Multi-GPU Implementation of the Minimum Volume Simplex Analysis Algorithm for Hyperspectral Unmixing

Alexander Agathos, Jun Li, Dana Petcu, and Antonio Plaza, *Senior Member, IEEE*

***Abstract*—Spectral unmixing is an important task in remotely sensed hyperspectral data exploitation. The linear mixture model has been widely used to unmix hyperspectral images by identifying a set of pure spectral signatures, called endmembers, and estimating their respective abundances in each pixel of the scene. Several algorithms have been proposed in the recent literature to automatically identify endmembers, even if the original hyperspectral scene does not contain any pure signatures. A popular strategy for endmember identification in highly mixed hyperspectral scenes has been the minimum volume simplex analysis (MVSA), known to be a computationally very expensive algorithm. This algorithm calculates the minimum volume enclosing simplex, as opposed to other algorithms that perform maximum simplex volume analysis (MSVA). The high computational complexity of MVSA, together with its very high memory requirements, has limited its adoption in the hyperspectral imaging community. In this paper, we develop several optimizations to the MVSA algorithm. The main computational task of MVSA is the solution of a quadratic optimization problem with equality and inequality constraints, with the inequality constraints being in the order of the number of pixels multiplied by the number of endmembers. As a result, storing and computing the inequality constraint matrix is highly inefficient. The first optimization presented in this paper uses algebra operations in order to reduce the memory requirements of the algorithm. In the second optimization, we use graphics processing units (GPUs) to effectively solve (in parallel) the quadratic optimization problem involved in the computation of MVSA. In the third optimization, we extend the single GPU implementation to a multi-GPU one, developing a hybrid strategy that distributes the computation while taking advantage of GPU accelerators at each node. The presented optimizations are tested in different analysis scenarios (using both synthetic and real hyperspectral data) and shown to provide state-of-the-art results from the viewpoint of unmixing accuracy and computational performance. The speedup achieved using the full GPU cluster compared to the CPU implementation in tenfold in a real hyperspectral image.**

***Index Terms*—Endmember identification, graphics processing units (GPUs), hyperspectral imaging, minimum volume simplex analysis (MVSA), spectral unmixing.**

Manuscript received November 19, 2013; revised March 15, 2014; accepted April 22, 2014. Date of publication May 21, 2014; date of current version August 01, 2014. The work of A. Agathos and D. Petcu was supported in part by the European Commission FP7 programme under Project FP7-REGPOT-CT-2011-284595-HOST. *(Corresponding author: Jun Li.)*

A. Agathos and D. Petcu are with the Computer Science Department, West University of Timisoara, Timisoara 300223, Romania.

J. Li is with the School of Geography and Planning, Sun Yat-sen University, Guangzhou 510275, China (e-mail: lijun48@mail.sysu.edu.cn).

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

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.2014.2320896

I. Introduction

HYPERSPECTRAL imaging instruments acquire hundreds of images, at nearly contiguous wavelengths, for the same area on the surface of the Earth [1]. This has led to significant improvements in many applications, such as agriculture, geography, geology, mineral identification, detection, and classification of man-made targets [2]–[4]. The analysis and interpretation of remotely sensed hyperspectral scenes has become a very active research topic in recent years [5].

Spectral unmixing is a very important technique for hyperspectral image exploitation [6]. It decomposes the (possibly mixed) pixel spectra into a collection of constituent spectra, or spectral signatures, and their corresponding fractional abundances that quantify the proportion of each pure material (also termed *endmember*) in the pixel. Depending on the mixing proportion at each pixel and on the geometry of the data set, the observed mixture can be either linear or nonlinear [4]. In linear mixing, the acquired spectral vectors are a linear combination of the endmember signatures present in the scene, weighted by the respective fractional abundances. In nonlinear mixing, the radiative transfer theory produces a model for the transfer of energy as photons interact with the materials in the scene [7]. In the literature, mostly the linear mixing model has been considered. The reason is that, despite its simplicity, it is an acceptable approximation of the light scattering mechanisms in many real scenarios. Furthermore, the linear mixing model constitutes the basis of many effective unmixing algorithms. This is to be contrasted with the nonlinear mixing model, where the inference of the spectral signatures and of material densities based on radiative transfer theory is a complex ill-posed problem, relying on scene parameters which are often very hard to obtain. There are particular situations in which a nonlinear model can be transformed to a linear (or bilinear) one [6].

In this paper, we focus on the linear model and particularly address a class of endmember identification algorithms that do not assume the presence of pure pixels in the scene, refer to [6] for a description of these algorithms. These algorithms generally rely on minimum volume concepts to identify the endmember signatures, without the need for such signatures to be present in the image data. The concept used by these algorithms is opposite to the one adopted by algorithms performing maximum simplex volume analysis (MSVA) [8], in which the goal is to inflate a simplex with maximum volume using real pixels in the data, and the endmembers are found as the real image pixels that define the simplex with maximum volume. Quite opposite, the concept

adopted by the algorithms in the minimum volume category is quite different, as they use a minimum volume enclosing strategy (meaning that the endmembers are found as the vertices of the simplex with minimum volume that encloses all data observations). A popular algorithm in this category is the minimum volume simplex analysis (MVSA) [9], which is characterized by its high computational complexity. Until now, there has been no effort to accelerate the MVSA algorithm using parallel techniques. However, as we will show in this work, this algorithm is quite suitable for parallel optimizations. Specifically, we will use the interior point method to accelerate the quadratic optimization, which is the most computationally demanding task of MVSA. Such iterative methods, like the interior point, have shown to be well suited for parallel optimizations [10], [11]. Moreover, we will optimize the interior-point algorithm for the specific problem of MVSA, drastically reducing the memory needed to solve the quadratic problem and also the computations, by avoiding the use of sparse matrices which produce memory bandwidth bounded operations. In this work, we also further explore the possibility of accelerating MVSA using multiple graphics processing units (GPUs), a low cost but massively parallel platform that has experienced great success in the implementation of hyperspectral imaging algorithms (see [12], and references therein). As a result, GPUs have become a relevant and inexpensive computing platform in order to accelerate hyperspectral-related computations. For instance, in [13], the role of GPUs in remote sensing missions is discussed, together with other high-performance computing architectures such as clusters, heterogeneous networks of workstations, or field programmable gate arrays (FPGAs). In [14], a similar study is conducted specifically for hyperspectral missions. In [15], a general overview of the most recent developments in high-performance computing applied to Earth observation applications is given. In [16], several advances toward real-time processing of remote sensing data are discussed. Finally, [17] gives a more general perspective on the exploitation of high-performance computing platforms in remote sensing studies, with a more extensive discussion on specific case studies.

As mentioned above, the major computational task of MVSA is the solution of a quadratic problem with linear equality and inequality constraints. The inequality constraints are in the order of $N \times p$, where N is the number of pixels in the hyperspectral image and p is the number of endmembers, making the problem very hard to solve and store in memory. In this work, we solve the quadratic problem using the interior point method [18] and further present several optimizations. The first optimization presented in this paper uses algebra operations in order to reduce the memory requirements of the algorithm. In the second optimization, we use GPUs to effectively solve (in parallel) the quadratic optimization problem involved in the computation of MVSA. In the third optimization, a multi-GPU implementation for distributed environments is presented. In our experiments, we report almost linear speedup and show that the proposed implementation is sublinearly scalable across multiple GPUs. The parallel GPU implementation uses existing libraries provided by the compute device unified architecture (CUDA)[1] toolkit, including a linear algebra library for CUDA (cuBLAS)[2] and Thrust.[3] The additional GPU kernels that need to be constructed are simple and also the memory consumption needed is well within the range of commodity GPUs.

The remainder of the paper is organized as follows. Section II presents related work in spectral unmixing, highlighting previous GPU implementations of unmixing algorithms available in the literature. Section III describes the MVSA algorithm. Section IV describes the parallel models used for the efficient implementations of MVSA carried out in this work. Section V presents several optimization of MVSA that constitute the main contribution of this work. Section VI describes the experimental results performed, using both synthetic and real hyperspectral data sets and a variety of computing platforms. Finally, Section VII concludes with some remarks and hints at plausible future research.

II. Related Work

This section provides an overview of existing developments in spectral unmixing algorithms with particular focus on available GPU implementations. The paper [6] categorizes existing unmixing algorithms based on the linear mixture model in three main categories.

1) In *geometrical methods*, techniques like the MSVA [8] are used to determine the endmembers of the image. A representative algorithm of this category is the N-FINDR [19], which is based on the fact that the volume defined by a simplex formed by the purest pixels of the hyperspectral data is larger than any other volume defined by any other combination of pixels. This algorithm finds the set of pixels defining the largest volume by inflating a simplex inside the data. N-FINDR has been recently parallelized for GPUs in [20]. Most importantly, a GPU implementation of the general MSVA framework has been given in [21]. Another relevant example is the vertex component analysis (VCA) [22], which iteratively projects data onto a direction orthogonal to the subspace spanned by the endmembers already determined. The new endmember signature corresponds to the extreme of the projection. The algorithm iterates until all endmembers are exhausted. VCA has been recently parallelized using GPUs in [23] and [24].
2) Another category of algorithms is given by *statistical methods*, in which Bayesian theory is used to find statistical estimators of the endmembers and the corresponding abundances [25]. Another family of statistical methods is composed by spatial–spectral approaches, which combine the spatial and the spectral information in the endmember identification process. An example of this kind of algorithms is the automated morphological endmember extraction (AMEE) algorithm [26], which has been implemented in GPUs in [27].
3) A third category of algorithms is given by sparse *regression methods*, in which the spectral endmembers used to

[1]Available: http://www.nvidia.com/object/cuda_home_new.html

[2]Available: https://developer.nvidia.com/cuBLAS

[3]Available: https://code.google.com/p/thrust/

solve the mixture problem are no longer extracted from the original hyperspectral data, but instead selected from a library containing a large number of spectral samples available a priori [28]. In this case, unmixing is transformed into the problem of finding the optimal subset of samples in the library that can best model each mixed pixel in the scene. A GPU implementation of sparse unmixing methods has been recently presented in [29].

It should be noted that the category of geometrical algorithms has been by far the most widely explored in the hyperspectral unmixing literature. In addition to algorithms assuming the presence of pure pixels in the data such as N-FINDR or VCA, another popular family of algorithms is based on the assumption that spectrally pure signatures may not be present in the data due to spatial resolution and other phenomena [6]. As opposed to N-FINDR, which follows a volume maximization strategy, these algorithms minimize the volume of the simplex that encloses all the pixel observations in the hyperspectral data. Some well-known algorithms in this category include MVSA [9], which uses sequential quadratic programming in order to minimize the volume of the simplex. The simplex identification via variable splitting and augmented lagrangian (SISAL), which uses the augmented lagrangian minimization method. The minimum volume constrained nonnegative matrix factorization (MVC-NMF) [30], which uses the nonegative matrix factorization algorithm to minimize the volume of the simplex. The minimum volume enclosing simplex (MVES) [31], which finds a simplex by minimizing the volume subject to the constraint that all the dimension-reduced pixels are enclosed by the simplex. We emphasize that, to the best of our knowledge, none of the geometrical algorithms without the pure pixel assumption have been implemented in GPUs in the past. In this work, we take a necessary first step in this direction and propose a multi-GPU implementation of MVSA, a popular algorithm in this category.

III. Minimum Volume Simplex Analysis

In this section, we describe the MVSA algorithm. For its initial definition, please refer to [9]. For the sake of the comprehensiveness, we reproduce it here and interpret it according to our needs for the parallel implementation.

We assume that the hyperspectral image is given by a $L \times N$ matrix $\mathbf{Y} \equiv [\mathbf{y}_1, \ldots, \mathbf{y}_N]$, where N is the number of pixels and L the number of spectral bands. The linear mixture model assumes $\mathbf{Y} = \mathbf{MA}$, where the $L \times p$ matrix $\mathbf{M} \equiv [\mathbf{m}_1, \ldots, \mathbf{m}_p]$ is the endmember matrix and the $p \times N$ matrix $\mathbf{A} \equiv [\mathbf{a}_1, \ldots, \mathbf{a}_N]$ is the abundance matrix, with p being the number of endmembers. The elements of $\mathbf{A}$ belong in the p-dimensional simplex $\mathcal{A}_p = \{\mathbf{a} \in \mathbb{R}^p : \mathbf{a} \succeq \mathbf{0}, \mathbf{1}_p^T \mathbf{a} = 1\}$. If the vectors of $\mathbf{M}$ are linearly independent, then the vectors of $\mathbf{Y}$ belong in a $p-1$-dimensional simplex as Fig. 1 suggests.

The number of endmembers present in a given scene is, very often, much smaller than the number of bands L. Therefore, it is common that spectral vectors lie in a lower dimensional linear subspace. In MVSA, the subspace

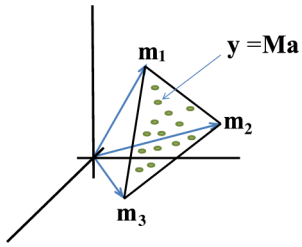

Fig. 1. Hyperspectral pixel vectors belong in the simplex formed by the endmembers: $\mathbf{m}_1$, $\mathbf{m}_2$, $\mathbf{m}_3$.

dimension is assumed to be p, and singular value decomposition (SVD) [32] provides the projection that best represents the data in the maximum power sense. As a result, it is assumed that the number of endmembers and the signal subspace are known beforehand, and that the observed vectors $\mathbf{y}_i$, $i = 1 \ldots N$, represent coordinates with respect to a p-dimensional basis of the signal subspace, where the basis is acquired by the first p eigenvectors of the empirical correlation matrix [32], i.e., $L = p$ and $\mathbf{Y} = \mathbf{MA}$, where $\mathbf{Y} \in \mathbb{R}^{p \times N}$, $\mathbf{M} \in \mathbb{R}^{p \times p}$, and $\mathbf{A} \in \mathbb{R}^{p \times N}$. MVSA finds the minimum volume simplex that encloses $\mathbf{Y}$ by the following optimization process:

$$\begin{aligned} &\widehat{\mathbf{M}} = \arg\min_{\mathbf{M}} |\det(\mathbf{M})| \\ &s.t.: \quad \mathbf{QY} \geq 0, \quad \mathbf{1}_p^T \mathbf{Q} = \mathbf{1}_N^T \mathbf{Y}^T (\mathbf{YY}^T)^{-1} \end{aligned} \tag{1}$$

where $\mathbf{Q} \equiv \mathbf{M}^{-1}$. This problem is in fact equivalent to

$$\begin{aligned} &\widehat{\mathbf{Q}} = -\arg\min_{\mathbf{Q}} \log(|\det(\mathbf{Q})|) \\ &s.t.: \quad \mathbf{A}_I \mathbf{q} \geq \mathbf{b}_I, \quad \mathbf{A}_E \mathbf{q} = \mathbf{b}_E \end{aligned} \tag{2}$$

where $\mathbf{A}_I = \mathbf{Y}^T \otimes \mathbf{I}_p$, $\mathbf{A}_\mathbf{E} = \mathbf{I}_p \otimes \mathbf{1}_p$, $\mathbf{b}_I = \mathbf{0}$, and $\mathbf{b}_E = (\mathbf{YY}^T)^{-1} \mathbf{Y} \mathbf{1}_N$. Here, $\mathbf{I}_p$ is the identity matrix of dimension p, $\mathbf{1}_N$ is the p-dimensional unit vector, $\otimes$ is the Kronecker product, and $\mathbf{q}$ is the vector produced by $\mathbf{Q}$ in a column-major order. So the problem consists of a minimization of a nonlinear function under linear equality and inequality constraints.

The optimization problem (2) is a nonconvex one, and the nonlinear function can be approximated by its second order Taylor approximation with a known feasible solution $\mathbf{q}_0$. If we assume $f(\mathbf{q}) = -\log(|\det(\mathbf{Q})|)$, then taking its second order Taylor approximation with a feasible solution $\mathbf{q}_0$, the following approximation holds:

$$f(\mathbf{q}) = \mathbf{c}_0 + \mathbf{c}^T \mathbf{q} + \frac{1}{2} \mathbf{q}^T \mathbf{G} \mathbf{q} \tag{3}$$

where $\mathbf{G} = \nabla^2 f(\mathbf{q}_0)$ and $\mathbf{c} = \nabla f(\mathbf{q}_0) - \nabla^2 f(\mathbf{q}_0) \mathbf{q}_0$. Thus, the function is transformed into a quadratic function.

MVSA uses the work in [33] and transforms the minimization problem (2) into a sequence of quadratic minimizations with

linear equality and inequality constraints. Each quadratic problem has the form

$$\widehat{\mathbf{q}} = \arg\min_{\mathbf{q}} \mathbf{c}^T\mathbf{q} + \frac{1}{2}\mathbf{q}^T\mathbf{G}\mathbf{q}$$
$$s.t.: \quad \mathbf{A}_I\mathbf{q} \geq \mathbf{b}_I, \quad \mathbf{A}_E\mathbf{q} = \mathbf{b}_E. \tag{4}$$

Algorithm 1: Pseudocode of sequential quadratic programming

1: **INPUT:** $\mathbf{A}_I$, $\mathbf{A}_E$, $\mathbf{b}_I$, $\mathbf{b}_E$, $\mathbf{q}_0$
2: $Convergence \leftarrow false$
3: **repeat**
4: Compute $\nabla^2 f(\mathbf{q}_0)$, $\nabla f(\mathbf{q}_0)$
5: $q \leftarrow$ solution of the quadratic optimization (4)
6: **if** $f(\mathbf{q}_0) < f(\mathbf{q})$ **then**
7: do line search until $f(\mathbf{q}_0) > f(\mathbf{q})$
8: **end if**
9: **if** $|f(\mathbf{q}_0) < f(\mathbf{q})| < threshold$ **then**
10: $Convergence \leftarrow true$
11: **end if**
12: $\mathbf{q}_0 \leftarrow \mathbf{q}$
13: **until** $Convergence$

The pseudocode of this sequence of quadratic programming is shown in Algorithm 1, which is terminated in a finite number of steps if convergence is not reached. The most demanding process is to solve the nonlinear quadratic problem with linear equality and inequality constraints (4). In the literature, there is a plethora of algorithms that can solve this specific problem, see [18] for a detailed description of these algorithms. In this work, we have chosen to use the interior point algorithm for two reasons.

1) First and foremost, the interior point algorithm uses the Newton step for the approximation of the solution of the nonlinear system of equations, thus converge quadratically to the solution.
2) Second, we will show that our efficient implementation of the interior point algorithm has much lower memory consumption than a naive implementation of the interior point which uses directly the constraint matrix $\mathbf{A_I}$. $\mathbf{A_I}$ becomes very large and sparse on real hyperspectral images. In our implementation, we will show how to avoid the use of $\mathbf{A_I}$ and utilize instead nonsparse matrices of much lower size. This makes the problem suitable for basic linear algebra subroutine (BLAS) operations on GPUs, which do not require very large memory and produce memory bandwidth limited operations on sparse matrices. Also, we will utilize the classical approach to relax the linear system produced by the Jacobian matrix using the interior-point algorithm.

The system of nonlinear equations that need to be solved satisfies the first order Karush, Kuhn, Tucker (KKT) conditions [18] and can be expressed by the following system:

$$\begin{aligned} \mathbf{G}\mathbf{x} - \mathbf{A}_I^T\boldsymbol{\lambda} + \mathbf{A}_E^T\boldsymbol{\mu} + \mathbf{c} &= 0 \\ \mathbf{A}_I\mathbf{x} - \mathbf{s} - \mathbf{b}_I &= 0 \\ \mathbf{A}_E\mathbf{x} - \mathbf{b}_E &= 0 \\ \mathbf{s}_i\boldsymbol{\lambda}_i = 0, \quad i &= 1 \ldots n_I \\ \boldsymbol{\lambda}, \mathbf{s} &\geq 0 \end{aligned} \tag{5}$$

where $\mathbf{s}$ is a positive slack variable and $\boldsymbol{\lambda}$ and $\boldsymbol{\mu}$ the lagrangian multipliers for the inequality and equality constraints, respectively. This system of nonlinear equations can be solved by an interior point algorithm with $\mathbf{s}$ and $\boldsymbol{\lambda}$ being the primal and dual variables, respectively.

From the plethora of interior point algorithms available [18], we use the predictor–corrector interior point algorithm. The predictor–corrector is an iterative algorithm in which, at each iteration, an initial Newton step is calculated and called affine step, and then this step is corrected using two scalar parameters, ρ and σ, calculating the final Newton step. In our context, both of these scalar parameters should converge to zero, so as to achieve convergence to the correct solution of the quadratic problem. These parameters have a significant meaning in our context, i.e., the convergence of ρ to zero is essential to achieve strict complementarity in the KKT conditions (see [34]), while the convergence of σ to zero is also essential, since the Newton method converges rapidly near the exact solution. Once convergence is achieved, the solution is provably the correct solution since the KKT conditions are satisfied. The affine Newton step is given by the solution of the following linear system:

$$\begin{bmatrix} \mathbf{G} & \mathbf{A}_E^T & \mathbf{0} & -\mathbf{A}_I^T \\ \mathbf{A}_I & \mathbf{0} & -\mathbf{I} & \mathbf{0} \\ \mathbf{A}_E & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{\Lambda} & \mathbf{S} \end{bmatrix} \begin{bmatrix} \boldsymbol{\Delta}\mathbf{x}^{aff} \\ \boldsymbol{\Delta}\boldsymbol{\mu}^{aff} \\ \boldsymbol{\Delta}\mathbf{s}^{aff} \\ \boldsymbol{\Delta}\boldsymbol{\lambda}^{aff} \end{bmatrix} = \begin{bmatrix} -\mathbf{r_d} \\ -\mathbf{r_I} \\ -\mathbf{r_E} \\ -\boldsymbol{\Lambda}\mathbf{S}\mathbf{e} \end{bmatrix} \tag{6}$$

where

$$\begin{aligned} \mathbf{r_d} &= \mathbf{G}\mathbf{x} - \mathbf{A}_I^T\boldsymbol{\lambda} + \mathbf{A}_E^T\boldsymbol{\mu} + \mathbf{c} \\ \mathbf{r_I} &= \mathbf{A}_I\mathbf{x} - \mathbf{s} - \mathbf{b}_I \\ \mathbf{r_E} &= \mathbf{A}_E\mathbf{x} - \mathbf{b}_E \\ \boldsymbol{\Lambda} &= \mathrm{diag}(\boldsymbol{\lambda}_1, \ldots, \boldsymbol{\lambda}_{n_I}), \quad \mathbf{S} = \mathrm{diag}(s_1, \ldots, s_{n_I}), \\ \mathbf{e} &= [1, \ldots, 1]^T. \end{aligned}$$

Using the affine step, now we calculate the following linear system:

$$\begin{bmatrix} \mathbf{G} & \mathbf{A}_E^T & \mathbf{0} & -\mathbf{A}_I^T \\ \mathbf{A}_I & \mathbf{0} & -\mathbf{I} & \mathbf{0} \\ \mathbf{A}_E & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{\Lambda} & \mathbf{S} \end{bmatrix} \begin{bmatrix} \boldsymbol{\Delta}\mathbf{x} \\ \boldsymbol{\Delta}\boldsymbol{\mu} \\ \boldsymbol{\Delta}\mathbf{s} \\ \boldsymbol{\Delta}\boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} -\mathbf{r_d} \\ -\mathbf{r_I} \\ -\mathbf{r_E} \\ -\boldsymbol{\Delta}\mathbf{S}\mathbf{e} - \mathbf{c} \end{bmatrix} \tag{7}$$

where $\mathbf{c} = \boldsymbol{\Delta}\boldsymbol{\Lambda}^{aff}\boldsymbol{\Delta}\mathbf{S}^{aff}\mathbf{e} + \sigma\rho\mathbf{e}$ is the correction in the prediction with $\boldsymbol{\Delta}\boldsymbol{\Lambda}^{aff} = \mathrm{diag}(\Delta\boldsymbol{\lambda}_1^{aff}, \ldots, \Delta\boldsymbol{\lambda}_{n_I}^{aff})$, $\boldsymbol{\Delta}\mathbf{S}^{aff} = \mathrm{diag}(\Delta s_1^{aff}, \ldots, \Delta s_{n_I}^{aff})$, $\rho = \frac{\mathbf{s}^T\boldsymbol{\lambda}}{n_I}$, and $\sigma \in (0,1]$. The predictor–corrector interior point algorithm for the solution of the quadratic problem is shown in Algorithm 2.

Algorithm 2: The predictor corrector interior point algorithm

1: Initialize $(\mathbf{x}_0, \boldsymbol{\mu}_0, \mathbf{s}_0, \boldsymbol{\lambda}_0)$ with $\mathbf{s}_0,\ \boldsymbol{\lambda}_0 > \mathbf{0}$

2: $k \leftarrow 0$

3: **while** σ, ρ not small enough **do**

4: $(\mathbf{x}, \boldsymbol{\mu}, \mathbf{s}, \boldsymbol{\lambda}) \leftarrow (\mathbf{x}_k, \boldsymbol{\mu}_k, \mathbf{s}_k, \boldsymbol{\lambda}_k)$

5: solve (6) and get $(\Delta\mathbf{x}^{aff}, \Delta\boldsymbol{\mu}^{aff}, \Delta\mathbf{s}^{aff}, \Delta\boldsymbol{\lambda}^{aff})$

6: $\rho \leftarrow \frac{\mathbf{s}^T\boldsymbol{\lambda}}{n_I}$

7: $\hat{\alpha}_{aff} \leftarrow \max\{\alpha \in (0,1] | (\mathbf{s}, \boldsymbol{\lambda}) + \alpha(\Delta\mathbf{s}^{aff}, \Delta\boldsymbol{\lambda}^{aff}) \geq 0\}$

8: $\rho_{aff} \leftarrow (\mathbf{s} + \hat{\alpha}_{aff}\Delta\mathbf{s}^{aff})^T(\boldsymbol{\lambda} + \hat{\alpha}_{aff}\Delta\boldsymbol{\lambda}^{aff})/n_I$

9: $\sigma \leftarrow \left(\frac{\rho_{aff}}{\rho}\right)^3$

10: solve (7) and get $(\Delta\mathbf{x}, \Delta\boldsymbol{\mu}, \Delta\mathbf{s}, \Delta\boldsymbol{\lambda})$

11: $\tau_k \leftarrow 1 - \frac{1}{k+1}$

12: $\hat{\alpha} \leftarrow \max\{\alpha \in (0,1] | (\mathbf{s}, \boldsymbol{\lambda}) + \alpha(\Delta\mathbf{s}, \Delta\boldsymbol{\lambda}) \geq (1-\tau_k)(\mathbf{s}, \boldsymbol{\lambda})\}$

13: $(\mathbf{x}_{k+1}, \boldsymbol{\mu}_{k+1}, \mathbf{s}_{k+1}, \boldsymbol{\lambda}_{k+1}) \leftarrow (\mathbf{x}_k, \boldsymbol{\mu}_k, \mathbf{s}_k, \boldsymbol{\lambda}_k) + \hat{\alpha}(\Delta\mathbf{x}, \Delta\boldsymbol{\mu}, \Delta\mathbf{s}, \Delta\boldsymbol{\lambda})$

14: $k \leftarrow k + 1$

15: **end while**

16: **return** $\mathbf{x}_k$

In the case of MVSA, the number of unknowns is p^2, the number of inequality constraints is $n_I = Np$ and the number of equality constraints is $n_E = p$. As it can be seen from Algorithm 2, the main computational task in the predictor–corrector interior point algorithm is the calculation of the Newton steps, which involve the solution of the linear systems (6) and (7). The size of the Jacobian matrices of these two systems is $(2Np + p^2 + p) \times (2Np + p^2 + p)$. This means that for a standard hyperspectral image with common size of $N = 350 \times 350$ pixels and $p = 19$ endmembers, the size of the linear systems (6) and (7) is prohibitively large to store and compute. However, by exploiting the structure of the Jacobian matrices, the two linear systems can be solved progressively by deriving the "normal equations" [18] as follows:

$$\begin{aligned} \left(\mathbf{G} + \mathbf{A}_I^T\mathbf{S}^{-1}\boldsymbol{\Lambda}\mathbf{A}_I\right)\Delta\mathbf{x} + \mathbf{A}_E^T\Delta\boldsymbol{\mu} &= \mathbf{r}_a \\ \mathbf{A}_E\Delta\mathbf{x} &= \mathbf{r}_b \\ \Delta\mathbf{s} &= \mathbf{r}_{\Delta\mathbf{x}} \\ \Delta\boldsymbol{\lambda} &= \mathbf{r}_{\Delta\mathbf{s}} \end{aligned} \tag{8}$$

where

$$\begin{aligned} \mathbf{r}_a &= -\mathbf{r}_d + \mathbf{A}_I^T\mathbf{S}^{-1}\boldsymbol{\Lambda}\left(-\mathbf{r}_I - \boldsymbol{\Lambda}^{-1}\mathbf{r}_{\Lambda S}\right) \\ \mathbf{r}_b &= -\mathbf{r}_E \\ \mathbf{r}_{\Delta\mathbf{x}} &= \mathbf{A}_I\Delta\mathbf{x} + \mathbf{r}_I \\ \mathbf{r}_{\Delta\mathbf{s}} &= -\mathbf{S}^{-1}\boldsymbol{\Lambda}(\boldsymbol{\Lambda}^{-1}\mathbf{r}_{\Lambda S} + \Delta\mathbf{s}). \end{aligned}$$

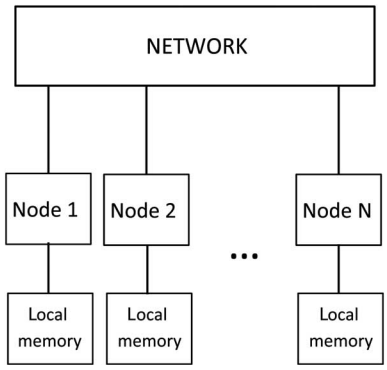

Fig. 2. Simple distributed environment model.

Here, $\mathbf{r}_{\Lambda S}$ is the last right term of both systems (6) and (7). Now the problem amounts to solving a $(p^2 + p) \times (p^2 + p)$ linear system for the computation of $\Delta\mathbf{x}$ and $\Delta\boldsymbol{\mu}$, and then for the successive computation of $\Delta\mathbf{s}$ and $\Delta\boldsymbol{\lambda}$ using just matrix to vector multiplications. In the following, we present several different optimizations for the algorithm described in this section.

IV. Parallel Programming Frameworks

This section describes the parallel frameworks that will be used to develop a distributed GPU implementation of the MVSA algorithm described in the following section. For the distributed part, we resort to the message passing interface (MPI).[4] For the GPU implementation, we use the CUDA[5] by `NVidia`,[6] the main GPU vendor worldwide.

A. MPI Programming

MPI is the most widely used library for distributed parallel programming. In a distributed computing environment, the nodes have access only to their local memory and each node may contain several processors, see Fig. 2. In order to access the memory content of the other nodes, a message transfer through the network needs to take place. A reduction operation among the nodes is a collective process that applies an operator to the memory contents of each node. In MPI, this operator is optimized and implemented by the built-in functions `MPI_Reduce`, where the operator is applied in one node, and `MPI_Allreduce`, where the operator is applied in all nodes. Broadcast is a collective operation where the memory pointed by one node is transferred to all other nodes, and it is implemented by the function `MPI_Bcast`. Scatter is a collective operation where the memory pointed by one node is evenly distributed to all nodes, e.g., this operation is relevant if we wish to evenly distribute the contents of a vector in one node. This operation is implemented by the function `MPI_Scatter`. These collective operations are described in detail in [35]. The bottleneck in a distributed environment is generally the communication network, thus the design of an effective distributed parallel program should avoid using excessively the network.

[4]Available: http://www.mcs.anl.gov/research/projects/mpi

[5]Available: https://developer.nvidia.com/cuda

[6]Available: http://www.nvidia.com

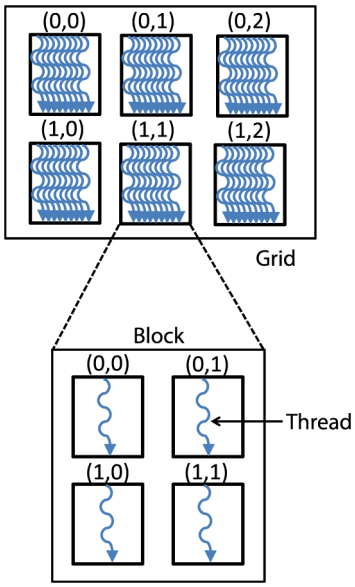

Fig. 3. Grid, block, and thread hierarchy in the CUDA model for GPU programming.

B. CUDA Programming

CUDA is the most commonly used programming framework for GPUs. Today GPUs are capable to execute massively light-weight threads of the same program (kernel). Thus, GPUs follow a single program multiple data (SPMD) model. In the finest level of a warp, GPUs follow the single instruction multiple data (SIMD) model. The program that is executed in the GPU is called kernel, and it is executed in parallel using up to thousands of threads. When a kernel is launched, it creates a grid containing blocks. The threads are executed inside the blocks. Threads and blocks can be one, two, and three dimensional, and they have an index space, as indicated in Fig. 3. In order to launch a kernel, there is a need to set the number of blocks and the number of threads that will be exectued inside the block. In our implementation of MVSA, there is a need to use two-dimensional blocks holding two-dimensional threads for matrix multiplications (performed by the library cuBLAS of NVIDIA) and also one dimensional blocks holding one-dimensional threads for operations like dot products, vector additions, pairwise vector multiplications, reduction operations, and the functions we will present later on.

The memory architecture of CUDA consists of a global memory (the video memory), which is connected via a high-bandwidth bus to the chip of the GPU, a shared on-chip memory (cache), and registers in the GPU chip. The global memory can be accessed by all threads in all blocks. The shared memory can be accessed only by the threads in a block, and registers can be accessed only by one thread in the block. A good programming practice is to have the threads access consecutive elements in global memory (coalescing) and also to use the shared memory when data are reused by threads. In our implementation of MVSA, we use this programming practice in the construction of the kernels presented later on. This means that consecutive threads in the kernels access consecutive memory locations in global memory and also elements that are reused are stored in the shared memory, which is significantly faster than the global memory.

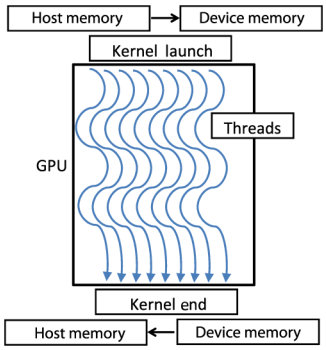

Fig. 4. CUDA execution model.

CUDA programming follows a host-device philosophy, meaning that the programmer can code for the CPU and GPU in the same program. In a simplified model, when programming for the GPU, the contents of the memory accessible from the CPU should be transferred to the memory accessible from the GPU with a CPU-to-GPU memory transfer, then the kernel is launched and the result is copied back to the host memory with a device to host memory transfer, as indicated in Fig. 4. We refer to [36] for a detailed introduction to CUDA programming.

The memory transfer between the host and device is generally performed via a relatively low bandwidth bus (such as PCI-Express), so a good GPU programming practice is to keep the data in the GPU memory and minimize memory transfers between the host and device. In a hybrid MPI/CUDA environment, such as the one that we will be using for our implementation, the memory transfers between several GPUs constitute a multistep process. First, a device to host memory transfer should take place. Then, a memory transfer happens from one host to another through the communication network, and then a host to device memory transfer takes place.[7] As a result, the memory exchange between GPUs is an expensive process and needs to be minimized. A good programming practice in this regard is to split the data across multiple GPUs and then perform local computations.

Algorithm 3: Pseudocode of an optimized predictor–corrector interior point algorithm

1: **INPUT:** $\mathbf{Y}$, $\mathbf{c}$, $\mathbf{G}$, $\mathbf{A}_E$, $\mathbf{b}_I$, $\mathbf{b}_E$

2: $(\mathbf{x}, \boldsymbol{\mu}, \mathbf{s}, \boldsymbol{\lambda}) \leftarrow (\mathbf{x_0}, \mathbf{e}, \mathbf{e}, \mathbf{e})$, $\mathbf{e} = [1, \ldots, 1]^T$

3: $\rho \leftarrow 1$, $\sigma \leftarrow 1$, $k \leftarrow 1$

4: **while** ($\sigma > 10^{-8}$ **or** $\rho > 10^{-8}$) **do**

5: $\quad \mathbf{s}^{-1}\boldsymbol{\lambda} \leftarrow \mathbf{s}^{-1}.*\boldsymbol{\lambda}$

6: $\quad \mathbf{r_d} \leftarrow \mathbf{G} * \mathbf{x} + \mathbf{c} - (\mathbf{L} * \mathbf{Y}^T)(:) + \mathbf{A}_E^{T*}\boldsymbol{\mu}$

7: $\quad \mathbf{r_I} \leftarrow (\mathbf{X} * \mathbf{Y})(:) - \mathbf{s} - \mathbf{b}_I$

8: $\quad \mathbf{r_E} \leftarrow \mathbf{A}_E * \mathbf{x} - \mathbf{b}_E$

[7]The new NVidia GPU-direct technology allows direct memory communication between GPUs in MPI. Available: https://developer.nvidia.com/gpudirect

9: $(\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda})_{compact} \leftarrow CalculateCompact(\mathbf{Y}, \mathbf{s}^{-1}\boldsymbol{\lambda}^T)$

10: $(\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda}\mathbf{A}_I)_{compact} \leftarrow (\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda})_{compact} * \mathbf{Y}^T$

11: $\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda}\mathbf{A}_I \leftarrow ConstructMatrix((\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda}\mathbf{A}_I)_{compact})$

12: $\mathbf{K} \leftarrow \mathbf{G} + \mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda}\mathbf{A}_I$

13: $\mathbf{Inv} \leftarrow \begin{bmatrix} \mathbf{K} & \mathbf{A}_E^T \\ \mathbf{A}_E & \mathbf{0} \end{bmatrix}^{-1}$

14: $\mathbf{rh} \leftarrow \mathbf{s}^{-1}\boldsymbol{\lambda} .* (\mathbf{r_I} + \mathbf{s})$

15: $\mathbf{rh} \leftarrow -\mathbf{r_d} - (\mathbf{RH} * \mathbf{Y}^T)(:)$

16: $\mathbf{rh} \leftarrow \begin{bmatrix} \mathbf{rh} \\ -\mathbf{r_E} \end{bmatrix}$

17: $\mathbf{\Delta x \Delta m} \leftarrow \mathbf{Inv} * \mathbf{rh}$

18: $\mathbf{\Delta x}^{aff} \leftarrow \mathbf{\Delta x \Delta m}(1 : p^2)$

19: $\mathbf{\Delta s}^{aff} \leftarrow (\mathbf{\Delta X}^{aff} * \mathbf{Y})(:) + \mathbf{r_I}$

20: $\mathbf{\Delta \lambda}^{aff} \leftarrow -\mathbf{s}^{-1}\boldsymbol{\lambda} .* (\mathbf{s} + \mathbf{\Delta} b\mathbf{s}^{aff})$

21: $\rho \leftarrow \frac{\mathbf{s}^T\boldsymbol{\lambda}}{n_I}$

22: $\alpha_{\mathbf{\Delta s}} \leftarrow \min_{\mathbf{\Delta s}^{aff} < 0} -\mathbf{s}./\mathbf{\Delta s}^{aff}$

23: $\alpha_{\mathbf{\Delta} b\boldsymbol{\lambda}} \leftarrow \min_{\mathbf{\Delta} b\boldsymbol{\lambda}^{aff} < 0} -\boldsymbol{\lambda}./\mathbf{\Delta} b\boldsymbol{\lambda}^{aff}$

24: $\alpha_{aff} \leftarrow min(\alpha_{\mathbf{\Delta} b\mathbf{s}}, \alpha_{\mathbf{\Delta} b\boldsymbol{\lambda}}, 1)$

25: $\rho_{aff} \leftarrow (\mathbf{s} + \alpha_{aff}\mathbf{\Delta} b\mathbf{s}^{aff})^T(\boldsymbol{\lambda} + \alpha_{aff}\mathbf{\Delta} b\boldsymbol{\lambda}^{aff})/n_I$

26: $\sigma \leftarrow \left(\frac{\rho_{aff}}{\rho}\right)^3$

27: $\mathbf{s}_{corrected} \leftarrow \mathbf{s} + \boldsymbol{\lambda}^{-1} .* \mathbf{\Delta}\boldsymbol{\lambda}^{aff} .* \mathbf{\Delta s}^{aff} - \sigma * \rho * \boldsymbol{\lambda}^{-1}$

28: $\mathbf{rh} \leftarrow \mathbf{s}^{-1}\boldsymbol{\lambda} .* (\mathbf{r_I} + \mathbf{s}_{corrected})$

29: $\mathbf{rh} \leftarrow -\mathbf{r_d} - (\mathbf{RH} * \mathbf{Y}^T)(:)$

30: $\mathbf{rh} \leftarrow \begin{bmatrix} \mathbf{rh} \\ -\mathbf{r_E} \end{bmatrix}$

31: $\mathbf{\Delta x \Delta m} \leftarrow \mathbf{Inv} * \mathbf{rh}$

32: $\mathbf{\Delta x} \leftarrow \mathbf{\Delta x \Delta m}(1 : p^2)$

33: $\mathbf{\Delta}\boldsymbol{\mu} \leftarrow \mathbf{\Delta x \Delta m}(p^2 + 1 : end)$

34: $\mathbf{\Delta s} \leftarrow (\mathbf{\Delta X} * \mathbf{Y})(:) + \mathbf{r_I}$

35: $\mathbf{\Delta}\boldsymbol{\lambda} \leftarrow -\mathbf{s}^{-1}\boldsymbol{\lambda} .* (\mathbf{s}_{corrected} + \mathbf{\Delta s})$

36: $\tau \leftarrow 1 - \frac{1}{k+1}$

37: $\alpha_{primal} \leftarrow \min_{\mathbf{\Delta s} < 0} -\tau * \mathbf{s}./\mathbf{\Delta s}$

38: $\alpha_{dual} \leftarrow \min_{\mathbf{\Delta}\boldsymbol{\lambda} < 0} -\tau * \boldsymbol{\lambda}./\mathbf{\Delta}\boldsymbol{\lambda}$

39: $\alpha \leftarrow min(\alpha_{primal}, \alpha_{dual}, 1)$

40: $(\mathbf{x}, \boldsymbol{\mu}, \mathbf{s}, \boldsymbol{\lambda}) \leftarrow (\mathbf{x}, \boldsymbol{\mu}, \mathbf{s}, \boldsymbol{\lambda}) + \alpha(\mathbf{\Delta x}, \mathbf{\Delta}\boldsymbol{\mu}, \mathbf{\Delta s}, \mathbf{\Delta}\boldsymbol{\lambda})$

41: $k \leftarrow k + 1$

42: **end while**

V. Efficient Implementations of MVSA

In this section, we describe several new strategies for efficient implementation of MVSA. First, we describe an efficient implementation based on algebraic operations. Then, we describe a single GPU implementation of MVSA. Finally, we present a multi-GPU implementation.

Algorithm 4: CUDA pseudocode of the optimized predictor–corrector interior point algorithm

1: **INPUT:** $\mathbf{Y}$, $\mathbf{c}$, $\mathbf{G}$, $\mathbf{A}_E$, $\mathbf{b}_I$, $\mathbf{b}_E$ //**Y is kept in global GPU memory since it is reused throughout MVSA**

2: $(\mathbf{x}, \boldsymbol{\mu}, \mathbf{s}, \boldsymbol{\lambda}) \leftarrow (\mathbf{x_0}, \mathbf{e}, \mathbf{e}, \mathbf{e})$, $\mathbf{e} = [1, \ldots, 1]^T$ //**All these values are set with simple kernels where each thread assigns a value**

3: $\rho \leftarrow 1$, $\sigma \leftarrow 1$, $k \leftarrow 1$

4: **while** ($\sigma > 10^{-8}$ **or** $\rho > 10^{-8}$) **do**

5: $\mathbf{s}^{-1}\boldsymbol{\lambda} \leftarrow \mathbf{s}^{-1} .* \boldsymbol{\lambda}$ //**the reciprocal and the pairwise multiplication is done by simple kenels where each thread computes the coreesponding value**

6: $\mathbf{r_d} \leftarrow \mathbf{cublasDgemv}(\mathbf{G}, \mathbf{x}) + \mathbf{c} - \mathbf{cublasDgemm}(\boldsymbol{\lambda}, \mathbf{Y}) + \mathbf{cublasDgemv}(\mathbf{A}_E, \boldsymbol{\mu})$

7: $\mathbf{r_I} \leftarrow \mathbf{cublasDgemm}(\mathbf{x}, \mathbf{Y^T}) - \mathbf{s} - \mathbf{b}_I$

8: $\mathbf{r_E} \leftarrow \mathbf{cublasDgemv}(\mathbf{A_E^T}, \mathbf{x}) - \mathbf{b}_E$

9: $(\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda})_{compact} \leftarrow CalculateCompact(\mathbf{Y}, \mathbf{s}^{-1}\boldsymbol{\lambda}^T)$ //**Kernel to transpose** $\mathbf{s}^{-1}\boldsymbol{\lambda}^T$

10: $(\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda}\mathbf{A}_I)_{compact} \leftarrow \mathbf{cublasDgemm}(\mathbf{Y}^T, (\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda})_{\mathbf{compact}})$

11: **Transfer** $(\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda}\mathbf{A}_I)_{compact}$ **to host memory**

12: $\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda}\mathbf{A}_I \leftarrow ConstructMatrix((\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda}\mathbf{A}_I)_{compact})$ //**host operation**

13: $\mathbf{K} \leftarrow \mathbf{G} + \mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda}\mathbf{A}_I$ //**Host operation**

14: $\mathbf{Inv} \leftarrow \begin{bmatrix} \mathbf{K} & \mathbf{A}_E^T \\ \mathbf{A_E} & \mathbf{0} \end{bmatrix}^{-1}$ //**Host operation with MKL**

15: Transfer Inv from host memory to GPU memory

16: $\mathbf{rh} \leftarrow \mathbf{s}^{-1}\boldsymbol{\lambda} .* (\mathbf{r_I} + \mathbf{s})$ //**Simple kernel operations as line 5**

17: $\mathbf{rh} \leftarrow -\mathbf{r_d} - \mathbf{cublasDgemm}(\mathbf{rh}, \mathbf{Y})$

18: $\mathbf{rh} \leftarrow \begin{bmatrix} \mathbf{rh} \\ -\mathbf{r_E} \end{bmatrix}$ //**cudaMemcpy from Device to Device**

19: $\mathbf{\Delta x \Delta m} \leftarrow \mathbf{cublasDgemv(Inv, rh)}$

20: $\mathbf{\Delta x}^{aff} \leftarrow \mathbf{\Delta x \Delta m}(1:p^2)$ //**Transfer the first p^2 elements with cudaMemcpy**

21: $\mathbf{\Delta s}^{aff} \leftarrow \mathbf{cublasDgemm}(\mathbf{\Delta x}^{\mathbf{aff}}, \mathbf{Y}^T) + \mathbf{r_I}$

22: $\mathbf{\Delta \lambda}^{aff} \leftarrow -\mathbf{s}^{-1}\mathbf{\lambda} .* (\mathbf{s} + \mathbf{\Delta s}^{aff})$ //**simple kernel operations as line 5**

23: $\rho \leftarrow \frac{\mathbf{cublasDdot(s,\lambda)}}{n_I}$

24: $\alpha_{\mathbf{\Delta s}} \leftarrow \mathbf{thrust :: reduce}(-\mathbf{s} ./ \mathbf{\Delta s}^{\mathbf{aff}}, \mathbf{thrust :: minimum})$

25: $\alpha_{\mathbf{\Delta \lambda}} \leftarrow \mathbf{thrust :: reduce}(-\mathbf{\lambda} ./ \mathbf{\Delta \lambda}^{\mathbf{aff}}, \mathbf{thrust :: minimum})$

26: $\alpha_{aff} \leftarrow min(\alpha_{\mathbf{\Delta s}}, \alpha_{\mathbf{\Delta \lambda}}, 1)$

27: $\rho_{aff} \leftarrow \mathbf{cublasDdot}(\mathbf{s} + \alpha_{\mathbf{aff}}\mathbf{\Delta s}^{\mathbf{aff}}, \mathbf{\lambda} + \alpha_{\mathbf{aff}}\mathbf{\Delta \lambda}^{\mathbf{aff}})/\mathbf{n_I}$

28: $\sigma \leftarrow \left(\frac{\rho_{aff}}{\rho}\right)^3$

29: $\mathbf{s}_{corrected} \leftarrow \mathbf{s} + \mathbf{\lambda}^{-1} .* \mathbf{\Delta \lambda}^{aff} .* \mathbf{\Delta s}^{aff} - \sigma * \rho * \mathbf{\lambda}^{-1}$ //**Simple kernel operations as line 5**

30: $\mathbf{rh} \leftarrow \mathbf{s}^{-1}\mathbf{\lambda} .* (\mathbf{r_I} + \mathbf{s}_{corrected})$ //**Simple kernel operations as line 5**

31: $\mathbf{rh} \leftarrow -\mathbf{r_d} - \mathbf{cublasDgemm(rh, Y)}$

32: $\mathbf{rh} \leftarrow \begin{bmatrix} \mathbf{rh} \\ -\mathbf{r_E} \end{bmatrix}$ //**memory transfer with cudaMemcpy from device to device**

33: $\mathbf{\Delta x \Delta m} \leftarrow \mathbf{cublasDgemv(Inv, rh)}$

34: $\mathbf{\Delta x} \leftarrow \mathbf{\Delta x \Delta m}(1:p^2)$ //**Transfer the first p^2 elements with cudaMemcpy**

35: $\mathbf{\Delta \mu} \leftarrow \mathbf{\Delta x \Delta m}(p^2+1:end)$ //**Transfer the last elements with cudaMemcpy**

36: $\mathbf{\Delta s} \leftarrow \mathbf{cublasDgemm}(\mathbf{\Delta x}, \mathbf{Y}) + \mathbf{r_I}$

37: $\mathbf{\Delta \lambda} \leftarrow -\mathbf{s}^{-1}\mathbf{\lambda} .* (\mathbf{s}_{corrected} + \mathbf{\Delta s})$ //**Simple kernel operations as line 5**

38: $\tau \leftarrow 1 - \frac{1}{k+1}$

39: $\alpha_{primal} \leftarrow \mathbf{thrust :: reduce}(-\tau * \mathbf{s} ./ \mathbf{\Delta s}, \mathbf{thrust :: minimum})$

40: $\alpha_{dual} \leftarrow \mathbf{thrust :: reduce}(-\tau * \mathbf{\lambda} ./ \mathbf{\Delta \lambda}, \mathbf{thrust :: minimum})$

41: $\alpha \leftarrow min(\alpha_{primal}, \alpha_{dual}, 1)$

42: $(\mathbf{x}, \mathbf{\mu}, \mathbf{s}, \mathbf{\lambda}) \leftarrow (\mathbf{x}, \mathbf{\mu}, \mathbf{s}, \mathbf{\lambda}) + \alpha(\mathbf{\Delta x}, \mathbf{\Delta \mu}, \mathbf{\Delta s}, \mathbf{\Delta \lambda})$ //**Update with simple kernels like line 5**

43: $k \leftarrow k + 1$

44: **end while**

A. Efficient Implementation Using Algebra Operations

The optimization described in this section is mainly related with the management of matrix $\mathbf{A}_I$ in (8). The size of this matrix is $Np \times p^2$, which means that the matrix is very large to store and make calculations. However, it is possible to replace the operations involving $\mathbf{A}_I$ with operations using lower dimensional matrices, making them very suitable for a GPU implementation. As a result, a main goal in this section is to redefine the pseudocode of Algorithm 2 using the optimizations described. The first operation that needs to be replaced is $\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda}\mathbf{A}_I$. In this operation, the first term to be computed is $\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda}$. It can be easily shown that this multiplication can be expressed using the following compact form:

$$\begin{bmatrix} \frac{\lambda_0}{s_0}Y_{00} & \frac{\lambda_p}{s_p}Y_{01} & \cdots & \frac{\lambda_{(N-1)p}}{s_{(N-1)p}}Y_{0N-1} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\lambda_{p-1}}{s_{p-1}}Y_{00} & \frac{\lambda_{2p-1}}{s_{2p-1}}Y_{01} & \cdots & \frac{\lambda_{Np-1}}{s_{Np-1}}Y_{0N-1} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\lambda_0}{s_0}Y_{(p-1)0} & \frac{\lambda_p}{s_p}Y_{(p-1)1} & \cdots & \frac{\lambda_{(N-1)p}}{s_{(N-1)p}}Y_{(p-1)N-1} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\lambda_{p-1}}{s_{p-1}}Y_{(p-1)0} & \frac{\lambda_{2p-1}}{s_{2p-1}}Y_{(p-1)1} & \cdots & \frac{\lambda_{Np-1}}{s_{Np-1}}Y_{(p-1)N-1} \end{bmatrix}$$

which will be referred to hereinafter as $(\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda})_{compact}$. It should be noted that without loss of generality, we can also write $(\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda}\mathbf{A}_I)_{compact} = (\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda})_{compact}\mathbf{Y}^T$. With these definitions in mind, the final form of the matrix $\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda}\mathbf{A}_I$ can be simply constructed using Algorithm 5.

Algorithm 5: Function ConstructMatrix

1: **INPUT:** $(\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda}\mathbf{A}_I)_{compact}$

2: $\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda}\mathbf{A}_I \leftarrow \mathbf{0}$

3: **for** i = 0 **to** p-1 **do**

4: **for** k = 0 **to** p-1 **do**

5: **for** j = 0 **to** p-1 **do**

6: $\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda}\mathbf{A}_I[k + j*p + (k + i*p)*p^2] \leftarrow (\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda}\mathbf{A}_I)_{comp}[j + (k + i*p)*p]$

7: **end for**

8: **end for**

9: **end for**

As it can be seen in the system of (8), the matrix $\mathbf{A}_I$ is also used to compute the terms $\mathbf{r}_a$, $\mathbf{r}_{\mathbf{\Delta x}}$, $\mathbf{r}_{\mathbf{I}}$, and $\mathbf{r}_{\mathbf{d}}$. The ultimate goal is to compute terms $\mathbf{A}_I\mathbf{v}$ and $\mathbf{A}_I^T\mathbf{v}$. Here, the term $\mathbf{A}_I\mathbf{v}$ can be computed by $\mathbf{VY}(:)$ (following Matlab notation) and the term $\mathbf{A}_I^T\mathbf{v}$ can be computed by $\mathbf{VY}^T(:)$, where $\mathbf{V}$ is the matrix created by the vector $\mathbf{v}$ in a column-major order and the symbol $(:)$ is the column vector created by a matrix in a column-major order also. It can also be observed that the matrices $\mathbf{\Lambda}$ and $\mathbf{S}$ do not need to be stored explicitly, only their diagonals are needed. As a result, there is no need to store extremely large matrices since the largest one that needs to be stored is of dimension $p^2 \times N$. This is extremely useful for our GPU implementation since GPUs generally have much less memory than the host. The pseudocode of the predictor–corrector interior point algorithm after the aforementioned optimizations is given in Algorithm 3.

In Algorithm 3, lines 9–11 compute the matrix $\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda}\mathbf{A}_I$, lines 12–20 compute the affine Newton step, lines 21–26 compute the parameters ρ and σ, lines 27–35 compute the final Newton step, and lines 35–40 compute the final feasible step of the iteration. Notice that $\mathbf{L}$, $\mathbf{X}$, and $\mathbf{RH}$ denote the matrices formed by the vectors $\mathbf{\lambda}$, $\mathbf{x}$, and $\mathbf{rh}$. The notations "$.*$" and "$./$" represent member-wise multiplication and division, respectively. The function $CalculateCompact(\mathbf{Y}, \mathbf{\lambda}^T)$ calculates the matrix $(\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda})_{compact}$ and will be described in the next subsection which is devoted to the specific implementation of the algorithm in GPU architectures.

B. GPU Implementation

The main operations that need to be optimized for the efficient implementation of MVSA in GPUs can be summarized as follows (see Algorithm 3).

1) Member-wise operations involving vectors (addition, multiplication, and division). These operations can be efficiently executed in the GPU by means of kernels.
2) Vector–vector operations (inner products), which can be optimized by resorting to linear algebra libraries specifically developed for GPUs. In this work, we use the cuBLAS library for this purpose.
3) Matrix–vector and matrix–matrix multiplications also optimized by using cuBLAS.
4) Reduction operations, which can be effectively implemented using the Thrust library.
5) Inverse matrix operations, which are more suitable for implementation in the host (taking advantage of multicore technology if available). For this purpose, here we use Intel's math kernel library (MKL),[8] which has been shown to be very effective in this kind of operations.
6) Finally, the vector transpose operation for computing $(\mathbf{\lambda}^T)$ can be accomplished in the GPU by means of a kernel function.

It should be noted that the transfers from device to host memory that need to be performed in our GPU implementation mostly take place during the calculation of the final matrix $\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda}\mathbf{A}_I$. However, the amount of data that needs to be transferred at each algorithm iteration is on the order of p^2, which makes the transfer time practically negligible when compared with the other computations. The inverse of the matrix can also be computed very fast since the size of the matrix is on the order of $p^2 + p$. In fact, this computation can be carried out very efficiently using MKL. The host to device memory transfers that need to be performed are also mainly related with the provision of the inverse to the GPU. Since the inverse is on the order of $p^2 + p$, this transfer is also trivial. All other involved operations do not require memory exchange between the host and device since the data involved in the operations reside only on the device memory.

Algorithm 6: Kernel CalculateCompact

1: **INPUT:** $\mathbf{Y}$, $\mathbf{s}^{-1}\mathbf{\lambda}^T$

2: **Shared** sY[1024]

3: $tx \leftarrow threadIdx.x$, $bx \leftarrow blockIdx.x$

4: $by \leftarrow blockIdx.y$

5: $Col \leftarrow tx + bx * blockDim.x$

6: **if** $Col < N$ **then**

7: $\quad sY[tx] \leftarrow Y[Col + by * N]$

8: **end if**

9: SynchronizeThreads

10: $startRow \leftarrow by * p$;

11: **for** i = 0 **to** p-1 **do**

12: $\quad Row \leftarrow startRow + i$

13: $\quad$ **if** $Col < N$ **then**

14: $\quad\quad (\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda})_{compact}[Col + Row * N] \leftarrow sY[tx] *$ $\mathbf{s}^{-1}\mathbf{\lambda}^T[Col + i * N]$

15: $\quad$ **end if**

16: **end for**

The only remaining nontrivial kernel that needs to be constructed for the GPU implementation is the one that calculates the matrix $(\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda})_{compact}$. For illustrative purposes, this kernel (called CalculateCompact) is illustrated in Algorithm 6. The kernel is initialized with $dim3 = (1024, 1)$ threads and $dim3 = (N/1024, p)$ blocks. An important component of this kernel is that it uses shared memory to store the elements of $\mathbf{Y}$, which are read in a coalesced manner. The elements of $\mathbf{s}^{-1}\mathbf{\lambda}$ and $(\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda})_{compact}$ in line 14 of Algorithm 6 are also read and written in a coalesced manner, respectively. This makes the kernel optimal for the calculation of $(\mathbf{A}_I^T\mathbf{S}^{-1}\mathbf{\Lambda})_{compact}$.

For the sake of completeness, we present Algorithm 4 that embeds on top of Algorithm 3 in a pseudocode manner; the basic CUDA operations needed to implement the later. Algorithm 4 shows the precise mapping to GPU functions and it can be observed that the mapping to Algorithm 3 is exact. As it can be observed, we efficiently use all the described functions to avoid using the large matrix $\mathbf{A}_I$, thus reducing the amount of memory

[8]Available: http://software.intel.com/en-us/intel-mkl

needed on the GPU. The basic operations like addition of vectors, subtraction, and multiplication with a constant and pairwise multiplication can be achieved with simple kernels, and they are similar with the very basic algorithms presented in the introduction of [36].

Algorithm 7: Pseudocode of a distributed optimized predictor–corrector interior point algorithm

1: **INPUT: All nodes:** $\mathbf{Y}_{prt}^T$, $\mathbf{Y}_{prt}$, $\mathbf{b}_I^{prt}$, $\mathbf{c}$, $\mathbf{G}$, $\mathbf{A}_E$, $\mathbf{b_E}$

2: $(\mathbf{s}_{prt}, \boldsymbol{\lambda}_{prt}) \leftarrow (\mathbf{e}, \mathbf{e})$

3: $(\mathbf{x}, \boldsymbol{\mu}) \leftarrow (\mathbf{x_0}, \mathbf{e})$

4: $\rho \leftarrow 1$, $\sigma \leftarrow 1$, $k \leftarrow 1$

5: **while** ($\sigma > 10^{-8}$ **or** $\rho > 10^{-8}$) **do**

6: $\mathbf{s}^{-1}\boldsymbol{\lambda}_{prt} \leftarrow \mathbf{s}_{prt}^{-1} .* \boldsymbol{\lambda}_{prt}$

7: $(\mathbf{A}_I^T\mathbf{S}^{-1}\boldsymbol{\Lambda})_{compact}^{prt} \leftarrow CalculateCompact(\mathbf{Y}_{prt}, \mathbf{s}^{-1}\boldsymbol{\lambda}_{prt}^T)$

8: $(\mathbf{A}_I^T\mathbf{S}^{-1}\boldsymbol{\Lambda}\mathbf{A}_I)_{compact}^{prt} \leftarrow (\mathbf{A}_I^T\mathbf{S}^{-1}\boldsymbol{\Lambda})_{compact}^{prt} * \mathbf{Y}_{prt}^T$

9: $\mathbf{Allreduce}((\mathbf{A}_I^T\mathbf{S}^{-1}\boldsymbol{\Lambda}\mathbf{A}_I)_{compact}^{prt}, (\mathbf{A}_I^T\mathbf{S}^{-1}\boldsymbol{\Lambda}\mathbf{A}_I)_{compact}, p^3, SUM, master)$

10: $\mathbf{A}^T\boldsymbol{\lambda}_{prt} \leftarrow (\mathbf{L}_{prt} * \mathbf{Y}_{prt}^T)(:)$

11: $\mathbf{Allreduce}(\mathbf{A}^T\boldsymbol{\lambda}_{prt}, \mathbf{A}^T\boldsymbol{\lambda}, p^2, SUM, master)$

12: $\mathbf{r}_I^{prt} \leftarrow (\mathbf{X} * \mathbf{Y}_{prt})(:) - \mathbf{s}_{prt} - \mathbf{b}_I^{prt}$

13: $\mathbf{rh}_{prt} \leftarrow \mathbf{s}^{-1}\boldsymbol{\lambda}_{prt} .* (\mathbf{r}_I^{prt} + \mathbf{s}_{prt})$

14: $\mathbf{rh}_{prt} \leftarrow (\mathbf{RH}_{prt} * \mathbf{Y}_{prt}^T)(:)$

15: $\mathbf{Allreduce}(\mathbf{rh}_{prt}, \mathbf{rh}, p^2, SUM, master)$

16: $\mathbf{r_d} \leftarrow \mathbf{G} * \mathbf{x} + \mathbf{c} - \mathbf{A}^T\boldsymbol{\lambda} + \mathbf{A}_E^T * \boldsymbol{\mu}$

17: $\mathbf{r_E} \leftarrow \mathbf{A}_E * \mathbf{x} - \mathbf{b}_E$

18: $\mathbf{A}_I^T\mathbf{S}^{-1}\boldsymbol{\Lambda}\mathbf{A}_I \leftarrow ConstructMatrix((\mathbf{A}_I^T\mathbf{S}^{-1}\boldsymbol{\Lambda}\mathbf{A}_I)_{compact})$

19: $\mathbf{K} \leftarrow \mathbf{G} + \mathbf{A}_I^T\mathbf{S}^{-1}\boldsymbol{\Lambda}\mathbf{A}_I$

20: $\mathbf{Inv} \leftarrow \begin{bmatrix} \mathbf{K} & \mathbf{A}_E^T \\ \mathbf{A}_E & \mathbf{0} \end{bmatrix}^{-1}$

21: $\mathbf{rh} \leftarrow -\mathbf{r_d} - \mathbf{rh}$

22: $\mathbf{rh}^{aug} \leftarrow \begin{bmatrix} \mathbf{rh} \\ -\mathbf{r_E} \end{bmatrix}$

23: $\boldsymbol{\Delta}\mathbf{x}\boldsymbol{\Delta}\mathbf{m} \leftarrow \mathbf{Inv} * \mathbf{rh}^{aug}$

24: $\boldsymbol{\Delta}\mathbf{x}^{aff} \leftarrow \boldsymbol{\Delta}\mathbf{x}\boldsymbol{\Delta}\mathbf{m}(1 : p^2)$

25: $\boldsymbol{\Delta}\mathbf{s}_{prt}^{aff} \leftarrow (\boldsymbol{\Delta}\mathbf{X}^{aff} * \mathbf{Y}_{prt})(:) + \mathbf{r}_I^{prt}$

26: $\boldsymbol{\Delta}\boldsymbol{\lambda}_{prt}^{aff} \leftarrow -\mathbf{s}^{-1}\boldsymbol{\lambda}_{prt} .* (\mathbf{s}_{prt} + \boldsymbol{\Delta}\mathbf{s}_{prt}^{aff})$

27: $\rho^{local} \leftarrow \mathbf{s}_{prt}^T\boldsymbol{\lambda}_{prt}$

28: $\mathbf{AllReduce}(\rho^{local}, \rho, SUM)$

29: $\rho \leftarrow \rho / n_I$

30: $\alpha_{\boldsymbol{\Delta}\mathbf{s}}^{local} \leftarrow \min_{\boldsymbol{\Delta}\mathbf{s}_{prt}^{aff} < 0} -\mathbf{s}_{prt} ./ \boldsymbol{\Delta}\mathbf{s}_{prt}^{aff}$

31: $\mathbf{AllReduce}(\alpha_{\boldsymbol{\Delta}\mathbf{s}}^{local}, \alpha_{\boldsymbol{\Delta}\mathbf{s}}, MIN)$

32: $\alpha_{\boldsymbol{\Delta}\boldsymbol{\lambda}}^{local} \leftarrow \min_{\boldsymbol{\Delta}\boldsymbol{\lambda}_{prt}^{aff} < 0} -\boldsymbol{\lambda}_{prt} ./ \boldsymbol{\Delta}\boldsymbol{\lambda}_{prt}^{aff}$

33: $\mathbf{AllReduce}(\alpha_{\boldsymbol{\Delta}\boldsymbol{\lambda}}^{local}, \alpha_{\boldsymbol{\Delta}\boldsymbol{\lambda}}, MIN)$

34: $\alpha_{aff} \leftarrow min(\alpha_{\boldsymbol{\Delta}\mathbf{s}}, \alpha_{\boldsymbol{\Delta}\boldsymbol{\lambda}}, 1)$

35: $\rho_{aff}^{local} \leftarrow (\mathbf{s}_{prt} + \alpha_{aff}\boldsymbol{\Delta}\mathbf{s}_{prt}^{aff})^T(\boldsymbol{\lambda}_{prt} + \alpha_{aff}\boldsymbol{\Delta}\boldsymbol{\lambda}_{prt}^{aff})$

36: $\mathbf{AllReduce}(\rho_{aff}^{local}, \rho_{aff}, SUM)$

37: $\rho_{aff} \leftarrow \rho_{aff} / n_I$

38: $\sigma \leftarrow \left(\frac{\rho_{aff}}{\rho}\right)^3$

39: $\mathbf{s}_{corrected}^{prt} \leftarrow \mathbf{s}_{prt} + \boldsymbol{\lambda}_{prt}^{-1} .* \boldsymbol{\Delta}\boldsymbol{\lambda}_{prt}^{aff} .* \boldsymbol{\Delta}\mathbf{s}_{prt}^{aff} - \sigma * \rho * \boldsymbol{\lambda}_{prt}^{-1}$

40: $\mathbf{rh}_{prt} \leftarrow \mathbf{s}^{-1}\boldsymbol{\lambda}_{prt} .* (\mathbf{r}_I^{prt} + \mathbf{s}_{corrected}^{prt})$

41: $\mathbf{rh}_{prt} \leftarrow (\mathbf{RH}_{prt} * \mathbf{Y}_{prt}^T)(:)$

42: $\mathbf{Allreduce}(\mathbf{rh}_{prt}, \mathbf{rh}, p^2, SUM, master)$

43: $\mathbf{rh} \leftarrow -\mathbf{r_d} - \mathbf{rh}$

44: $\mathbf{rh}^{aug} \leftarrow \begin{bmatrix} \mathbf{rh} \\ -\mathbf{r_E} \end{bmatrix}$

45: $\boldsymbol{\Delta}\mathbf{x}\boldsymbol{\Delta}\mathbf{m} \leftarrow \mathbf{Inv} * \mathbf{rh}^{aug}$

46: $\boldsymbol{\Delta}\mathbf{x} \leftarrow \boldsymbol{\Delta}\mathbf{x}\boldsymbol{\Delta}\mathbf{m}(1 : p^2)$

47: $\boldsymbol{\Delta}\boldsymbol{\mu} \leftarrow \boldsymbol{\Delta}\mathbf{x}\boldsymbol{\Delta}\mathbf{m}(p^2 + 1 : end)$

48: $\boldsymbol{\Delta}\mathbf{s}_{prt} \leftarrow (\boldsymbol{\Delta}\mathbf{X} * \mathbf{Y}_{prt})(:) + \mathbf{r}_I^{prt}$

49: $\boldsymbol{\Delta}\boldsymbol{\lambda}_{prt} \leftarrow -\mathbf{s}^{-1}\boldsymbol{\lambda}_{prt} .* (\mathbf{s}_{corrected}^{prt} + \boldsymbol{\Delta}\mathbf{s}_{prt})$

50: $\tau \leftarrow 1 - \frac{1}{k+1}$

51: $\alpha_{primal}^{local} \leftarrow \min_{\boldsymbol{\Delta}\mathbf{s}_{prt} < 0} -\tau * \mathbf{s}_{prt} ./ \boldsymbol{\Delta}\mathbf{s}_{prt}$

52: $\mathbf{AllReduce}(\alpha_{primal}^{local}, \alpha_{primal}, MIN)$

53: $\alpha_{dual}^{local} \leftarrow \min_{\boldsymbol{\Delta}\boldsymbol{\lambda}_{prt} < 0} -\tau * \boldsymbol{\lambda}_{prt} ./ \boldsymbol{\Delta}\boldsymbol{\lambda}_{prt}$

54: $\mathbf{AllReduce}(\alpha_{dual}^{local}, \alpha_{dual}, MIN)$

55: $\alpha \leftarrow min(\alpha_{primal}, \alpha_{dual}, 1)$

56: $(\mathbf{x}, \boldsymbol{\mu}, \mathbf{s}_{prt}, \boldsymbol{\lambda}_{prt}) \leftarrow (\mathbf{x}, \boldsymbol{\mu}, \mathbf{s}_{prt}, \boldsymbol{\lambda}_{prt}) + \alpha(\boldsymbol{\Delta}\mathbf{x}, \boldsymbol{\Delta}\boldsymbol{\mu}, \boldsymbol{\Delta}\mathbf{s}_{prt}, \boldsymbol{\Delta}\boldsymbol{\lambda}_{prt})$

57: $k \leftarrow k + 1$

58: **end while**

C. Distributed GPU Implementation

This section describes the methodology followed in order to make the GPU implementation Algorithm 3 distributed, using the MPI library. The result is a hybrid MPI+CUDA implementaion, which is not easy to design due to several additional aspects that need to be managed, including the CPU-GPU communication, multi-GPU memory transfers, and buffer transfers happening between different nodes connected through a high-speed communication network, as illustrated in Fig. 2. Without careful handling of these transfers, the hybrid

TABLE I
COMPARISON OF ENDMEMBER IDENTIFICATION ALGORITHMS FOR A SYNTHETIC IMAGE WITHOUT PURE SIGNATURES, SIMULATED USING SPECTRAL SIGNATURES OBTAINED FROM THE USGS SPECTRAL LIBRARY AND WITH DIFFERENT NOISE LEVELS

	MVSA			MVES			N-FINDR		
dB	$\|\epsilon\|_F$	$r\epsilon$	SAD	$\|\epsilon\|_F$	$r\epsilon$	SAD	$\|\epsilon\|_F$	$r\epsilon$	SAD
90	8e-4	1.7e-8	0.0367	4.3e-3	1.7e-8	0.2545	0.46	1.3e-4	18
70	8e-4	1.67e-7	0.04	1.8e-3	1.67e-7	0.0857	0.59	1.15e-4	21.4
50	2.3e-3	1.72e-6	0.1228	4e-3	1.72e-6	0.1964	0.39	1.42e-4	17.4
30	2.1e-2	1.75e-5	1.2843	2.2e-2	1.75e-5	1.1602	0.43	1.28e-4	18.3

MPI+CUDA implementation could be slower than the single GPU implementation. The general paradigm that we have followed in the design of the hybrid version is to make calculations as local as possible, thus reducing the cost of communications as much as possible. For that purpose, we partition the hyperspectral image $\mathbf{Y}$ in full pixel vectors, meaning that a single pixel entity is never partitioned across different nodes in order to reduce interprocessor communications [37].

Let the number of nodes (processes) available in the parallel system be denoted as n. In a first step, we partition $\mathbf{Y}^T$ in row-wise fashion obtaining n submatrices of size $\frac{N}{n} \times p$ each. For simplicity, we assume that N is divisible by n although this is not an absolute requirement and the formulation can be easily redefined when this is not the case. This operation is performed by a simple `MPI_Scatter` operation. Resulting from this, all the nodes receive a part of $\mathbf{Y}^T$, denoted as $\mathbf{Y}^T_{prt}$. Then each node computes $\mathbf{Y}_{prt} = (\mathbf{Y}^T_{prt})^T$. With this strategy, the slack $\mathbf{s}$, the inequality lagrangian, $\boldsymbol{\lambda}$ and $\mathbf{r}_I$ variables can be split in n parts holding $\frac{N}{n}p$ elements each. Each node is then assigned one of these parts, denoted as $\mathbf{s}_{prt}$, $\boldsymbol{\lambda}_{prt}$, and $\mathbf{r}_I^{prt}$, respectively. $\mathbf{b}_I$ is also scattered to the nodes holding each a part of it with $\frac{N}{n}p$ elements, denoted as $\mathbf{b}_I^{prt}$. With these ideas in mind, Algorithm 7 presents a pseudocode for the distributed implementation of Algorithm 3.

Although Algorithm 7 appears to be complex, it can be systematically explained as follows (the algorithm runs on all nodes). The first lines of Algorithm 7 are intended to partitioning the data. Lines 10–12 in Algorithm 7 calculate $(\mathbf{A}_I^T\mathbf{S}^{-1}\boldsymbol{\Lambda}\mathbf{A}_I)_{compact}$. It should be noted that the function $CalculateCompact$ (presented in Section V-B) can still be used with the proper parameters $\mathbf{Y}_{prt}$ and $\mathbf{s}^{-1}\boldsymbol{\lambda}^T_{prt}$, and N is replaced by $\frac{N}{n}$ in the kernel input. Lines 19–32 in Algorithm 7 compute the affine Newton step, which is distributed for the slack and inequality lagrangian variables. Lines 33–43 of Algorithm 7 compute ρ and σ. Lines 45–58 of Algorithm 7 compute the final Newton step. Finally, lines 59–68 of Algorithm 7 compute the corrected step of the iteration.

At this point, it is important to emphasize that only local computations and collective communications are used in the implementation of Algorithm 7. The most significant communication and memory transfers involved in Algorithm 7 are in the order of p^3, which is quite reasonable since the number of endmembers p in a standard hyperspectral scene is rarely superior to 30. Collective communications (Allreduce) are necessary to provide the nodes with the necessary elements to compute the affine and final Newton steps. It is also worth noting that the distributed Algorithm 7 bears a strong resemblance with regards to Algorithm 3, except for the fact that Algorithm 7 resorts to local variables and matrices, and also uses collective communications in order to obtain the necessary elements. From the discussion above, we can conclude that the communication needed in Algorithm 7 is minimized to the absolute necessary.

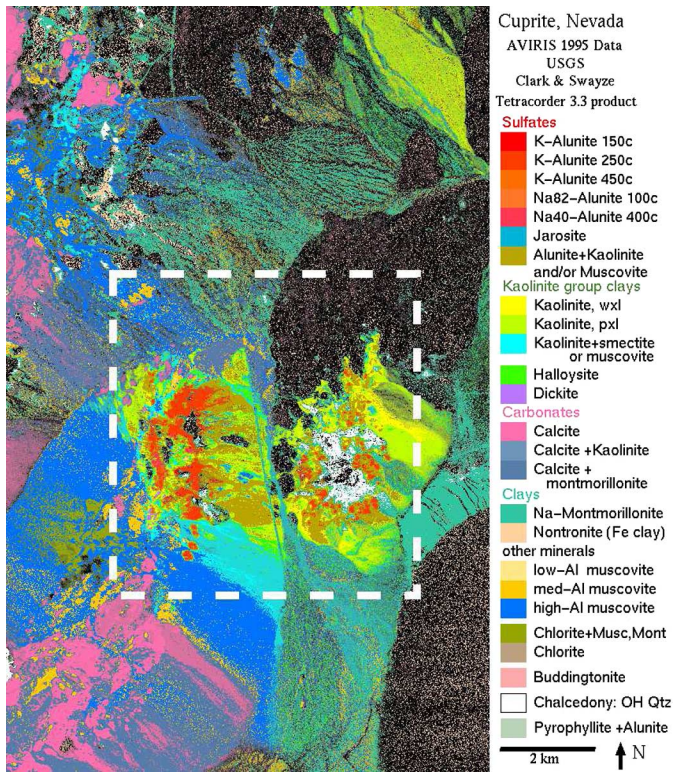

Fig. 5. USGS map showing the location of different minerals in the Cuprite mining district in Nevada. The map is available online at: http://speclab.cr.usgs.gov/cuprite95.tgif.2.2um_map.gif.

In Section VI, we illustrate the parallel performance of our newly developed, hybrid MPI+CUDA implementation. In terms of GPU operations, the algorithm performs the same operations in all the distributed GPUs as in the single GPU case. The only difference is that in the collective `MPI_Allreduce` operations, additional host to GPU memory transfers should be made. However, as mentioned before, these operations are in the order

TABLE II
SAD BETWEEN FIVE USGS REFERENCE MINERAL SPECTRAL SIGNATURES AND THE ENDMEMBERS EXTRACTED BY DIFFERENT METHODS FROM THE AVIRIS CUPRITE SCENE

Algorithm	Alunite	Buddingtonite	Calcite	Kaolinite	Muscovite
MVSA	6.3749	5.7251	4.9491	8.5991	5.8432
MVES	5.5992	5.6729	4.9426	6.7488	6.0219
N-FINDR	4.8125	4.2659	5.9315	7.9575	4.6310

of p^3, which is quite low given the actual number of endmembers in standard hyperspectral scenes. Also it should be noted that all nodes have the same computational burden making our computation and communications symmetric. An important consideration at this point is that the memory used by each GPU in the hybrid implementation is n times lower than in the single GPU case.

VI. EXPERIMENTAL RESULTS

In this section, we present an experimental assessment of the efficient implementations of MVSA described in Section III. The core of the MVSA algorithm is the sequential quadratic programming procedure described in Algorithm 1 for which we have developed several optimization strategies that will be evaluated next from the viewpoint of both endmember identification accuracy and parallel performance.

A. Evaluation of Endmember Identification Accuracy

As discussed in Section III, we first developed an optimized version of MVSA in the C programming language, and then obtained a GPU implementation using CUDA and a hybrid MPI+CUDA distributed multi-GPU implementation. It should be noted that the three versions produce exactly the same results. Hence, the only difference between these versions is their computational performance. For the sake of evaluating the endmember identification accuracy of the three implemented optimizations, we perform a comparison with other algorithms such as N-FINDR [19] (taken here as a representative of methods with the pure pixel assumption) and MVES [31], a state-of-the-art method without the pure pixel assumption. The metrics used in our comparison are the following ones:

1) Mean square error (MSE), denoted as $\|\epsilon\|_F = \|\widehat{\mathbf{M}} - \mathbf{M}\|_F$, where $\|\cdot\|_F$ stands for the Frobenius norm.
2) Reconstruction error, computed as $r\epsilon = \left\|\widehat{\mathbf{Y}} - \mathbf{Y}\right\|_F = \|\widehat{\mathbf{M}}\widehat{\mathbf{A}} - \mathbf{Y}\|_F$.
3) Spectral angle distance (SAD), expressed as $\mathrm{SAD} = \cos^{-1}(\frac{\mathbf{m}_i^T \widehat{\mathbf{m}}_i}{\|\mathbf{m}_i\|\|\widehat{\mathbf{m}}_i\|})$ and measured in degrees as defined in [4].

In order to perform a comparison of algorithms in a fully controlled environment, we first use a synthetic hyperspectral image consisting of $N = 100 \times 100$ pixels and constructed using the linear mixture model, following the simulation procedure described in [38] so that no pure pixels are present in the simulated scenes, which is dominated by highly mixed pixels. The spectral signatures used in the generation of the hyperspectral scene are selected from the United States Geological Survey (USGS) library denoted splib06[9] and released in September 2007 [39], and the number of signatures used for the simulation in our experiments is set to $p = 5$ and are set this low because we want to test the precision on high levels of noise. Noise in different proportions has been added to the simulated scene in order to evaluate the impact of noise on algorithm performance. Table I shows a comparison of MVSA, MVES, and N-FINDR using the aforementioned metrics for the considered synthetic scene simulated with different noise levels (in dBs). As indicated by Table I, the proposed implementations of MVSA outperform the other tested algorithms in terms of $\|\epsilon\|_F$, $r\epsilon$ and, particularly, SAD. This was expected in comparison with N-FINDR, since the synthetic images are simulated without pure pixels and N-FINDR belongs the the category of endmember identification algorithms with the pure pixel assumption. However, it is remarkable that the performance metrics obtained by MVSA are slightly superior to those reported for MVES, which is a state-of-the-art algorithm in the same category as MVSA, i.e., without the pure pixel assumption.

In a second experiment, we evaluate the performance of the implementations of MVSA using a real hyperspectral data set. The scene used in our real data experiments is the well-known Airborne Visible Infra-Red Imaging Spectrometer (AVIRIS) Cuprite data set, available online in reflectance units.[10] This scene has been widely used to validate the performance of endmember extraction algorithms. The portion used in experiments corresponds to a 350×350-pixel subset of the sector labeled as f970619t01p02_r02_sc03.a.rfl in the online data. The scene comprises 224 spectral bands between 0.4 and $2.5\ \mu\mathrm{m}$, with nominal spectral resolution of 10 nm. Prior to the analysis, bands 1–2, 105–115, 150–170, and 223–224 were removed due to water absorption and low SNR in those bands, leaving a total of 188 spectral bands for a total size of about 50 MB. The Cuprite site is well understood mineralogically and has several exposed minerals of interest, all included in the USGS library. In our experiments, we use spectra obtained from this library in order to substantiate the quality of the endmembers derived by MVSA and compare them with those produced by other algorithms. For illustrative purposes, Fig. 5 shows a mineral map produced in 1995 by USGS, in which the Tricorder 3.3 software product was used to map different minerals present in the Cuprite mining district.[11]

Specifically, in this experiment, we used the spectral signatures of the minerals: alunite (GDS84), buddingtonite (GDS85), calcite (WS272), kaolinite (KGa-1), and muscovite (GDS107) that are present in the Cuprite data set and computed the SAD between the reflectance spectra for these minerals and the endmembers extracted by MVSA, MVES and N-FINDR. All algorithms have been set to extract $p = 15$ endmembers from the data, which is a reasonable estimate according to previous studies. The best matching extracted endmembers with regards to the reference USGS mineral signatures in terms of SAD are selected for comparison. The choice of these mineral signatures was made based on recent works conducted with the AVIRIS Cuprite data [40], [41]. In the MVSA experiments, we fix the

[9]Available: http://speclab.cr.usgs.gov/spectral.lib06

[10]Available: http://aviris.jpl.nasa.gov/html/aviris.freedata.html

[11]Available: http://speclab.cr.usgs.gov/cuprite95.tgif.2.2um_map.gif

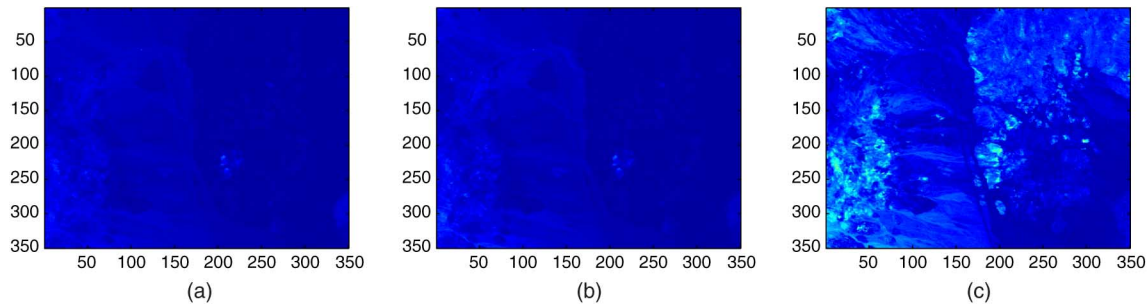

Fig. 6. MSE obtained in each pixel of the AVIRIS Cuprite scene after approximating the pixel in the original image with a linear combination formed using the endmembers derived by each considered method. The total MSE in each case is reported in the parentheses. (a) MVSA (0.2172). (b) MVES (0.2265). (c) N-FINDR (0.6257).

TABLE III
PROCESSING TIMES, MPI TIMES, I/O TIMES, AND SPEEDUPS WITH REGARDS TO: A) 1 GPU, B) A MULTI-THREADED, AND C) A SINGLE-THREADED MKL IMPLEMENTATION OF MVSA THAT EXPLOITS THE 8 CORES AVAILABLE IN THE CONSIDERED SYSTEM

Number of GPUs	Application time (s)	Mean MPI time (s)	I/O time (s)	Mean MPI (%)	Mean I/O (%)	Speedup with regards to 1 GPU	Speedup with regards to 8 CPU cores	Speedup with regards to 1 CPU core
1	9.91	0.007	0.21	0.07	2	1	5.2	8.6
2	5.58	0.06	0.18	1.1	3	1.7	9.2	15.1
3	4.14	0.1	0.21	2.63	5	2.3	12.3	20
4	3.36	0.076	0.19	2.26	6	2.9	15.1	24.7
5	2.76	0.1	0.2	3.3	7	3.4	18.2	29.7
6	2.52	0.2	0.2	8	8	3.6	19.1	31.3
7	2.34	0.13	0.21	5	9	4	21.1	34.4

The Multi-threaded implementation took 52 s and the single-threaded implementation took 85 s to complete 150 iterations of the interior-point algorithm.

TABLE IV
MPI TIMES IN EACH OF THE NODES FOR 150 ITERATIONS OF THE INTERIOR-POINT ALGORITHM USING ALL SEVEN IBM BLADES

Task ID rank	Application time (s)	MPI time (s)	MPI (%)
0	2.34	0.128	5.47
1	2.34	0.112	4.78
2	2.34	0.095	3.04
3	2.34	0.09	3.84
4	2.34	0.162	6.91
5	2.34	0.157	6.68
6	2.34	0.157	6.72

number of iterations of the interior point algorithm empirically to 150. Table II shows the SAD between the endmembers identified by MVSA, MVES, and N-FINDR and the reference spectral signatures for the selected minerals. The SAD comparison reveals that the performance of the algorithms tested depends on the reference mineral considered. Finally, Fig. 6 shows the MSE obtained in each pixel after approximating the original image with a linear combination formed using the endmembers derived by each method and the fractional abundances estimated by fully constrained linear spectral unmixing [42]. As shown by Fig. 6, the MVSA performs better than both MVES and N-FINDR according to this metric.

B. Evaluation of Parallel Performance

In this subsection, we evaluate the computational performance of our proposed optimizations for MVSA. Our test environment consists of a parallel system made up of seven IBM blades, each equipped with an NVIDIA Tesla M2070Q GPU and two Intel Xeon processors having four cores each (hyper-threading was disabled). Each M2070Q GPU features 448 cores, double precision floating point performance (peak) of 515 Gflops, single precision floating point performance (peak) of 1.03 Tflops, total dedicated memory of 6 GB (GDDR5) and maximum power consumption of 225W TDP. All blades are connected with Infiniband 40 Gbit/s network cards.

For comparative purposes, in addition to the proposed optimizations, we have developed a multithreaded CPU version of MVSA using the MKL library (thus taking advantage of the eight cores of the system) and used this version as or reference in order to calculate the acceleration factors (speedups). In this version, all level-3 BLAS functions like matrix multiplication are highly optimized, taking advantage of the processor caches, while all level-2 BLAS functions are also multithreaded. It should also be noted that our multi-GPU implementation allows any kind of accelerator to be used, including recent platforms as the Intel Xeon Phi. In this work, we have chosen GPU accelerators mainly because we have seen empirically observed that the execution time scale linearly among multiple GPUs. Moreover, the new NVidia GPUs allow multiple MPI processes (currently, up to 32) running on different queues with the

TABLE V
PROCESSING TIMES, MPI TIMES, I/O TIMES, AND SPEEDUP FOR 50, 100, AND 150 ITERATIONS

Number of iterations	Application time (s)	Mean MPI time (s)	I/O time (s)	Mean MPI%	Mean I/O%	Time for 1 GPU (s)	1 CPU core time (s)	8 CPU core time (s)	Speedup w.r.t. 1 GPU	Speedup w.r.t. 8 CPU cores	Speedup w.r.t. 1 CPU core
50	0.774	0.05	0.07	6.6	9	3.3	28.6	17.9	4	21.7	34.7
100	1.55	0.08	0.15	5.3	9.6	6.6	56.8	35.1	4	21.5	34.8
150	2.34	0.13	0.21	5	9	9.9	85	52	4	21.1	34.4

Hyper-Q technology,[12] thus significantly increasing the utilization of the GPU.

The hyperspectral image used in our experiments is the AVIRIS Cuprite scene described in the previous subsection, with a size of approximately 50 MB, and the MVSA algorithm was run using the same conditions described in the previous experiment. It should be noted that, as mentioned before, there is no parallel implementation of MVES available in the literature while the GPU implementation of N-FINDR described in [20] provides much higher processing times than the ones reported for MVSA (there is no multi-GPU implementation of N-FINDR currently available); hence we only report the computational times measured for our proposed optimizations of MVSA. Specifically, the multithreaded implementation of MVSA (developed in MKL to exploit the 8 cores available in the system) took 52 s to process the AVIRIS Cuprite scene, while the single-threaded version took 85 s.

In our first experiment, we measured the execution time and the I/O timings from host to device and from device to host, as well as the MPI time for all collective operations as measured from `mpiP`,[13] a lightweight profiler for MPI. The latter reports accurately the application time and the MPI time. The I/O time is measured with the UNIX C function `gettimeofday`, which is very accurate. The CPU version of the algorithm is also measured with the `gettimeofday` C function. Since the interior point algorithm was made distributed and is the main computationally demanding task of MVSA, our first evaluation will be carried out by running 150 iterations of the algorithm. Table III displays the aforementioned timings. From Table III, we can make the following observations.

1) The MPI time is below 10% of the whole interior-point execution, and we can even observe that for 7 GPUs, it is only 5%. This indicates that our algorithm manages to minimize the communication time as much as possible.
2) The I/O time remains constant and below 10% of the whole application time, which means that over 90% of the time the GPUs are performing computational work.
3) For 7 GPUs, we manage to achieve a speedup of 4x over one GPU and more than 20x over the multithreaded CPU version. This is a significant achievement, as the CPU implementation used as reference efficiently exploits the 8 cores available in the system through a highly optimized MKL implementation.

Another important consideration at this point is how the computation is distributed across the nodes, and whether the MPI usage is evenly distributed between the nodes. When describing our multi-GPU implementation, we have claimed that this is indeed the case. For illustrative purposes, Table IV shows this distribution. It can be observed that the MPI timings are close to the mean value (0.13 s) which means that the MPI workload is evenly distributed across the nodes.

In a last experiment, we evaluated the sensitivity of our multi-GPU implementation of the interior-point algorithm to the number of iterations taken. As indicated by Table V, we considered the timings for 50, 100, and 150 iterations. The results reported on Table V indicate that the MPI%, I/O%, and speedups measured are insensitive to the number of iterations, i.e., the MPI% and I/0% and speedup remain roughly the same across the number of iterations. We used all of the 7 IBM blades for this experiment.

From the above performance evaluations, we have shown that our multi-GPU is optimal following all good principles and guidelines in distributed programming and we were successful in accelerating MVSA dramatically. This represents a very significant improvement with regards to previous implementations of MVSA, in which it was already very difficult to process a full hyperspectral scene due to memory requirements and the algorithm required much higher computation time to finish, e.g., a Matlab implementation of MVSA was reported in [43] to take more than 25 000 s (416 min) to process the considered AVIRIS Cuprite scene in a desktop PC with Intel Core I7 920 CPU at 2.67 GHz CPU at 2.65 HHz with 4 GB of RAM memory. Although the CPU platforms are not comparable, the serial (52 s), GPU (9.91 s), and multi-GPU (2.47 s) processing times reported in this paper make the MVSA algorithm a near real-time one and readily available for fast processing of large hyperpectral images in affordable computing architectures. These results are expected to enhance the popularity of MVSA in the hyperspectral imaging community, as the endmember identification accuracy results obtained by this algorithm are equal or slightly superior than those obtained by other state-of-the-art methods.

VII. CONCLUSION AND FUTURE RESEARCH LINES

In this work, we have presented several optimizations for the MVSA algorithm, a popular strategy for endmember identification in the difficult case in which it is assumed that the original hyperspectral scene does not contain any spectrally pure signatures. Specifically, we have developed three optimizations. In the first one, we used algebra operations in order to reduce the very high memory requirements of the algorithm. In the second one, we have taken advantage of graphics processing units (GPUs) to effectively solve (in parallel) the quadratic optimization problem involved in the computation of MVSA. For this purpose, we used

[12]Available: http://www.nvidia.com/object/nvidia-kepler.html

[13]Available: http://mpip.sourceforge.net/

an optimized interior point method. In the third optimization, we extended the single GPU implementation to a multi-GPU one, developing a hybrid strategy that distributes the computation while taking advantage of GPU accelerators at each node. Our experimental results, conducted using both synthetic and real hyperspectral scenes, indicate that the proposed optimizations provide significant improvements in the computation time (and also in the memory requirements) of the MVSA algorithm. Although the reported processing times represent a very significant improvement with regards to previously available implementations of the algorithm, in future work we will continue exploring additional strategies for optimization, such as splitting the original hyperspectral image into subimages and applying the multi-GPU impelmentation to each of them. Other high-performance computing architectures such as digital signal processors (DSPs) or FPGAs will be also explored due to their capacity to be used as onboard processing modules in airborne and (particularly) spaceborne Earth observation missions.

Acknowledgment

The authors gratefully acknowledge the Associate Editor and the Anonymous Reviewers for their careful assessment and suggestions for improving our manuscript, which have been really helpful in order to enhance its presentation and technical quality.

References

[1] R. O. Green *et al.*, "Imaging spectroscopy and the airborne visible/infrared imaging spectrometer (AVIRIS)," *Remote Sens. Environ.*, vol. 65, no. 3, pp. 227–248, 1998.

[2] G. Shaw and D. Manolakis, "Signal processing for hyperspectral image exploitation," *IEEE Signal Process. Mag.*, vol. 19, no. 1, pp. 12–16, Jan. 2002.

[3] D. Landgrebe, "Hyperspectral image data analysis," *IEEE Signal Process. Mag.*, vol. 19, no. 1, pp. 17–28, Jan. 2002.

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

[5] A. Plaza *et al.*, "Recent advances in techniques for hyperspectral image processing," *Remote Sens. Environ.*, vol. 113, pp. S110–S122, 2009.

[6] J. 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.

[7] S. Chandrasekhar, *Radiative Transfer*. New York, NY, USA: Dover, 1960.

[8] H. Qu, B. Huang, J. Zhang, and Y. Zhang, "An improved maximum simplex volume algorithm to unmixing hyperspectral data," in *Proc. SPIE*, 2013, vol. 8895, pp. 889507–889507-7. doi: 10.1117/12.2034759.

[9] J. Li and J. Bioucas-Dias, "Minimum volume simplex analysis: a fast algorithm to unmix hyperspectral data," in *Proc. IEEE Int. Geosci. Remote Sens. Symp.*, 2008, vol. 3, pp. 250–253.

[10] J. H. Jung and D. P. OLeary, "Implementing an interior point method for linear programs on a CPU-GPU system," *Electron. Trans. Numer. Anal.*, vol. 28, pp. 174–189, 2008.

[11] CULA. (2014). *GPU-Accelerated Linear Algebra* [Online]. Available: http://www.culatools.com/

[12] A. Plaza, Q. Du, Y.-L. Chang, and R. L. King, "Foreword to the special issue on high performance computing in earth observation and remote sensing," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 4, no. 3, pp. 503–507, Sep. 2011.

[13] A. Plaza, J. Plaza, A. Paz, and S. Sánchez, "Parallel hyperspectral image and signal processing [applications corner]," *IEEE Signal Process. Mag.*, vol. 28, no. 3, pp. 119–126, May 2011.

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

[15] C. Lee, S. Gasster, A. Plaza, C.-I. Chang, and B. Huang, "Recent developments in high performance computing for remote sensing: A review," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 4, no. 3, pp. 508–527, Sep. 2011.

[16] A. Plaza, "Special issue on architectures and techniques for real-time processing of remotely sensed images," *J. Real-Time Image Proc.*, vol. 4, no. 3, pp. 191–193, 2009.

[17] A. Plaza and C.-I. Chang, *High Performance Computing in Remote Sensing*. New York, NY, USA: Taylor & Francis, 2007.

[18] J. Nocedal and S. Wright, *Numerical Optimization, Series in Operations Research and Financial Engineering*. New York, NY, USA: Springer-Verlag, 2006.

[19] M. Winter, "N-FINDR: An algorithm for fast autonomous spectral endmember determination in hyperspectral data," in *Proc. SPIE*, 1999, vol. 3753, pp. 266–270.

[20] S. Sanchez, R. Ramalho, L. Sousa, and A. Plaza, "Real-time implementation of remotely sensed hyperspectral image unmixing on gpus," *J. Real-Time Image Process.*, pp. 1–15, Sep. 2012 [Online]. Available: http://dx.doi.org/10.1007/s11554-012-0269-2

[21] Z. Wu *et al.*, "Fast endmember extraction for massive hyperspectral sensor data on GPUs," *Int. J. Distrib. Sensor Netw.*, vol. 2013, no. 217180, pp. 1–7, 2013, doi: 10.1155/2013/217180.

[22] J. Nascimento and J. 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.

[23] J. M. R. Alves, J. M. P. Nascimento, J. M. Bioucas-Dias, V. Silva, and A. Plaza, "Parallel implementation of vertex component analysis for hyperspectral endmember extraction," in *Proc. IGARSS*, 2012, pp. 4078–4081.

[24] B. Huang, A. Plaza, and Z. Wu, "Acceleration of vertex component analysis for spectral unmixing with CUDA," in *Proc. SPIE*, 2013, vol. 8895, pp. 889 509–889 509–10, doi: 10.1117/12.2031527.

[25] S. Moussaoui, C. Carteret, D. Brie, and A. Mohammad-Djafari, "Bayesian analysis of spectral mixture data using Markov chain Monte Carlo methods," *Chemom. Intell. Lab. Syst.*, vol. 81, no. 2, pp. 137–148, 2006.

[26] 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.

[27] J. Setoain, M. Prieto, C. Tenllado, A. Plaza, and F. Tirado, "Parallel morphological endmember extraction using commodity graphics hardware," *IEEE Geosci. Remote Sens. Lett.*, vol. 4, no. 3, pp. 441–445, Jul. 2007.

[28] M.-D. Iordache, J. Bioucas-Dias, and A. Plaza, "Sparse unmixing of hyperspectral data," *IEEE Trans. Geosci. Remote Sens.*, vol. 49, no. 6, pp. 2014–2039, Jun. 2011.

[29] J. M. R. Alves, J. Nascimento, J. M. Bioucas-Dias, A. Plaza, and V. Silva, "Parallel sparse unmixing of hyperspectral data," in *Proc. IEEE Int. Geosci. Remote Sens. Symp.*, 2013, pp. 1446–1449.

[30] L. Miao and H. Qi, "Endmember extraction from highly mixed data using minimum volume constrained nonnegative matrix factorization," *IEEE Trans. Geosci. Remote Sens.*, vol. 45, no. 3, pp. 765–777, Mar. 2007.

[31] A. Ambikapathi, T.-H. Chan, W.-K. Ma, and C.-Y. Chi, "Chance-constrained robust minimum-volume enclosing simplex algorithm for hyperspectral unmixing," *IEEE Trans. Geosci. Remote Sens.*, vol. 49, no. 11, pp. 4194–4209, Nov. 2011.

[32] L. Scharf, *Statistical Signal Processing, Detection Estimation and Time Series Analysis*. Reading, MA, USA: Addison-Wesley, 1991.

[33] K. Lange, D. Hunter, and I. Yang, "Optimization transfer using surrogate objective functions," *J. Comput. Graphical Statist.*, vol. 9, pp. 1–59, 2000.

[34] I. Griva, S. G. Nash, and A. Sofer, *Linear and Nonlinear Optimization*, 2nd ed. Philadelphia, PA, USA: SIAM, 2008.

[35] P. Pacheco, *An Introduction to Parallel Programming*, 1st ed. San Mateo, CA, USA: Morgan Kaufmann, 2011.

[36] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-On Approach*, 2nd ed. San Mateo, CA, USA: Morgan Kaufmann, 2012.

[37] A. Plaza, J. Plaza, and D. Valencia, "Impact of platform heterogeneity on the design of parallel algorithms for morphological processing of high-dimensional image data," *J. Supercomput.*, vol. 40, pp. 81–107, 2007.

[38] J. Nascimento and J. 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.

[39] R. N. Clark *et al.*, "USGS digital spectral library splib06a," in *U.S. Geological Survey, Digital Data Series 231*. U.S. Geological Survey, a bureau of the U.S. Department of the Interior [Online]. Available: http://speclab.cr.usgs.gov/spectral.lib06, accessed on Nov. 25, 2013.

[40] 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.

[41] 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.
[42] D. Heinz and C.-I. Chang, "Fully constrained least squares linear mixture analysis for material quantification in hyperspectral imagery," *IEEE Trans. Geosci. Remote Sens.*, vol. 39, no. 3, pp. 529–545, Mar. 2001.
[43] 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.

Alexander Agathos received the Ph.D. degree in computer science from the University of the Aegean, Lesvos, Greece.

He worked as a Post-Doctoral Researcher with Tel Aviv University, Tel Aviv, Israel. He is a Researcher with the Computer Science Department, West University of Romania, Timişoara, Romania and is currently working on the EU funded project HOST under Prof. Dana Petcu's coordination. His research interests include computer graphics, image processing, as well as parallel and distributed programming with emphasis on developing algorithms for accelerators such as GPUs and Xeon Phi. He has published several papers on a plethora of subjects involving computer graphics, image processing, and 3-D computer vision.

Jun Li received the B.S. degree in geographic information systems from Hunan Normal University, Changsha, China, in 2004, the M.E. degree in remote sensing from Peking University, Beijing, China, in 2007, and the Ph.D. degree in electrical engineering from the Instituto de Telecomunicações, Instituto Superior Técnico (IST), Universidade Técnica de Lisboa, Lisbon, Portugal, in 2011.

From 2007 to 2011, she was a Marie Curie Research Fellow with the Departamento de Engenharia Electrotécnica e de Computadores, Coimbra, Portugal, and the Instituto de Telecomunicações, IST, Universidade Técnica de Lisboa, Lisbon, Portugal, in the framework of the European Doctorate for Signal Processing (SIGNAL). She has also been actively involved in the Hyperspectral Imaging Network, a Marie Curie Research Training Network involving 15 partners in 12 countries and intended to foster research, training, and cooperation on hyperspectral imaging at the European level. Since 2011, she has been a Postdoctoral Researcher with the Hyperspectral Computing Laboratory, Department of Technology of Computers and Communications, Escuela Politécnica, University of Extremadura, Cáceres, Spain. Her research interests include hyperspectral image classification and segmentation, spectral unmixing, signal processing, and remote sensing.

Dr. Li received the 2012 Best Reviewer Award of the IEEE JOURNAL OF SELECTED TOPICS IN APPLIED EARTH OBSERVATION AND REMOTE SENSING. She has been a Reviewer of several journals, including the IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING, the IEEE GEOSCIENCE AND REMOTE SENSING LETTERS PATTERN RECOGNITION, OPTICAL ENGINEERING, *Journal of Applied Remote Sensing*, and *Inverse Problems and Imaging*.

Dana Petcu is a Professor with Computer Science Department, West University of Timisoara, Timişoara, Romania, and Director of its private research spin-off Institute e-Austria Timisoara, Romania. Her research interests include parallel and distributed computing.

She has authored more than 250 peer-reviewed articles. She is a Chief Editor of the international journal *Scalable Computing: Practice and Experiences*. She is leading and has lead several research and development project related to HPC and cloud computing funded by European and national agencies. She received an international award for women in science and an IBM award, and is a nominated expert in several European forums related to e-infrastructures and ICT.

Antonio Plaza (M'05–SM'07) is an Associate Professor (with accreditation for Full Professor) with the Department of Technology of Computers and Communications, University of Extremadura, Badajoz, Spain, where he is the Head of the Hyperspectral Computing Laboratory (HyperComp). He was the Coordinator of the Hyperspectral Imaging Network, a European project with total funding of 2.8 MEuro. He authored more than 400 publications, including more than 110 JCR journal papers (66 in IEEE journals), 20 book chapters, and over 240 peer-reviewed conference proceeding papers (94 in IEEE conferences). He has guest edited seven special issues on JCR journals (three in IEEE journals).

Dr. Plaza 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 has served as Associate Editor in 2007–2012. He has been a Chair for the IEEE Workshop on Hyperspectral Image and Signal Processing: Evolution in Remote Sensing (2011). He is also an Associate Editor for IEEE ACCESS and the IEEE GEOSCIENCE AND REMOTE SENSING MAGAZINE and was a member of the Editorial Board of the IEEE GEOSCIENCE AND REMOTE SENSING NEWSLETTER (2011–2012) and a member of the steering committee of the IEEE JOURNAL OF SELECTED TOPICS IN APPLIED EARTH OBSERVATIONS AND REMOTE SENSING (2012). 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 President of the Spanish Chapter of IEEE GRSS (since November 2012). He is currently serving as the Editor-in-Chief of the IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING journal (since January 2013). Additional information: http://www.umbc.edu/rssipl/people/aplaza.