly indicate the importance of efficiently using shared and cache memories in order the best possible performance of GPU implementations. In our approach, the L1-cache version performs in most of the cases as well as the shared memory-based implementation, while the shared memory-based implementation NS-SPP provides good performance despite the extra calculations that it must tackle to maintain a fully user-managed data cache approach.

Fig. 9 shows the percentage of total stalls (the lower this number, the better) which are produced because a memory operation cannot be performed when executing the parallel versions of SPP on GTX580. It can be seen that both NM-SPP 48 kB L1 and NS-SPP versions are able to significantly reduce the percentage of memory stalls. It can also be seen that, for window sizes from 3 to 7, the NM-SPP 16 kB L1 version is also able to significantly reduce the percentage of memory stalls; however, for larger window sizes, the percentage of memory stalls increase significantly. This can be explained because the L1 cache memory saturates when the window size is larger than 7. In addition, as it would be expected, the 1M-SPP version presents a higher percentage of memory stalls.

Last but not least, Fig. 10 shows that the L1-cache hit rates (the higher this number, the better) of all parallel versions executed on GTX580. This figure shows that NM-SPP 48 kB L1 version maintain a high L1-cache hit rate through all window sizes. However, the NM-SPP 16 kB L1 reduces the hit rate drastically when the window size is larger than 7. This is expected since, as mentioned in our discussion of the results in Fig. 9, at this point the 16 kB L1 cache saturates. On the other hand, we can see that NS-SPP provides a very reduced L1 hit rate. This

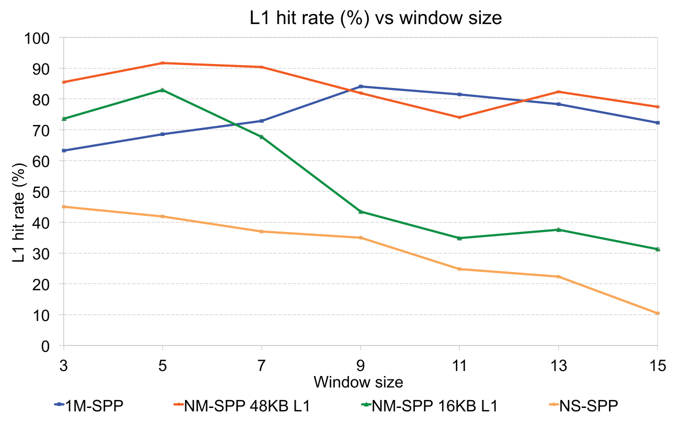

Fig. 10. L1-cache hit rate of parallel versions executed on GTX580.

is also expected because this version does not take advantage of the L1 cache, but it uses the shared memory instead. In the case of NS-SPP, most of the memory accesses are loads from global memory to shared memory which are not likely to produce L1 hits. Finally, an unexpected high percentage hit rate is observed in the 1M-SPP version, which for some windows sizes is higher than the NM-SPP 48 kB L1 version. This phenomenon can be only explained if we take into account the coalescence: in the case of NM-SPP 48 kB L1 version, more cores are accessing to the same memory positions in the same block and, thus, it is more likely that these accesses are coalesced, resulting in a lower number of memory loads. Due to the fact that Fig. 10 shows the results in relative terms (percentage of L1 hit rates), it is possible that the 1M-SPP has a higher hit rate while the number of stalls is significantly lower for the NM-SPP. It can also be seen that both metrics are inversely proportional for each version, i.e., the higher percentage of stalls for a given window size, the lower hit rate percentage for the same version with the same window size and conversely.

Before concluding this section, we emphasize that the cross-track line scan time in AVIRIS, a push-broom instrument, is quite fast [21] (8.3 ms to collect 512 full pixel vectors). This introduces the need to process the considered scene (614×512 pixels and 224 spectral bands) in less than 5.09 s to fully achieve real-time performance. As we can see in Table III, the execution times for the parallel versions of the SPP algorithm are in real time for both GPUs and for all the considered window sizes. If we analyze the results reported in [12] for full spectral unmixing chains, we can conclude that the GPU implementation of SPP can be added as the first step of such chains without sacrificing real-time processing performance. Further, the speedups reported for the GPU implementations of SPP increase with the window size, which is expected due to the fact that the windows are processed in parallel and the larger the windows the more computations can be performed in parallel. In the best case, a speedup of about $180\times$ with regards to the serial version is reported, which represents a significant improvement over the (optimized) serial implementation of SPP.

VI. Conclusion and Future Lines

In this paper, we have presented three different GPU implementations of an SPP algorithm designed to include spatial information in spectral unmixing of hyperspectral data. The proposed implementations are based on different memory management policies and are highly scalable and efficient, achieving real-time results in two different GPU architectures from NVidia, after processing widely used benchmark hyperspectral scenes collected by AVIRIS. The proposed implementation can be combined as a preprocessing module to other GPU implementations of spectral unmixing chains, thus achieving a full operational procedure that provides an end-to-end approach to process hyperspectral data in computationally efficient fashion. Future work will be focused on the development of other implementations of the full spectral unmixing chain (including SPP) on alternative hardware devices such as field programmable gate arrays (FPGAs).

Acknowledgment

The authors gratefully acknowledge the associate editor and the anonymous reviewers for their outstanding comments and suggestions.

References

[1] J. M. Bioucas-Dias *et al.*, “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.

[2] J. Plaza, A. Plaza, R. Perez, and P. Martinez, “On the use of small training sets for neural network-based characterization of mixed pixels in remotely sensed hyperspectral images,” *Pattern Recognit.*, vol. 42, pp. 3032–3045, 2009.

[3] A. Plaza, P. Martinez, R. Perez, and J. Plaza, “Spatial/spectral endmember extraction by multidimensional morphological operations,” *IEEE Trans. Geosci. Remote Sens.*, vol. 40, no. 9, pp. 2025–2041, Sep. 2002.

[4] D. M. Rogge, B. Rivard, J. Zhang, A. Sanchez, J. Harris, and J. Feng, “Integration of spatial–spectral information for the improved extraction of endmembers,” *Remote Sens. Environ.*, vol. 110, no. 3, pp. 287–303, 2007.

[5] S. Lopez, J. F. Moure, A. Plaza, G. M. Callico, J. F. Lopez, and R. Sarmiento, “A new preprocessing technique for fast hyperspectral endmember extraction,” *IEEE Geosci. Remote Sens. Lett.*, vol. 10, no. 5, pp. 1070–1074, Sep. 2013.

[6] M. Zortea and A. Plaza, “Spatial preprocessing for endmember extraction,” *IEEE Trans. Geosci. Remote Sens.*, vol. 47, no. 8, pp. 2679–2693, Aug. 2009.

[7] G. Martin and A. Plaza, “Region-based spatial preprocessing for endmember extraction and spectral unmixing,” *IEEE Geosci. Remote Sens. Lett.*, vol. 8, no. 4, pp. 745–749, Jul. 2011.

[8] G. Martin and A. Plaza, “Spatial-spectral preprocessing prior to endmember identification and unmixing of remotely sensed hyperspectral data,” *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 5, no. 2, pp. 380–395, Apr. 2012.

[9] J. Plaza, E. M. T. Hendrix, I. Garcia, G. Martin, and A. Plaza, “On endmember identification in hyperspectral images without pure pixels: A comparison of algorithms,” *J. Math. Imag. Vis.*, vol. 42, no. 2–3, pp. 163–175, 2012.

[10] G. M. Gallicó, S. Lopez, B. Aguillar, J. F. López, and R. Sarmiento, “Parallel implementation of the modified vertex component analysis algorithm for hyperspectral parallel implementation of the modified vertex component analysis algorithm for hyperspectral unmixing using opencl,” *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 7, no. 8, pp. 3650–3659, Aug. 2014.

[11] S. Sanchez, A. Paz, G. Martin, 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, 2011.

[12] S. Sánchez, R. Ramalho, L. Sousa, and A. Plaza, “Real-time implementation of remotely sensed hyperspectral image unmixing on GPUs,” *J. Real-Time Image Process.*, vol. 10, no. 3, pp. 469–483, 2015.

[13] J. M. P. Nascimento, J. M. Bioucas-Dias, J. M. Rodriguez Alves, V. Silva, and A. Plaza, “Parallel hyperspectral unmixing on GPUs,” *IEEE Geosci. Remote Sens. Lett.*, vol. 11, no. 3, pp. 666–670, Mar. 2013.

[14] M. Harris *et al.*, "Optimizing parallel reduction in CUDA," NVIDIA Corp., Santa Clara, CA, USA, Tech. Rep., vol. 2, no. 4, 2007.

[15] N. Whitehead and A. Fit-Florea, "NVIDIA's white paper of precision and performance: Floating point and IEEE 754 compliance for NVIDIA GPUs," NVIDIA corp., Santa Clara, CA, USA, Technical White Paper, Tech. Rep., 2011.

[16] M. J. Corden and D. Kreitzer, "Consistency of floating-point results using the intel compiler or why doesn't my application always give the same answer," Intel Corp., Software Solutions Group, Santa Clara, CA, USA, Tech. Rep., 2009.

[17] G. Motta, F. Rizzo, and J. A. Storer, *Hyperspectral Data Compression*. New York, NY, USA: Springer, 2006.

[18] C.-I. Chang and Q. Du, "Estimation of number of spectrally distinct signal sources in hyperspectral imagery," *IEEE Trans. Geosci. Remote Sens.*, vol. 42, no. 3, pp. 608–619, Mar. 2004.

[19] J. C. Harsanyi and C.-I. Chang, "Hyperspectral image classification and dimensionality reduction: An orthogonal subspace projection approach," *IEEE Trans. Geosci. Remote Sens.*, vol. 32, no. 4, pp. 779–785, Jul. 1994.

[20] *Cuda C Programming Guide*, pg-02829-001 v7.0 ed., NVIDIA Corp., Santa Clara, CA, USA, Mar. 2015.

[21] R. O. Green *et al.*, "Imaging spectroscopy and the airborne visible/infrared imaging spectrometer (AVIRIS)," *Remote Sens. Environ.*, vol. 65, pp. 227–248, 1998.

Jaime Delgado was born in Badajoz, Spain, in 1985. He received the M.Sc. degree in computer engineering from the University of Extremadura, Caceres, Spain, in 2014. He was an Exchange Student at the Universidade de Aveiro, Aveiro, Portugal, and also at Otto-Friedrich Universität, Bamberg, Germany.

He has developed his career as System Engineer specialized in SAP Complex Systems based on Linux OS and Oracle databases. He has been part of System Engineering Team in companies like the Birchman Group or SAP Basis Administrator with Indra. Currently, he is working as a SAP Systems Administrator Chief with Imagina Group, Barcelona, Spain. His research interests include high-level projects such as system copies, upgrades, or system performance improvements.

Gabriel Martín received the degree in computer engineering, in 2008, the M.Sc. and Ph.D. degrees from the University of Extremadura, Cáceres, Spain, in 2010 and 2013, respectively.

He was a Predoctoral Research Associate (funded by the Spanish Ministry of Science and Innovation) with the Hyperspectral Computing Laboratory and is now a Postdoctoral Researcher (funded by FCT) with the Instituto Superior Técnico, Technical University of Lisbon, Lisbon, Portugal, where he is developing research on compressive sensing and efficient hardware implementations for remotely sensed hyperspectral images. His research interests include the development of new techniques for unmixing remotely sensed hyperspectral data sets, as well as the efficient processing and interpretation of these data in different types of high-performance computing architectures.

Dr. Martín served as a Reviewer for the IEEE JOURNAL OF SELECTED TOPICS IN APPLIED EARTH OBSERVATIONS AND REMOTE SENSING and for the IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING.

Javier Plaza (M'09) was born in Cáceres, Spain, in 1979. He received the M.Sc. and Ph.D. degrees in computer engineering from the University of Extremadura, Cáceres, Spain, in 2004 and 2008, respectively.

Currently, he is an Associate Professor with the Department of Technology of Computers and Communications, University of Extremadura. He has authored or coauthored more than 100 scientific publications, including more than 30 journal papers, 10 book chapters, and over 60 peer-reviewed conference proceeding papers. His research interests include remotely sensed hyperspectral imaging, pattern recognition, signal and image processing, neural networks, and efficient implementation of large-scale scientific problems on parallel and distributed computer architectures.

Dr. Plaza has served as a Reviewer for more than 150 papers submitted to more than 25 different journals, including the IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING, IEEE JOURNAL OF SELECTED TOPICS IN APPLIED EARTH OBSERVATION AND REMOTE SENSING, IEEE GEOSCIENCE AND REMOTE SENSING LETTERS, IEEE GEOSCIENCE AND REMOTE SENSING MAGAZINE, IEEE TRANSACTIONS ON IMAGE PROCESSING, IEEE TRANSACTIONS ON SIGNAL PROCESSING, IEEE JOURNAL OF SELECTED TOPICS IN SIGNAL PROCESSING, and IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS C. He was the recipient of the Outstanding Ph.D. Dissertation Award from the University of Extremadura in 2008.

Luis Ignacio Jiménez (S'15) received the B.S. and M.Sc. degrees in computer engineering in 2011 and 2012, respectively, and is currently a Ph.D. student with the Hyperspectral Computing Laboratory, Department of Technology of Computers and Communications, University of Extremadura, Cáceres, Spain.

His research interests include hyperspectral image analysis, research software development, and efficient implementations of large-scale scientific problems on commodity graphical processing units (GPUs). He was working with the Computer Vision Laboratory (LVC), Department of Electrical Engineering, Catholic University of Rio de Janeiro, Rio de Janeiro, RJ, Brazil, under the European project "Tools for Open Multi-Risk Assessment Using Earth Observation Data" (TOLOMEO).

Antonio Plaza (M'05–SM'07–F'15) was born in Caceres, Spain, in 1975.

He is an Associate Professor (with accreditation for Full Professor) with the Department of Technology of Computers and Communications, University of Extremadura, Cáceres, Spain, where he is the Head of the Hyperspectral Computing Laboratory (HyperComp). He has been the Advisor of 12 Ph.D. dissertations and more than 30 M.Sc. dissertations. He was the Coordinator of the Hyperspectral Imaging Network, a European project with total funding of 2.8 million Euro. He has authored more than 500 publications, including 159 journal papers (more than 100 in IEEE journals), 22 book chapters, and over 240 peer-reviewed conference proceeding papers (94 in IEEE conferences). He has reviewed more than 500 manuscripts for over 50 different journals. He has edited a book on *High-Performance Computing in Remote Sensing* (CRC Press/Taylor and Francis) and guest edited nine special issues on hyperspectral remote sensing for different journals. His research interests include hyperspectral data processing and parallel computing of remote sensing data.

Dr. Plaza was also a member of the steering committee of the IEEE JOURNAL OF SELECTED TOPICS IN APPLIED EARTH OBSERVATIONS AND REMOTE SENSING (JSTARS). He is also an Associate Editor for IEEE ACCESS, and was a member of the Editorial Board of the IEEE Geoscience and Remote Sensing Newsletter (2011–2012) and the IEEE Geoscience and Remote Sensing Magazine (2013). He has served as a Proposal Evaluator for the European Commission, the National Science Foundation, the European Space Agency, the Belgium Science Policy, the Israel Science Foundation, and the Spanish Ministry of Science and Innovation. He is currently serving as the Editor-in-Chief of the IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING Journal. He is a recipient of the recognition of Best Reviewers of the IEEE GEOSCIENCE AND REMOTE SENSING LETTERS (in 2009) and a recipient of the recognition of Best Reviewers of the IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING (in 2010), a journal for which he served as Associate Editor in 2007–2012. He is a recipient of the 2013 Best Paper Award of the JSTARS journal, and a recipient of the most highly cited paper (2005–2010) in the Journal of Parallel and Distributed Computing. He received the Best Paper Awards at the IEEE International Conference on Space Technology and the IEEE Symposium on Signal Processing and Information Technology. He is a recipient of the Best Ph.D. Dissertation Award at the University of Extremadura. He served as the Director of Education Activities for the IEEE Geoscience and Remote Sensing Society (GRSS) in 2011–2012, and is currently serving as a President of the Spanish Chapter of IEEE GRSS (since November 2012).