GPU Implementation of Large-Scale Abundance Estimation with Constraint Linear Spectral Mixture Model for Hyperspectral Unmixing

Lu Li [1], Wei Li [2,*]

[1] *School of Automation, Beijing Information Science and Technology University, Beijing, 100192 China (e-mail: 1141017448@qq.com)*

[2] *Department of Electrical and Computer Engineering, Mississippi State University, MS 39762 USA (e-mail: du@ece.msstate.edu)*

[*] *Corresponding author; Phone Number: +86-010-64413467; Fax Number: +86-010-64434726*

Abstract

Linear spectral mixture Model (LMM) is the most commonly used model in hyperspectral unmixing. There are two constraints introduced into LMM, i.e.(1) abundance non-negetive constraint(ANC) and (2) abundance sum to one constraint (ASC). Non-negetive least squares (NNLS) and fully constraint least squares (FCLS) algorithms can be used for abundance estimation with ANC, and both ANC and ASC respectively. However, due to the burden of the frequent matrix inversion, both NNLS and FCLS are costly and difficult to parallelize for real-time large-scale hyperspectral image (HSI) processing. Therefore, in this paper, a parallelized sequential least squares algorithm is proposed for large-scale abundance estimation. In proposed algorithm, partitioned matrix inversion theory is applied to avoid frequent matrix inversion in NNLS or FCLS and an improved iterative process is also introduced to further decrease time-consuming. Furthermore, NVIDIA Compute Unified Device Architecture(CUDA) stream with device overlap is employed for GPU parallel implementation to reach the purpose of real-time processing. In our experiments, the real HSI datasets were tested to verify the effect and efficiency of GPU parallel implementation. Experimental results demonstrate that our implementation is already much more efficient than NNLS, fast NNLS and FCLS in traditional version such as Matlab or Visual Studio with the same accuracy. And if the GPU hardware conditions, It is absolutely possible to estimate abundance with LMM in real-time for a large-scale HSI if the computing power of GPUs hardware satisfy the demands.

Keywords: Non-negetive Least Square, GPU, Hyperspectral Unmixing.

1. Introduction

Hyperspectral imagery (HSI) has been widely applied in many applications of earth observation and geosciences. Among them, hyperspectral unmixing is an important task for subpixel-level hyperspectral data exploitation. The reason is that when the HSI spatial resolution is generally not enough to separate different pure spectral signatures (called endmembers), some of them can be extracted in the same pixel with hyperspectral unmixing technique. Therefore, hyperspectral unmixing could improve the performance of HSI classification and target detection. In generally, hyperspectral unmixing contains two main steps: 1)endmembers extraction and 2) abundances estimation. Endmembers extraction is to determine spectral signatures of the pure constituent materials in the scene, and abundances estimation provides the fractional corresponding amount of each pure constituent at each image pixel. There are two popular mixture models in both steps above, i.e. linear mixture model (LMM) and nonlinear mixture models (NLMM). Due to simple and unsupervised, LMM is the most widely used. In LMM, the *ith* pixel vector $\mathbf{y_i} \in \mathrm{R}^L$ can be represented as

$$\mathbf{y}_i = \mathbf{E}\mathbf{a}_i + \mathbf{n}_i = \sum_{k=1}^{p} \mathbf{e}_k a_{ik} + \mathbf{n}_i \tag{1}$$

where L is the number of HSI bands, $\mathbf{E} \in \mathrm{R}^{L\times p}$ is the endmembers matrix whose *kth* column corresponds to the *kth* endmember $\mathbf{e_k}$, p is the number of total endmemmbers in the scene, $\mathbf{a_i} \in \mathrm{R}^{\mathbf{p}}$ is abundance vector of the *ith* pixel, whose *kth* component a_ik represents the fraction of the *kth* endmember in the *ith* pixel, and $\mathbf{n_i}$ is an additive noise vector. There are two priors introduced into Eq. 1 to satisfy two constraints, i.e. (1)the abundance nonnegative constraint

(ANC) $\forall i, k, \quad a_{ik} \geq 0$ and (2) the abundance sum-to-one constraint (ASC) $\forall i, \sum_{k=1}^{p} a_{ik} = 1$. Famous non-negetive least squares (NNLS) algorithm ([1]) can be used for abundance estimation Eq. 1 with ANC,meanwhile, fully constraint least squares (FCLS) algorithms is applied with both ANC and ASC respectively. Both NNLS and FCLS are the special case of the active-set quadratic programming algorithm. Both they need to update the active-set in each iteration by removing or adding an endmember, and then repeat to estimate abundance by least squares. In [2], a fast-NNLS is proposed to improve efficiency by pre-define some matrix and in [3], an improved iterative process decrease the number of iteration and only removing endmember would be conducted, but least squares is inevitable in both of them , which requires time-consuming matrix inversion work, not to mention a lot of repetition in each iteration just for unmixing one pixel in HSI. Taking into account the fact that the number of pixels in HSI usually ranges from 100,000 to 100,000,000, hyperspectral unmixing need up to tens of billions of float operations in the execution process, which cause that it is very difficult for (near) real-time processing.

Taking the well-known Airborne Visible Infrared Imaging Spectrometer(AVIRIS)[4] sensor as an example, a HSI cube with size of 614 times 512 pixels and 224 spectral bands should be processed in about 5 s in order to meet real-time processing performance [5].But the high time cost cannot satisfy the demand of (near) real-time large-scale hyperspectral unmixing based on CPU serial at present. Unfortunately, (near) real-time hyperspectral unmixing is a highly desired goal in many applications, such as biological threat detection[6], monitoring of chemical contamination[7], and wildfire tracking[8]. Especially, with the development that the advanced hyperspectral sensors have increasingly high spatial, spectral, and temporal resolutions, it is hence mandatory to follow new highly parallel unmixing solutions that can take full advantage of the characteristics of nowadays high-performance computing (HPC) architectures. In order to improve the speed of hyperspectral unmixing, some researcher have introduced several high-performance computing technology and varieties of parallel systems, such as graphics processing units (GPU) [9][10], multi-core digital signal processor (DSP)[11][12], field programmable gate arrays (FPGAs)[13][14], even supercomputers and small clusters of computers[15][16]. Despite the growing interest in hardware to accelerate computation of hyperspectral unmixing research[17][18][19][20], it is still possible to optimize iterative process and some implementations to achieve a HSI processing in real-time. In this paper, we address on (near) real-time abundance estimation for three contributions, i.e.(1) rearrangement iterative process both in NNLS and FCLS; (2) matrix inversion could be updated adaptively in each iteration according to partitioned matrix inversion theory; and (3) GPU implementation. In the whole GPU implementation, each thread on the device needs to do the following three steps. Firstly, all endmembers are chosen, and abundances are estimated by unconstraint least squares. Secondly, algorithm would determine whether the stopping criterion is met, and if not, select a least relevant endmember to be removed. Thirdly, abundances are re-estimated. In the 3rd step, partitioned matrix inversion theory is applied to avoid frequent matrix inversion in NNLS or FCLS. The proposed method can not only save the time-consuming of matrix inversion but also suit for thread programming under NVIDIA compute unified device architecture (CUDA). The reason why the GPU is chosen is that a large number of parallel multi-threaded tasks exist in hyperspectral unmixing and CUDA makes the GPU implementation much easier. The proposed method can still be easier transplanted into other parallel systems.

The remainder of the paper is organized as follows. Section II describes the preliminary including NNLS, partitioned matrix inversion theory and FCLS algorithms. Section III introduces the proposed methods and the GPU implementation of the large-scale hyperspectral unmixing framework. Section III provides experimental results and discussion. Finally, conclusions are drawn in Section IV.

2. Preliminary

2.1. Non-Negative Least squares

Problem NNLS is defined by Eq. 1 with ANC. The classic solution of NNLS is to transform the original problem into the quadratic programming problem and then solve it by active set method. The details of NNLS algorithm can be seen in Algorithm. 1. In Algorithm. 1, there are two loop,i.e. a main loop from line 5 to line 18 and an inner loop from line 7 to line 16. In the main loop, dual vector $\mathbf{w}$ is derived at line 4 and 17 and an index is selected to be removed from set $\mathbf{Z}$ to set $\mathbf{P}$ at line 6. In the inner loop, tentative solution vector $\mathbf{z}$ is computed at line 8 and an index is selected to be removed from set $\mathbf{P}$ to set $\mathbf{Z}$ at line 10 and line 12. Linear stretching at Line 11 ensures the constraint that the coefficients of $\mathbf{z}$ are non-negative.

In Algorithm. 1, the solution of least squares at line 8 requires a positive-define matrix inversion:

$$\mathbf{z_P} = (\mathbf{E_P^T E_P})^{-\mathbf{1}} \mathbf{E_P^T y} \tag{2}$$

If n denotes the number of indices in set $\mathbf{P}$,complexity of matrix inversion is O(n^3). The number of iterations in the inner loop must not exceed $n-1$, and the number of iterations in the main loop is about $p/2$. Therefore, the frequent matrix inversion makes NNLS exhausted. In fast NNLS [2], $\mathbf{E^T E}$ and $\mathbf{E^T y}$ is pre-defined before main loop beginning and the time-consuming of calculation on $\mathbf{E_P^T E_P}$ and $\mathbf{E_P^T y}$ would be partially decreased, but matrix inversion is still inevitable.

2.2. Partitioned Matrix Inversion

As mentioned above, in main loop, removing an index from set $\mathbf{Z}$ to set $\mathbf{P}$ means endmembers matrix would add an endmember. On the contrary, in inner loop, removing an index from set $\mathbf{Z}$ to set $\mathbf{P}$ means delete an endmember. As long as endmembers matrix changes, the least squares will be re-computed. Let $\mathbf{E}$ and $\mathbf{E}'$ denote original and changed endmember matrix respectively, the relationship between $(\mathbf{E}^T\mathbf{E})^{-1}$ and $(\mathbf{E}'^T\mathbf{E}')^{-1}$ can be derived by partitioned

Algorithm 1 Non-Negative Least squares

```
1:  **Input** : Endmembers matrix $\mathbf{E} \in \mathbb{R}^{L\times p}$, Pixel vector $\mathbf{y} \in \mathbb{R}^{L}$.
2:  **Output** : Abundance vector $\mathbf{a} \in \mathbb{R}^{p}$.
3:  **Initializiation** : Passive set $\mathbf{P} = \mathbf{NULL}$,and Active Set $\mathbf{Z} = \{1, 2, \cdots, \mathrm{p}\}$,
    and $\mathbf{a} = \mathbf{0}$.
4:  Compute the dual vector $\mathbf{w} = \mathbf{E^T}(\mathbf{y} - \mathbf{Ea})$
    **Main-loop beginning**
5:  **while** $\mathbf{Z} \neq \mathbf{NULL}$ && $\exists \mathbf{w_j} \geq \mathbf{0}, \mathbf{j} \in \mathbf{Z}$ **do**
6:    Find an index $t \in \mathbf{Z}$ **such that** $\mathbf{w_t} = \mathbf{max}\{\mathbf{w_j} : \mathbf{j} \in \mathbf{Z}\}$.Move the index $t$
      from the set **Z to set P**
      **inner-loop beginning**
7:    **repeat**
8:      Let $\mathbf{E_P}$ denote the matrix defined by:
        Column $j$ of $\mathbf{E_P} := \mathit{column} \quad P[j] \quad \mathit{of} \quad \mathbf{E}$
        Compute the p-vector $\mathbf{z}$ as a solution of the least squares problem
        $\mathbf{E_P z_P} \cong \mathbf{y}$. Note that only the components $\mathbf{z_j}, \mathbf{j} \in \mathbf{P}$, are determined
        by this problem. Define $\mathbf{z_j} := \mathbf{0}$ for $j \in \mathbf{Z}$.
9:      **if** $\exists \mathbf{z_j} < 0, \mathbf{j} \in \mathbf{P}$ **then**
10:       Find an index $q \in \mathbf{P}$ such that $\mathbf{a_q}/(\mathbf{a_q} - \mathbf{z_q}) = \mathbf{min}\{\mathbf{a_j}/(\mathbf{a_j} - \mathbf{z_j}) :$
          $\mathbf{z_j} \leq \mathbf{0}, \mathbf{j} \in \mathbf{P}\}$.
11:       Set $\alpha :=\mathbf{a_q}/(\mathbf{a_q} - \mathbf{z_q})$. and set $\mathbf{a} := \mathbf{a} + \alpha(\mathbf{z} - \mathbf{a})$
12:       Move an index q from set **P** to set **Z**
13:     **else**
14:       Set $\mathbf{a} := \mathbf{z}$
15:     **end if**
16:   **until** $\mathbf{z_j} \geq \mathbf{0}, \mathbf{j} \in \mathbf{P}$
      **inner-loop ending**
17:   Compute the dual vector $\mathbf{w} = \mathbf{E^T}(\mathbf{y} - \mathbf{Ea})$
18: **end while**
    **Main-loop ending**
```

Matrix Inversion theory as below.

(1) **Adding an endmember**. As in Fig. 1(a)shown, original endmembers matrix has n column and L rows. Each column vector $\mathbf{e_i}$ of original endmembers matrix denotes an endmember. Supposed a new endmember $\mathbf{e_{n+1}}$ is added to the last column of the endmembers matrix $\mathbf{E}$ to reform $\mathbf{E}'$ (can be seen in Fig. 1(b)), then $\mathbf{E'^T E'}$ is defined by,

$$\mathbf{E}'^T\mathbf{E}' = \begin{bmatrix} \mathbf{E}^T\mathbf{E} & \mathbf{A} \\ \mathbf{A}^T & b \end{bmatrix} \tag{3}$$

where $\mathbf{A} = e_{n+1}^T\mathbf{E}$ and $b = e_{n+1}^T e_{n+1}$ respectively. If $(\mathbf{E}^T\mathbf{E})^{-1}$ is known, then $(\mathbf{E}'^T\mathbf{E}')^{-1}$ could be obtained by Eq. 4,

$$(\mathbf{E}'^T\mathbf{E}')^{-1} = \begin{bmatrix} \mathbf{E}^T\mathbf{E} & \mathbf{A} \\ \mathbf{A}^T & b \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{P}_1 & \mathbf{P}_2 \\ \mathbf{P}_2^T & P_3 \end{bmatrix} \tag{4}$$

where $P_3 = (b - \mathbf{A}^T(\mathbf{E}^T\mathbf{E})^{-1}\mathbf{A})^{-1}$, $\mathbf{P}_2 = -P_3(\mathbf{E}^T\mathbf{E})^{-1}\mathbf{A}$, and $\mathbf{P}_1 = (\mathbf{E}^T\mathbf{E})^{-1} + P_3(\mathbf{E}^T\mathbf{E})^{-1}\mathbf{A}\mathbf{A}^T(\mathbf{E}^T\mathbf{E})^{-1} = (\mathbf{E}^T\mathbf{E})^{-1} - (\mathbf{P}_2\mathbf{P}_2^T)/P_3$.

(2) **Deleting an endmember.** As in Fig. 1(c)shown, supposed the endmember $\mathbf{e_i}$ would be removed from endmembers matrix $\mathbf{E}$, then $\mathbf{E'^T E'}$ could be obtained by removing the i-th column and the i-th row from $\mathbf{E^T E}$ (see middle of Fig. 1(c)). Supposed that positive-define symmetric matrix$(\mathbf{E^T E})^{\mathbf{-1}}$ is known and define by,

$$(\mathbf{E^T E})^{\mathbf{-1}} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1i} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2i} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{ii} & \cdots & a_{in} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{ni} & \cdots & a_{nn} \end{bmatrix} \tag{5}$$

Let $\mathbf{P}_1 = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1(i-1)} & a_{1(i+1)} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2(i-1)} & a_{2(i+1)} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{(i-1)1} & a_{(i-1)2} & \cdots & a_{(i-1)(i-1)} & a_{(i-1)(i+1)} & \cdots & a_{(i-1)n} \\ a_{(i+1)1} & a_{(i+1)2} & \cdots & a_{(i+1)(i-1)} & a_{(i+1)(i+1)} & \cdots & a_{(i+1)n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{n(i-1)} & a_{n(i+1)} & \cdots & a_{nn} \end{bmatrix}$,

$\mathbf{P}_2 = \begin{bmatrix} a_{1i} & a_{2i} & \cdots & a_{(i-1)i} & a_{(i+1)i} & \cdots & a_{ni} \end{bmatrix}^T$ and $P_3 = a_{ii}$, then $(\mathbf{E'^T E'})^{\mathbf{-1}}$ could be updated by Eq. 6 as below,

$$(\mathbf{E}'^T\mathbf{E}')^{-1}{=}\mathbf{P}_1 - (\mathbf{P}_2\mathbf{P}_2^T)/P_3 \tag{6}$$

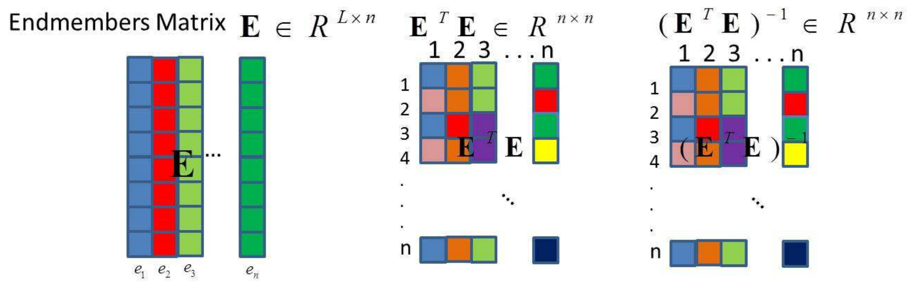
3. Proposed method

In this section, we firstly improve NNLS by partitioned Matrix Inversion theory, then we rearrange the iterative process both in NNLS and FCLS, respectively. At last, we introduce GPU implementation in details.

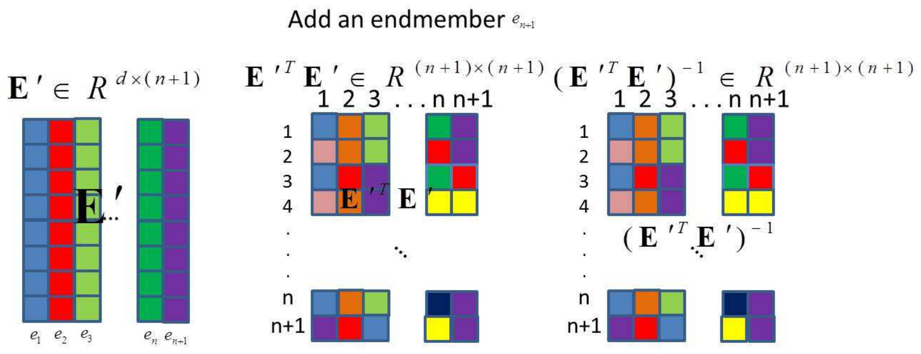
3.1. NNLS with Partitioned Matrix Inversion

In order to accelerate matrix inversion, we embed sequential partitioned matrix inversion in NNLS. Compared with NNLS, we improve NNLS in four steps as shown in Algorithm. 2. First, we pre-define two matrix, i.e. $\mathbf{invE_PTE_P}$ and $\mathbf{ETy}$ in step 3. Secondly, we upgrade the matrix $\mathbf{invE_PTE_P}$ at step 6 and 12. In step 6, it is a situation that an endmember is added to endmember matrix, therefore, Eq. 4 is applied. On the contrary, In step 12 Eq. 6 is applied for removing an endmember from endmember matrix. Finally, we replace least squares with matrix multiply i.e. $\mathbf{invE_PTE_P} \times \mathbf{ETy_P}$ at step 8.

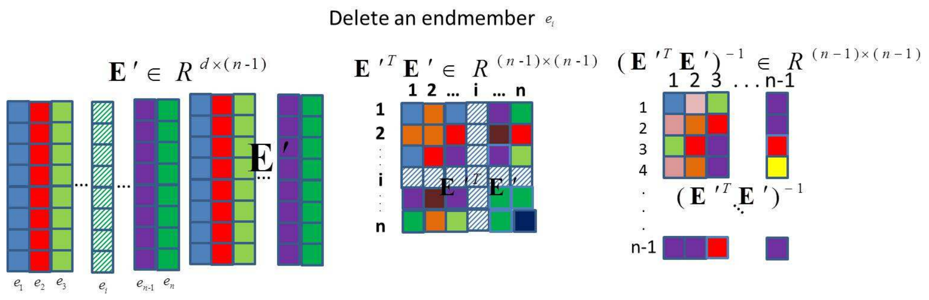
With partitioned matrix inversion, we avoid matrix inversion and improve algorithm efficiency. As we all know, the algorithm complexity of n-dimension matrix inversion is $O(n^3)$. If an endmember is added to endmember matrix, complexity of computation matrix $\mathbf{A}$ is $O(Ln)$,b is $O(L)$, $\mathbf{P_1}$,$\mathbf{P_2}$ and P_3 are all


Endmembers Matrix $\mathbf{E} \in R^{L\times n}$
$\mathbf{E}^T\mathbf{E} \in R^{n\times n}$
$(\mathbf{E}^T\mathbf{E})^{-1} \in R^{n\times n}$
e_1 e_2 e_3 e_n

(a) Original endmembers matrix


Add an endmember e_{n+1}
$\mathbf{E}' \in R^{d\times(n+1)}$
$\mathbf{E}'^T\mathbf{E}' \in R^{(n+1)\times(n+1)}$
$(\mathbf{E}'^T\mathbf{E}')^{-1} \in R^{(n+1)\times(n+1)}$
e_1 e_2 e_3 e_n e_{n+1}

(b) Adding an endmember


Delete an endmember e_i
$\mathbf{E}' \in R^{d\times(n-1)}$
$\mathbf{E}'^T\mathbf{E}' \in R^{(n-1)\times(n-1)}$
$(\mathbf{E}'^T\mathbf{E}')^{-1} \in R^{(n-1)\times(n-1)}$
e_1 e_2 e_3 e_i e_{n-1} e_n

(c) Deleting an endmember

Figure 1: Changes of endmembers matrix $\mathbf{E}$,$\mathbf{E^TE}$ and $(\mathbf{E^TE})^{-1}$.

$O(n^2)$. Complexity of matrix multiply is equal to $O(n^2)$. Therefore, complexity of adding an endmember by partitioned matrix inversion is $O(n(n+L))$. Similar analysis on the situation that an endmember is deleted to endmember matrix, complexity is $O(n^2)$. Therefore, complexity of replacing least squares with partitioned matrix inversion would decrease from $O(n^3)$ to $O(n(n+L))$ for one pixel unmixing. Furthermore, as far as we know, CUDA does not include a single-threaded matrix inversion function on the device side. Therefore, it is much easier to GPU-implement by CUDA with a sequential approach.

3.2. Rearrangement Iterative Process

In Algorithm. 1 and Algorithm. 2, we choose initial solution is equal to $\mathbf{0}$ and iteration sometimes need add an endmember and sometimes delete one. If we reasonably choose initial solution and rearrangement iterative process, both NNLS and FCLS could be greatly simplified. In [3], there is an improved iterative process to be introduced in FCLS. Thus, we refer to this iterative process and introduce partitioned matrix inversion method into it for both FCLS and NNLS. Firstly, different from NNLS, the result of unconstraint least squares is set as initial solution, which means all indices are pushed into the passive set. Secondly, according to the constraint whether includes ASC or not, the Lagrange multiplier function is introduced or not. it is necessary to modify initial solution with ASC. Lagrange multiplier function for least squares with ASC can be defined by,

$$L(\mathbf{a},\lambda) = \frac{1}{2}(\mathbf{y}-\mathbf{Ea})^T(\mathbf{y}-\mathbf{Ea}) - \lambda(\mathbf{1}^T\mathbf{a}-1) \tag{7}$$

Derivative of L respect to λ, and set it to zero, we can derived,

$$\mathbf{1}^T\mathbf{a} = 1 \tag{8}$$

Derivative of L respect to $\mathbf{a}$ and set it to zero, we can derived,

$$\begin{gathered}\mathbf{E}^T(\mathbf{y}-\mathbf{Ea}) + \lambda\mathbf{1} = \mathbf{0} \\ (\mathbf{E}^T\mathbf{E})^{-1}\mathbf{E}^T\mathbf{y} + \lambda(\mathbf{E}^T\mathbf{E})^{-1}\mathbf{1} = \mathbf{a}\end{gathered} \tag{9}$$

Algorithm 2 Non-Negative Least squares with Partitioned Matrix Inversion

```
1: **Input** : Endmembers matrix $\mathbf{E} \in \mathbb{R}^{L\times p}$, Pixel vector $\mathbf{y} \in \mathbb{R}^{L}$
2: **Output** : Abundance vector $\mathbf{a} \in \mathbb{R}^{p}$.
3: **Initializiation** : Passive set $\mathbf{P} = \mathbf{NULL}$,and Active Set $\mathbf{Z} = \{1, 2, \cdots, \mathrm{p}\}$,
   $\mathbf{a} = \mathbf{0}$, $\mathbf{invE_PTE_P} = \mathbf{NULL}$ and $\mathbf{ETy} = \mathbf{E^T y}$.
4: Compute the dual vector $\mathbf{w} = \mathbf{ETY}$
   **Main-loop beginning**
5: **while** $\mathbf{Z \neq NULL}$ && $\exists \mathbf{w_j} \geq \mathbf{0}, \mathbf{j} \in \mathbf{Z}$ **do**
6:     Find an index $t \in \mathbf{Z}$ **such that** $\mathbf{w_t} = \mathbf{max}\{\mathbf{w_j} : \mathbf{j} \in \mathbf{Z}\}$.Move the index $t$
       from the set **Z to set P**
       and upgrade $\mathbf{invE_PTE_P}$ by Eq. 4.
       **inner-loop beginning**
7:     **repeat**
8:         Compute the vector $\mathbf{z_P} = (\mathbf{invE_PTE_P})(\mathbf{ETy_P})$. where $\mathbf{ETy_P}$ is a
           vector, the j-th element of which is equal to the P[j]-th element of
           $\mathbf{ETy}$, and set vector $\mathbf{z}$, the P[j]-th element of which is equal to the j-th
           element of $\mathbf{z_P}$, the rest element is set to 0.
9:         **if** $\exists \mathbf{z_j} < 0, \mathbf{j} \in \mathbf{P}$ **then**
10:            Find an index $q \in \mathbf{P}$ such that $\mathbf{a_q}/(\mathbf{a_q} - \mathbf{z_q}) = \mathbf{min}\{\mathbf{a_j}/(\mathbf{a_j} - \mathbf{z_j}) :$
               $\mathbf{z_j} \leq \mathbf{0}, \mathbf{j} \in \mathbf{P}\}$.
11:            Set $\alpha :=\mathbf{a_q}/(\mathbf{a_q} - \mathbf{z_q})$. and set $\mathbf{a} := \mathbf{a} + \alpha(\mathbf{z} - \mathbf{a})$
12:            Move an index q from set $\mathbf{P}$ to set $\mathbf{Z}$ and upgrade $\mathbf{invE_PTE_P}$ by
               Eq. 6.
13:        **else**
14:            Set $\mathbf{a} := \mathbf{z}$
15:        **end if**
16:    **until** $\mathbf{z_j} \geq \mathbf{0}, \mathbf{j} \in \mathbf{P}$ **inner-loop ending**
17:    Compute the dual vector $\mathbf{w} = \mathbf{E^T}(\mathbf{y} - \mathbf{Ea})$
18: **end while**
    **Main-loop ending**
```

From Eq. 8, we obtain

$$\mathbf{1}^T(\mathbf{E}^T\mathbf{E})^{-1}\mathbf{E}^T\mathbf{y} + \lambda\mathbf{1}^T(\mathbf{E}^T\mathbf{E})^{-1}\mathbf{1} = \mathbf{1}^T\mathbf{a} = 1$$

$$\lambda = \frac{1 - \mathbf{1}^T(\mathbf{E}^T\mathbf{E})^{-1}\mathbf{E}^T\mathbf{y}}{\mathbf{1}^T(\mathbf{E}^T\mathbf{E})^{-1}\mathbf{1}} \tag{10}$$

From Eq. 9 and Eq. 10, we obtain modified solution with ASC

$$\mathbf{a} = (\mathbf{E}^T\mathbf{E})^{-1}\mathbf{E}^T\mathbf{y} + \left(\frac{1 - \mathbf{1}^T(\mathbf{E}^T\mathbf{E})^{-1}\mathbf{E}^T\mathbf{y}}{\mathbf{1}^T(\mathbf{E}^T\mathbf{E})^{-1}\mathbf{1}}\right)(\mathbf{E}^T\mathbf{E})^{-1}\mathbf{1} \tag{11}$$

And then ANC constraint should be considered. As in NNLS algorithm, we find out all negative components of **a**, which violate the ANC constraints. Now, we should determine which component is the largest offset from constraint. Absolute value function is distinctly a metric to indicate the component, but the scaling factor in Mahalanobis space must be considered. In [3], $|\mathbf{a}[j]/\mathbf{s}[j]|$ is employed as an indicator, the index of the largest value of indicator should be removed from passive set. Finally, partitioned matrix inversion is applied for upgrade system matrix $(\mathbf{E^T E})^{-\mathbf{1}}$ and accelerate least squares at next iteration. The proposed algorithm can be seen in Algorithm 3. Compared with the NNLS algorithm, the proposed method simplifies two loops into only one loop and the outer loop(the main loop) in Algorithm .1 is removed. Thus, only deleting an endmember would happen at each iteration. The whole algorithm requires a maximum of p-1 iterations at the worst situation.

3.3. GPU implementation

The proposed method is not only to improve the speed of single pixel hyperspectral unmixing, but more importantly, it is suitable for GPU parallel development. Hyperspectral unmixing algorithm should be applied for each pixel vector in HSI. Therefore, the proposed method should map to multiple threads on GPU to process different hyperspectral data, which is a typical Single Instruction-Multiple-Thread (SIMT) architecture. Scalar-processors (SP) in GPU handles the instruction address and management multiple threads to ensure they execute algorithm independently. Matrix inversion library function in single threads instruction set is unavailable. Fortunately, our algorithm

Algorithm 3 Abundance Estimation with Constraint Linear Spectral Mixture Model for Hyperspectral Unmixing

```
1:  Input : Endmembers matrix $\mathbf{E} \in \mathbb{R}^{L\times p}$, Pixel vector $\mathbf{y} \in \mathbb{R}^{L}$
2:  Output : Abundance vector $\mathbf{a} \in \mathbb{R}^{p}$.
3:  Initializiation : Passive set $\mathbf{P} = \{1, 2, \cdots, \mathrm{p}\}$, $\mathbf{invE_PTE_P} = (\mathbf{E^TE})^{-1}$ and
    $\mathbf{ETy} = \mathbf{E^Ty}$.
    Main-loop beginning
4:  repeat
5:      Compute the vector $\mathbf{z_P} = (\mathbf{invE_PTE_P})(\mathbf{ETy_P})$. where $\mathbf{ETy_P}$ is a vector, the j-th element of which is equal to the P[j]-th element of $\mathbf{ETy}$
6:      Compute the vector $\mathbf{s} = (\mathbf{invE_PTE_P})\mathbf{1}$ where $\mathbf{1}$ is a vector with all elements equal to one.
7:      if Problem with ASC or Full Constraint then
8:          Compute the Lagrange multiplier $\lambda = (1 - \mathbf{1}^T\mathbf{z_P})/(\mathbf{1}^T\mathbf{s})$ and $\mathbf{z_P} = \mathbf{z_P} + \lambda\mathbf{s}$
9:      end if
10:     if $\exists\mathbf{z_j} < 0, \mathbf{j} \in \mathbf{P}$ then
11:         Find an index $t \in \mathbf{P}$ such that $max\{\left|\frac{\mathbf{z_P}[j]}{\mathbf{s}[j]}\right| : j \in \mathbf{P}\}$.Move the index $t$ from the set P to set Z and update $\mathbf{invE_PTE_P}$ by Eq. 6.
12:     end if
13: until $\mathbf{z_j} \geq \mathbf{0}, \mathbf{j} \in \mathbf{P}$
    Main-loop ending
14: Set abundance vector $\mathbf{a}$: $\mathbf{a}[\mathbf{P}[j]] = \mathbf{z_P}[j]$ and the rest elements of $\mathbf{a}$ are equal to 0
```

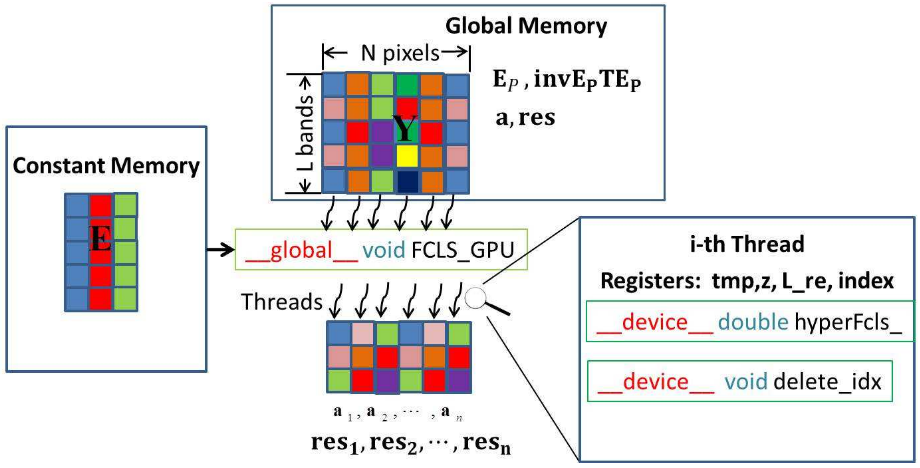
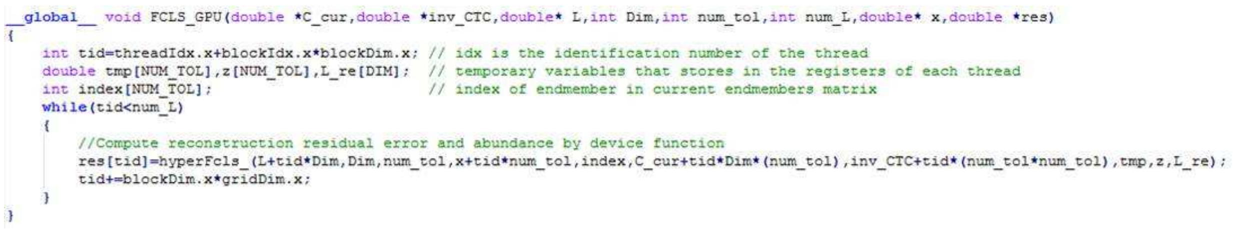

Global Memory
N pixels
L bands
Y
$\mathbf{E}_P$, $\mathbf{invE_P TE_P}$
$\mathbf{a}$, $\mathbf{res}$
Constant Memory
E
__global__ void FCLS_GPU
Threads
$\mathbf{a}_1, \mathbf{a}_2, \cdots, \mathbf{a}_n$
$\mathbf{res_1}, \mathbf{res_2}, \cdots, \mathbf{res_n}$
i-th Thread
Registers: tmp,z, L_re, index
__device__ double hyperFcls_
__device__ void delete_idx

Figure 2: Framework of the proposed FCLS algorithm for GPU implementation.

avoids complicated matrix algebra, and only needs arithmetic operations. We take FCLS as an example to illustrate the framework of our GPU implementation. The framework of the proposed FCLS algorithm is shown in Fig. 2. First, all pixels in Hyperspectral cube are rearranged in lexicographical matrix $\mathbf{Y}$, which are allocated in global memory. Besides that, current endmember matrix $\mathbf{E_p}$, current $\mathbf{invE_p^TE_p}$, the output of abundance $\mathbf{a}$ and reconstruction residual error $\mathbf{res}$ are also allocated in global memory of device. Secondly, since the original endmember matrix $\mathbf{E}$ is often used, we load it into constant memory to speed up the algorithm further. Then, a global function, called kernel in CUDA, *__global__void*$\mathbf{FCLS\_GPU}$ (see Fig. 5) is employed to manage each thread on the device to execute device-side function *__device__double*$\mathbf{hyperFcls\_}$ (see Fig. 4). In this kernel, some temporary variables $\mathbf{tmp}, \mathbf{z}, \mathbf{L_re}$ and the index of endmember $\mathbf{index}$ are allocated in the registers of each thread for faster accessible. Device function *__device__void*$\mathbf{delete\_idx}$ (see Fig. 5) is applied for deleting an endmember and upgrading matrix $\mathbf{invE_p^TE_p}$ by partitioned matrix inversion. Finally, we merge the outputs of each thread. NNLS could be similarly implemented as FCLS.

In addition to the GPU framework, there are some tricks to further accel-



```
__global__ void FCLS_GPU(double *C_cur,double *inv_CTC,double* L,int Dim,int num_tol,int num_L,double* x,double *res)
{
    int tid=threadIdx.x+blockIdx.x*blockDim.x; // idx is the identification number of the thread
    double tmp[NUM_TOL],z[NUM_TOL],L_re[DIM];  // temporary variables that stores in the registers of each thread
    int index[NUM_TOL];                        // index of endmember in current endmembers matrix
    while(tid<num_L)
    {
        //Compute reconstruction residual error and abundance by device function
        res[tid]=hyperFcls_(L+tid*Dim,Dim,num_tol,x+tid*num_tol,index,C_cur+tid*Dim*(num_tol),inv_CTC+tid*(num_tol*num_tol),tmp,z,L_re);
        tid+=blockDim.x*gridDim.x;
    }
}
```

Figure 3: CUDA kernel for FCLS.

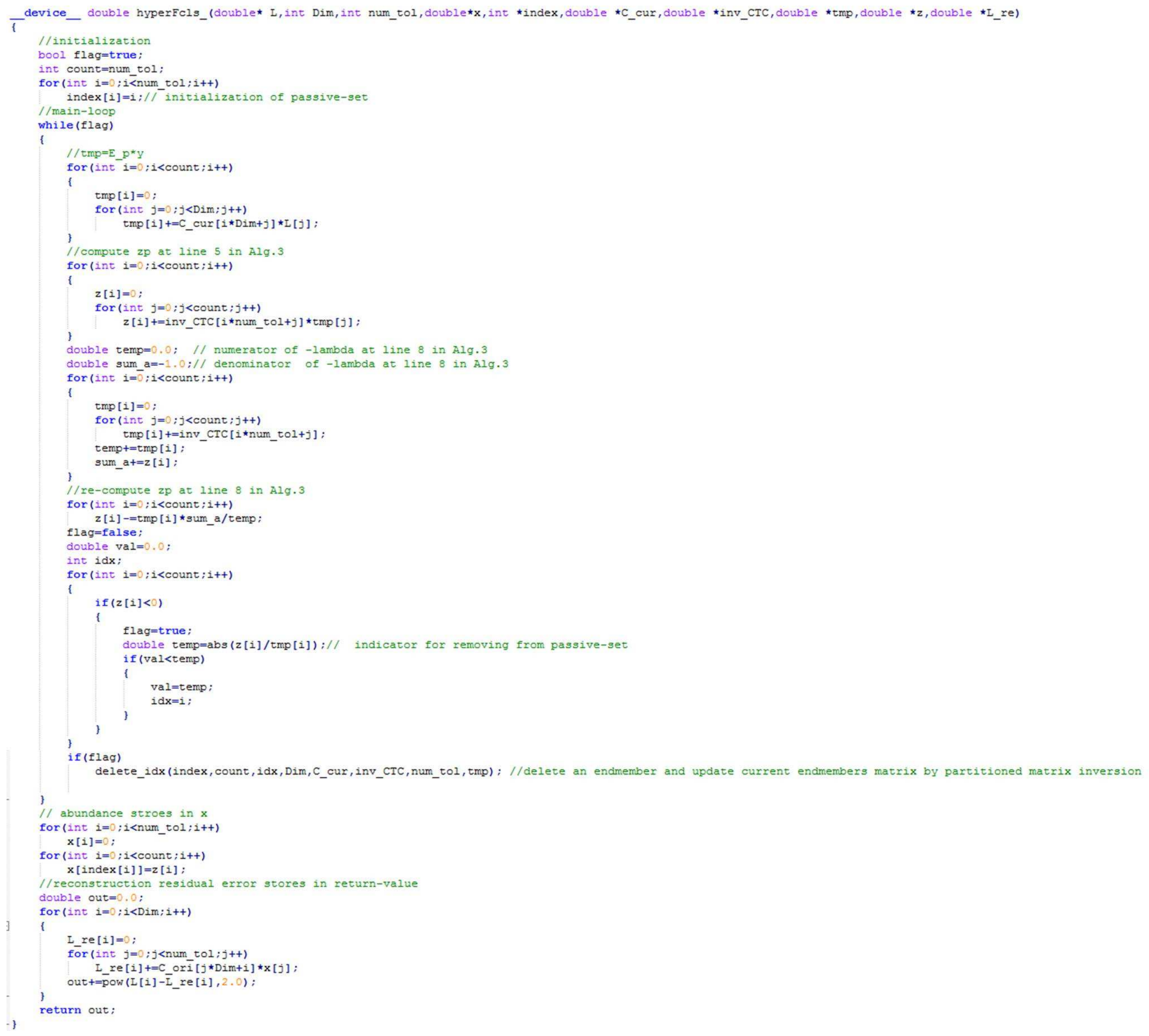
erate the proposed implementation. Pined memory is used to speed up the exchange of global variables (i.e.$\mathbf{E_p}$, $\mathbf{invE_p^TE_p}$, $\mathbf{a}$, $\mathbf{res}$ and $\mathbf{Y}$). And then the other important trick is called CUDA stream with device overlap, which replace one stream processing in with two or multi-streams. Alternate streams processing can hide the time-comsuming of memory copy and accelerate the proposed implementation further. As shown in Fig. 6, the timeline for the proposed GPU-implementation with one stream is displayed when the number of endmember is set to 9 by the NVIDIA Visual Profile tool. We found the usage rate of GPU reaches 100%. However, Almost 1/3 time-consuming spends on memory copy. And Fig. 7 shows the time-line for the proposed GPU-implementation with two streams. Obviously CUDA stream with device overlap saves a lot of time-consuming.

4. Experimental Results

4.1. Hyperspectral Data Sets

Two real hyperspectral data [1] sets are employed to evaluate the efficiency of the proposed method on hyperspectral unmixing. The proposed NNLS and FCLS methods are implemented in Microsoft Visual Studio 2012 on a desktop computer with an Intel(R) Core(TM) i7-7700K 4.20-GHz central processing unit with 16-GB memory and a NVIDIA Geforce GTX960 for GPU computing,

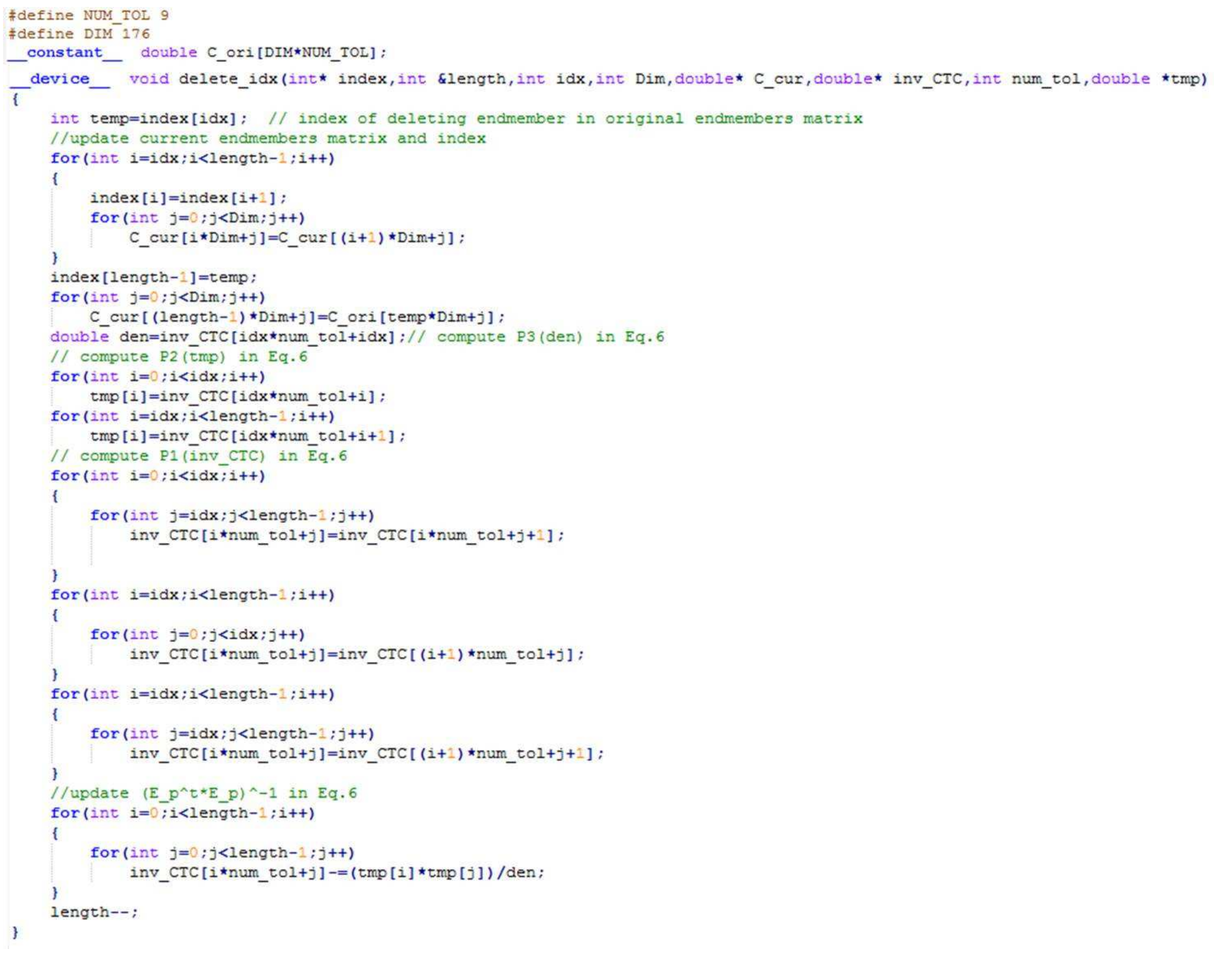
[1] All the experimental data sets are available from http://www.ehu.eus/ccwintco/index.php?title=Hyperspectral_Remote_Sensing_Scenes



```
__device__ double hyperFcls_(double* L,int Dim,int num_tol,double*x,int *index,double *C_cur,double *inv_CTC,double *tmp,double *z,double *L_re)
{
    //initialization
    bool flag=true;
    int count=num_tol;
    for(int i=0;i<num_tol;i++)
        index[i]=i;// initialization of passive-set
    //main-loop
    while(flag)
    {
        //tmp=E_p*y
        for(int i=0;i<count;i++)
        {
            tmp[i]=0;
            for(int j=0;j<Dim;j++)
                tmp[i]+=C_cur[i*Dim+j]*L[j];
        }
        //compute zp at line 5 in Alg.3
        for(int i=0;i<count;i++)
        {
            z[i]=0;
            for(int j=0;j<count;j++)
                z[i]+=inv_CTC[i*num_tol+j]*tmp[j];
        }
        double temp=0.0;  // numerator of -lambda at line 8 in Alg.3
        double sum_a=-1.0;// denominator  of -lambda at line 8 in Alg.3
        for(int i=0;i<count;i++)
        {
            tmp[i]=0;
            for(int j=0;j<count;j++)
                tmp[i]+=inv_CTC[i*num_tol+j];
            temp+=tmp[i];
            sum_a+=z[i];
        }
        //re-compute zp at line 8 in Alg.3
        for(int i=0;i<count;i++)
            z[i]-=tmp[i]*sum_a/temp;
        flag=false;
        double val=0.0;
        int idx;
        for(int i=0;i<count;i++)
        {
            if(z[i]<0)
            {
                flag=true;
                double temp=abs(z[i]/tmp[i]);//  indicator for removing from passive-set
                if(val<temp)
                {
                    val=temp;
                    idx=i;
                }
            }
        }
        if(flag)
            delete_idx(index,count,idx,Dim,C_cur,inv_CTC,num_tol,tmp); //delete an endmember and update current endmembers matrix by partitioned matrix inversion

    }
    // abundance stroes in x
    for(int i=0;i<num_tol;i++)
        x[i]=0;
    for(int i=0;i<count;i++)
        x[index[i]]=z[i];
    //reconstruction residual error stores in return-value
    double out=0.0;
    for(int i=0;i<Dim;i++)
    {
        L_re[i]=0;
        for(int j=0;j<num_tol;j++)
            L_re[i]+=C_ori[j*Dim+i]*x[j];
        out+=pow(L[i]-L_re[i],2.0);
    }
    return out;
}
```

Figure 4: Device function for FCLS.



```
#define NUM_TOL 9
#define DIM 176
__constant__  double C_ori[DIM*NUM_TOL];
__device__  void delete_idx(int* index,int &length,int idx,int Dim,double* C_cur,double* inv_CTC,int num_tol,double *tmp)
{
    int temp=index[idx];  // index of deleting endmember in original endmembers matrix
    //update current endmembers matrix and index
    for(int i=idx;i<length-1;i++)
    {
        index[i]=index[i+1];
        for(int j=0;j<Dim;j++)
            C_cur[i*Dim+j]=C_cur[(i+1)*Dim+j];
    }
    index[length-1]=temp;
    for(int j=0;j<Dim;j++)
        C_cur[(length-1)*Dim+j]=C_ori[temp*Dim+j];
    double den=inv_CTC[idx*num_tol+idx];// compute P3(den) in Eq.6
    // compute P2(tmp) in Eq.6
    for(int i=0;i<idx;i++)
        tmp[i]=inv_CTC[idx*num_tol+i];
    for(int i=idx;i<length-1;i++)
        tmp[i]=inv_CTC[idx*num_tol+i+1];
    // compute P1(inv_CTC) in Eq.6
    for(int i=0;i<idx;i++)
    {
        for(int j=idx;j<length-1;j++)
            inv_CTC[i*num_tol+j]=inv_CTC[i*num_tol+j+1];

    }
    for(int i=idx;i<length-1;i++)
    {
        for(int j=0;j<idx;j++)
            inv_CTC[i*num_tol+j]=inv_CTC[(i+1)*num_tol+j];
    }
    for(int i=idx;i<length-1;i++)
    {
        for(int j=idx;j<length-1;j++)
            inv_CTC[i*num_tol+j]=inv_CTC[(i+1)*num_tol+j+1];
    }
    //update (E_p^t*E_p)^-1 in Eq.6
    for(int i=0;i<length-1;i++)
    {
        for(int j=0;j<length-1;j++)
            inv_CTC[i*num_tol+j]-=(tmp[i]*tmp[j])/den;
    }
    length--;
}
```

Figure 5: Device function for removing an endmember.

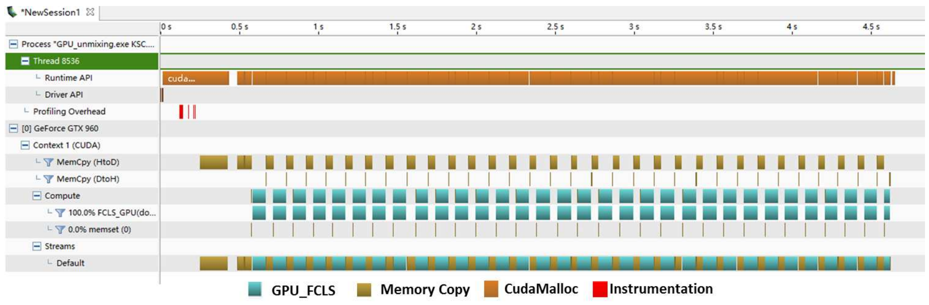


Figure 6: NVIDIA Visual Profile for the proposed GPU-implementation with one stream when the number of endmember is set to 9.

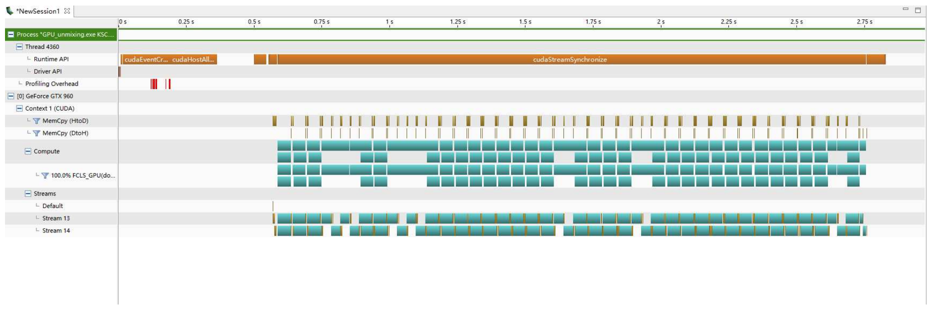
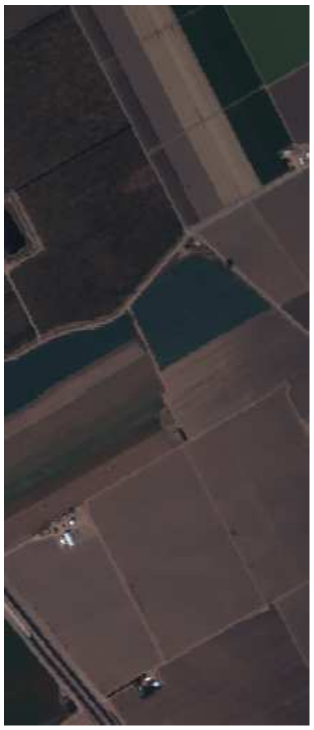


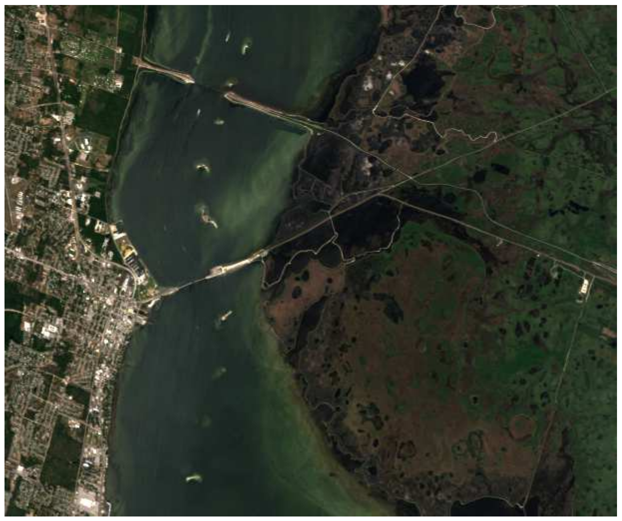
Figure 7: NVIDIA Visual Profile for the proposed GPU-implementation with two streams when the number of endmember is set to 9.

Figure 8: NVIDIA Geforce GTX 960 GPU

as shown in Fig. 8. First hyperspectral data set was derived from the AVIRIS sensor with a ground resolution of 3.7m over Salinas Valley, California. We select an area of 512×217 pixels with 224 bands for the following experiments. The color image is shown in Fig. 9(a). The second hyperspectral data set was also acquired by AVIRIS over the Kennedy Space Center (KSC), Florida, on March 23, 1996. Different from Salinas data, the image scene of KSC data consists of size 512×614 pixels and with a spatial resolution of 18m and after removing water absorption and low SNR bands, 176 bands would be employed for the analysis. The color image is shown in Fig. 9(b). In both data sets, vertex component analysis (VCA [21]) algorithm is applied to extract endmembers. Different number of endmembers consists of the endmembers matrix as an input to test the efficiencies of the proposed implementations (called GPU-NNLS-RIP-Stream) and other implementations. Two assessment metrics are adopted for quantitative evaluation of these efficiencies, i.e. the runtime of implementation



(a) the image scene of Salinas data



(b) the image scene of KSC data

Figure 9: Experimental hyperspectral data sets.

and the speedup ratio(SR), which is defined as,

$$speedup\_ratio = \frac{T_1}{T_2} \tag{12}$$

where T_1 and T_2 are the runtime of the implementation before and after speedup, respectively. The number of endmembers is set to $[3-15, 20, 25]$ for Salinas data and $[3:3:30, 32]$ for KSC data respectively. To verify the efficiency of the proposed NNLS algorithm, several NNLS implementations with different versions (i.e., Matlab, CPU and GPU versions, respectively) are compared with the proposed implementation. Matlab implementations include Matlab's lsqnonneg function (as a baseline) and fast-NNLS ([2], called Matlab-fast-NNLS), note that Matlab2016b version is employed in our experiments, which use Intels Math Kernel Library (MKL) with multi-cores CPU to solve the least squares problems (called Matlab-lsqnonneg-MKL). We also implement Matlab's lsqnonneg without MKL function (called Matlab-lsqnonneg). CPU implementation includes re-implementation lsqnonneg function with C++ codes and CLAPACK library, called CPU-NNLS. GPU implementations include the NNLS with/without rearrangement iterative process (called GPU-NNLS and GPU-NNLS-RIP,respectively) and CUDA stream device overlap are not applied by both of them. To verify the efficiency of the proposed FCLS implementations (called GPU-NNLS-Stream), Matlab FCLS, CPU-FCLS with CLAPACK library and GPU-FCLS without CUDA stream device overlap are empolyed for comparison. Note that, the initialization at line 3 in Algorithm. 2, CULA library [22]or CLAPARK library [2]could be applied for pre-computing $(\mathbf{E^T E})^{-1}$. All implementations (including compared algorithms) could be downloaded in website: https://github.com/l7170/NNLS-FCLS-GPU.git.

4.2. Experimental Results Analysis

First of all, comparison of NNLS implementations is evaluated. Total number of pixels in Salinas data is equal to 111104, and 314368 in KSC data, re-

[2]CLAPARK library is available from http://www.netlib.org/clapack/

Table 1: Runtime (seconds) and Speedup ratio of NNLS with Salinas data.

Number of Endmembers	Matlab-lsqnonneg	Matlab-fast-NNLS	Matlab-lsqnonneg-MKL	CPU-NNLS	GPU-NNLS	GPU-NNLS-RIP	GPU-NNLS-RIP-Stream
3	16.21s	14.50s	14.73s	10.50s	0.3604s	0.3475s	**0.1805s**
4	20.02s	17.43s	17.18s	13.39s	0.7840s	0.4661s	**0.2454s**
5	19.03s	17.09s	16.95s	14.34s	0.8264s	0.5737s	**0.3119s**
6	20.12s	19.16s	17.43s	15.74s	0.9455s	0.6980s	**0.4076s**
7	29.40s	23.34s	23.59s	22.40s	2.371s	0.8797s	**0.5066s**
8	29.82s	25.35s	24.29s	27.09s	2.852s	1.065s	**0.6435s**
9	29.56s	25.79s	24.04s	31.74s	3.381s	1.322s	**0.7792s**
10	28.63s	23.22s	24.34s	27.99s	4.734s	1.631s	**1.022s**
11	29.65s	25.99s	24.58s	31.48s	3.858s	1.975s	**1.298s**
12	30.90s	26.40s	25.66s	25.57s	3.474s	2.051s	**1.426s**
13	30.90s	25.66s	26.40s	27.82s	5.361s	2.389s	**1.651s**
14	24.36s	23.99s	20.09s	26.26s	3.335s	2.648s	**1.799s**
15	29.73s	26.88s	24.61s	27.19s	6.474s	3.233s	**2.164s**
20	39.84s	30.69s	31.38s	44.58s	13.11s	5.315s	**3.610s**
25	41.51s	34.32s	34.18s	41.02s	17.31s	8.616s	**6.013s**
Average of SR	37.37	32.57	31.84	30.47	3.217	1.622	

Table 2: Runtime (seconds) and Speedup ratio of NNLS with KSC data.

Number of Endmembers	Matlab-lsqnonneg	Matlab-fast-NNLS	Matlab-lsqnonneg-MKL	CPU-NNLS	GPU-NNLS	GPU-NNLS-RIP	GPU-NNLS-RIP-Stream
3	48.78s	42.32s	43.70s	28.31s	0.9538s	0.8542s	**0.4770s**
6	46.60s	44.69s	41.69s	29.11s	2.490s	1.656s	**0.9098s**
9	53.92s	51.97s	48.11s	39.35s	4.524s	3.264s	**1.973s**
12	58.56s	54.70s	51.19s	54.51s	6.350s	4.841s	**2.864s**
15	69.02s	63.91s	57.81s	66.18s	10.28s	7.814s	**4.664s**
18	69.39s	65.93s	60.13s	66.55s	14.59s	11.09s	**6.987s**
21	95.86s	78.38s	80.61s	120.5s	21.06s	14.47s	**8.972s**
24	79.59s	64.03s	60.20s	89.19s	20.22s	19.14s	**12.20s**
27	142.9s	102.4s	126.9s	196.8s	46.98s	25.27s	**15.19s**
30	111.4s	81.07s	89.40s	141.4s	31.92s	31.12s	**19.14s**
32	127.5s	109.9s	101.7s	204.4s	40.65s	35.87s	**21.28s**
Average of SR	24.04	21.50	21.12	18.61	2.201	1.670	

spectively. The runtime comparisons of Salinas data and KSC data are respectively shown in Fig. 10 and Fig. 11. The detail runtime and speedup ratio are listed in Tab. 1 and Tab. 2, respectively. In both data sets, our proposed GPU implementation (GPU-NNLS-RIP) has the highest efficiency in all algorithms and implementations regardless of the number of endmembers. Compared with Matlab-lsqnonneg-MKL, the baseline implementation, the proposed implementation speeds up 31.84 times on the average. Compared with Matlab-lsqnonneg without MKL, our implementation speedups 37.37 times on the average. Compared with fast-NNLS, our implementation speeds up 32.57 times on the average. Compared with CUP-NNLS, our implementation speeds up 30.47 times on the average. In addition to the improvement in efficiency, the implementations with RIP are more stable than other implementations. The time-consuming of the implementation with RIP increases as the grow in number of the endmembers, but the time-consuming of others maybe have some fluctuations such as the number is equal to 10, 12 and 14 in Salinas data and 24, 30 in KSC data (See in Fig. 10 and Fig. 11). Due to the uncertainty that the run-time of abundance estimation by these other implementations, it would affect hyperspectral unmixing in real-time. Among Matlab versions of NNLS, Matlab-lsqnonneg without MKL has the largest time-consuming, and because MKL is employed by Multi-cores acceleration , Matlab-lsqnonneg-MKL could improve by 10%-26%. Through pre-defining the variables, fast-NNLS could improve by 4%-39%. When the number of endmembers is less than 8, CPU-NNLS is faster than Matlab versions, but slower as the grow in number of endmembers. By utilizing the GPU parallel computing architecture, GPU implementations have been greatly improved in run-time. Compared with GPU versions i.e. GPU-NNLS and GPU-NNLS-RIP, our proposed implementation could further increase the efficiency by 3.217 and 1.622 times on the average, respectively.

Secondly, we also compare the FCLS implementations in Fig. 12 for Salinas data and Fig. 13 for KSC data, respectively. The detail runtime and speedup ratio are listed in Tab. 3 and Tab. 4, respectively. Our proposed implementation GPU-FCLS-Stream has the highest efficiency again in both data set. Because

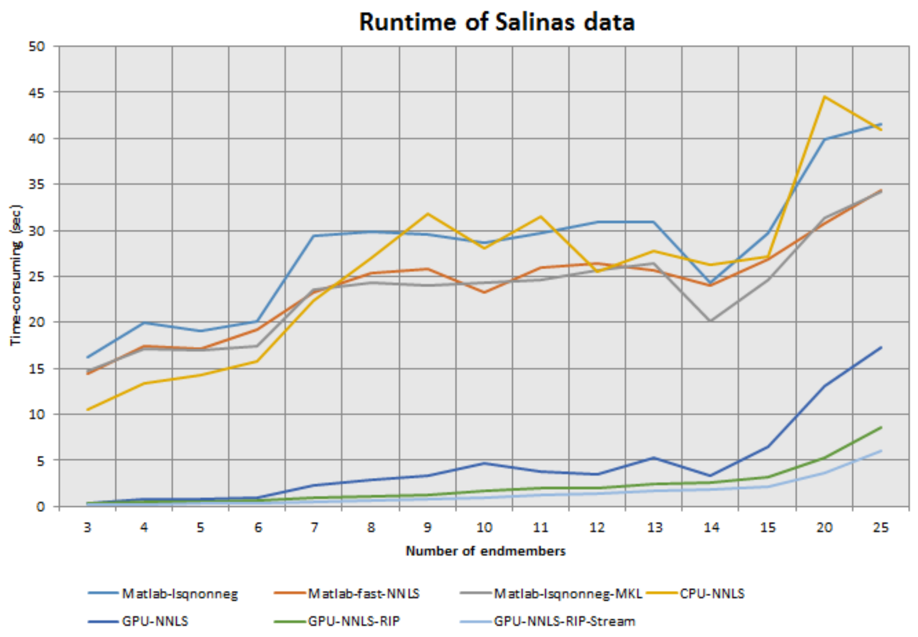


Figure 10: The runtime (seconds) comparisons of NNLS with Salinas data.

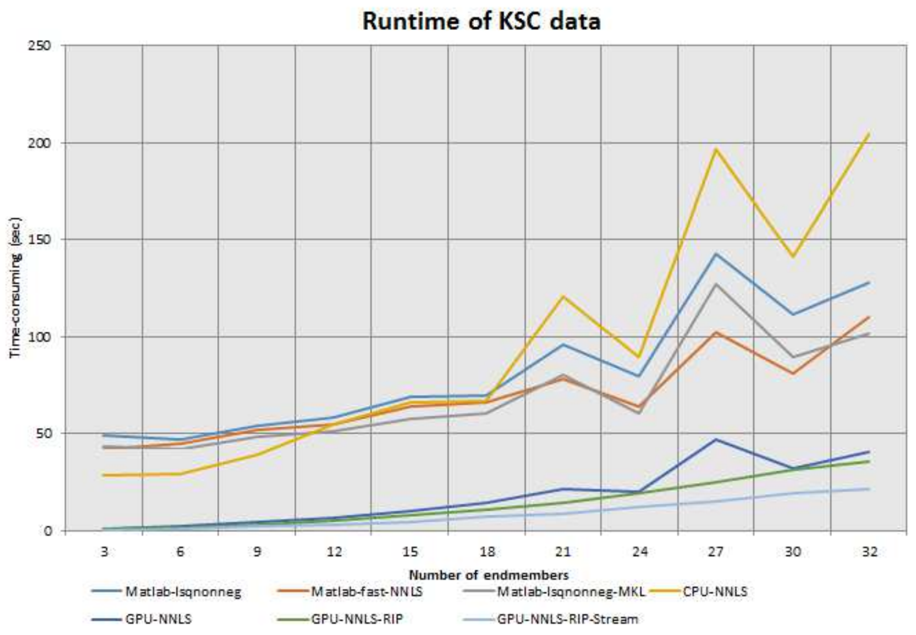


Figure 11: The runtime (seconds) comparisons of NNLS with KSC data.

Table 3: Runtime (seconds) and Speedup ratio of FCLS with Salinas data.

Number of Endmembers	Matlab-FCLS	CPU-FCLS	GPU-FCLS	GPU-FCLS-Stream
3	18.71s	3.265s	0.3645s	**0.1955s**
4	39.85s	7.015s	0.5271s	**0.2833s**
5	46.98s	6.281s	0.6413s	**0.3474s**
6	61.83s	7.437s	0.7943s	**0.4249s**
7	81.64s	11.98s	1.027s	**0.5642s**
8	90.43s	13.88s	1.211s	**0.6880s**
9	116.5s	17.63s	1.437s	**0.7827s**
10	162.5s	30.41s	1.826s	**1.033s**
11	213.4s	41.03s	2.241s	**1.284s**
12	216.5s	47.03s	2.419s	**1.401s**
13	262.3s	53.73s	2.975s	**1.716s**
14	307.4s	72.31s	3.167s	**1.892s**
15	324.8s	88.79s	3.811s	**2.304s**
20	486.3s	157.7s	6.651s	**3.918s**
25	724.5s	333.2s	11.15s	**6.704s**
Average of SR	140.6	28.93	1.768	

Table 4: Runtime (seconds) and Speedup ratio of FCLS with KSC data.

Number of Endmembers	Matlab-FCLS	CPU-FCLS	GPU-FCLS	GPU-FCLS-Stream
3	81.66s	9.952s	0.8667s	**0.4956s**
6	232.4s	23.25s	1.730s	**0.9511s**
9	389.3s	44.36s	3.895s	**2.197s**
12	627.9s	119.3s	5.385s	**3.052s**
15	820.1s	214.6s	8.054s	**4.740s**
18	1047s	289.0s	11.38s	**6.962s**
21	1377s	497.6s	15.72s	**9.793s**
24	1662s	694.5s	21.12s	**13.41s**
27	3139s	914.1s	26.64s	**16.21s**
30	2470s	1004s	33.04s	**20.48s**
32	5133s	1444s	38.51s	**23.04s**
Average of SR	174.3	41.94	1.686	

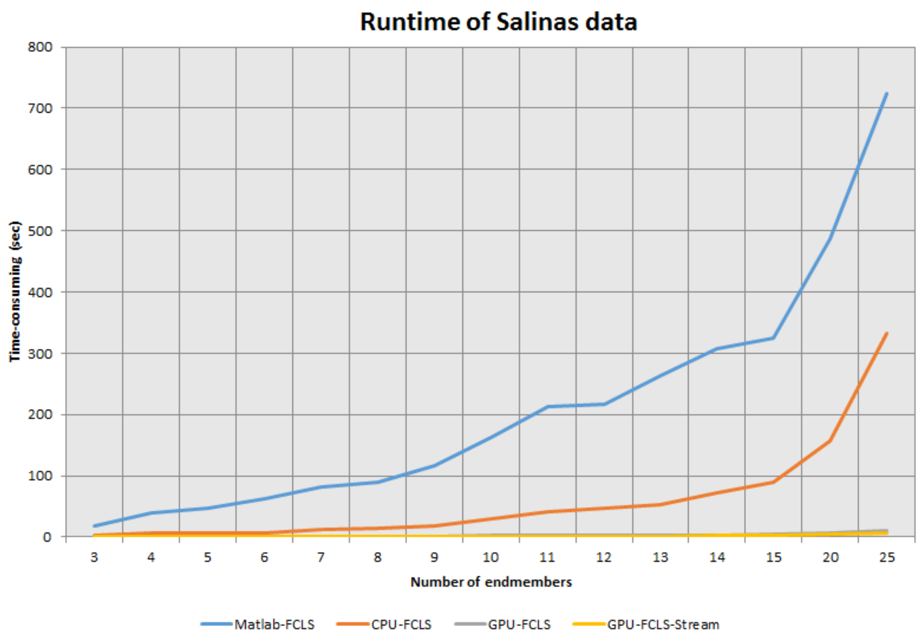


Figure 12: The runtime (seconds) comparisons of FCLS with Salinas data.

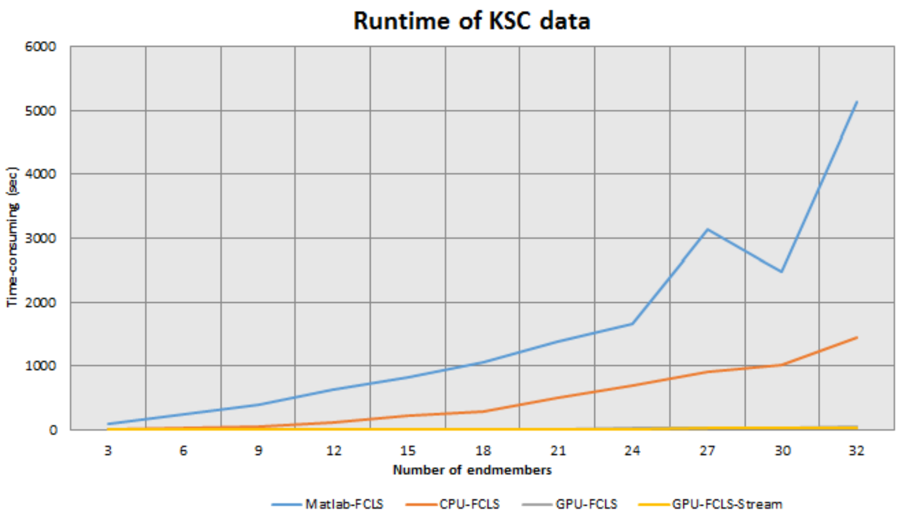


Figure 13: The runtime (seconds) comparisons of FCLS with KSC data.

Matlab-FCLS has not employed MKL, least square and inversion matrix take up too much time-consuming in Matlab implementations. Compared with the Matlab version, our proposed implementation increases 140.6 times in Salinas and 174.3 times in KSC on the average. Similarly, compared with the CPU version, our proposed implementation increases 28.93 times in Salinas and 41.94 times in KSC on the average. With CUDA stream device overlap, the proposed implementation could be further improve 1.768 and 1.686 times respectively in both data sets on the average. The experimental results of both two data sets indicate the larger number of endmembers employed in FCLS abundance estimation, the higher speedup rate of the proposed GPU implementation is available.

Finally, the feasibility of real-time large-scale abundance estimation with constraint linear spectral mixture model for hyperspectral unmixing would be analyzed. As the conclusion of [5], Salinas data set should estimate all abundance in $5 \times 217 \div 614 = 1.767$sec, and KSC data set should be in $5 \times 176 \div 224 = 3.929$sec. Therefore, from Tab. 1,Tab. 2,Tab. 3 and Tab. 4, when the number of endmembers is less than 14, real-time processing in both data sets could be completely satisfied for a single NVIDIA Geforce GTX960 GPU. If equal to or more than 14 endmembers is used, more than one GPU or more computing power GPU need to solve the problem.

5. Conclusions

In this paper, a parallelized sequential least squares algorithm is proposed for large-scale abundance estimation. Firstly, partitioned matrix inversion theory is applied to avoid frequent matrix inversion in NNLS or FCLS. And then an improved iterative process is also introduced to further decrease time-consuming. Finally, GPU parallelized implementation with CUDA stream device overlap is employed to reach the purpose of real-time processing. In our experiments, the real HSI datasets were tested to verify the efficiency of GPU parallel implementation. Experimental results demonstrate that our implementation is already much more efficient than NNLS, fast NNLS and FCLS in traditional version such as Matlab or Visual Studio with the same accuracy. With the proper GPU, the proposed implementation could be completely satisfied the demand of real-time large-scale abundance estimation with NNLS or FCLS for hyperspectral unmixing. In the future, we will apply multi-GPU collaboration processing to our implementation.

Acknowledgment

This work was supported by

References

[1] C. L. Lawson, R. J. Hanson, Solving least squares problems.

[2] R. Bro, S. D. Jong, A fast non-negativity-constrained least squares algorithm, Journal of Chemometrics 11 (5) (2015) 393–401.

[3] D. Heinz, C. . Chang, M. L. G. Althouse, Fully constrained least-squares based linear unmixing [hyperspectral image classification], in: IEEE 1999 International Geoscience and Remote Sensing Symposium. IGARSS'99 (Cat. No.99CH36293), Vol. 2, 1999, pp. 1401–1403 vol.2. doi:10.1109/IGARSS.1999.774644.

[4] R. O. Green, M. L. Eastwood, C. M. Sarture, T. G. Chrien, M. Aronsson, B. J. Chippendale, J. A. Faust, B. E. Pavri, C. J. Chovit, M. Solis, Imaging spectroscopy and the airborne visible/infrared imaging spectrometer (aviris), Remote Sensing of Environment 65 (3) (1998) 227–248.

[5] E. Torti, G. Danese, F. Leporati, A. Plaza, A hybrid cpugpu real-time hyperspectral unmixing chain, IEEE Journal of Selected Topics in Applied Earth Observations & Remote Sensing 9 (2) (2016) 945–951.

[6] Parallel unmixing of remotely sensed hyperspectral images on commodity graphics processing units, 2011 Concurrency and Computation Practice and Experience.

[7] Near-infrared hyperspectral unmixing based on a minimum volume criterion for fast and accurate chemometric characterization of counterfeit tablets, Analytical Chemistry 82 (4) 1462–1469.

[8] C. E. Koulas, Extracting wildfire characteristics using hyperspectral, lidar, and thermal ir remote sensing systems.

[9] L. Han, L. Li, W. Li, Gpu implementation of rx detection using spectral derivative features, Journal of Real-Time Image Processing.

[10] S. Bernabe, S. Lopez, A. Plaza, R. Sarmiento, Gpu implementation of an automatic target detection and classification algorithm for hyperspectral image analysis, IEEE Geoscience and Remote Sensing Letters 10 (2) (2013) 221–225. doi:10.1109/LGRS.2012.2198790.

[11] Y. Li, W. Li, L. Li, Hyperspectral anomaly dectection on multicore dsps, in: 2018 11th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI), 2018, pp. 1–5. doi:10.1109/CISP-BMEI.2018.8633118.

[12] M. I. Castillo, J. C. Fernndez, F. D. Igual, A. Plaza, E. S. Quintana-Ort, A. Remn, Hyperspectral unmixing on multicore dsps: Trading off performance for energy, IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing 7 (6) (2014) 2297–2304. doi:10.1109/JSTARS.2013.2266927.

[13] C. Gonzlez, S. Snchez, A. Paz, J. Resano, D. Mozos, A. Plaza, Use of fpga or gpu-based architectures for remotely sensed hyperspectral image processing, Integration 46 (2) (2013) 89 – 103. doi:https://doi.org/10.1016/j.vlsi.2012.04.002.

URL http://www.sciencedirect.com/science/article/pii/S0167926012000223

[14] J. M. P. Nascimento, M. Vestias, G. Martin, Fpga-based architecture for hyperspectral unmixing, in: IGARSS 2015 - 2015 IEEE International Geoscience and Remote Sensing Symposium, 2015.

[15] M. A. Hossam, H. M. Ebied, M. H. Abdel-Aziz, Hybrid cluster of multicore cpus and gpus for accelerating hyperspectral image hierarchical segmentation, in: The 8th International Conference on Computer Engineering & Systems (ICCES'13), pp. 262-267, 2013.

[16] D. Valencia, A. J. Plaza, P. M. Cobo, J. Plaza, On the use of cluster computing architectures for implementation of hyperspectral image analysis algorithms, in: Proceedings of the 10th IEEE Symposium on Computers and Communications (ISCC 2005), 27-30 June 2005, Murcia, Cartagena, Spain, 2005.

[17] E. M. Sigurdsson, A. Plaza, J. A. Benediktsson, Gpu implementation of iterative-constrained endmember extraction from remotely sensed hyperspectral images, IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing 8 (6) (2015) 2939–2949. doi:10.1109/JSTARS.2015.2441699.

[18] E. Torti, G. Danese, F. Leporati, A. Plaza, A hybrid cpugpu real-time hyperspectral unmixing chain, IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing 9 (2) (2016) 945–951. doi:10.1109/JSTARS.2015.2485399.

[19] E. Martel, R. Guerra, S. Lpez, R. Sarmiento, A gpu-based processing chain for linearly unmixing hyperspectral images, IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing 10 (3) (2017) 818–834. doi:10.1109/JSTARS.2016.2614842.

[20] Y. Luo, R. Duraiswami, Efficient parallel nonnegative least squares on multicore architectures, Siam Journal on Scientific Computing 33 (5) 2848–2863.

[21] J. M. P. Nascimento, J. M. B. Dias, Vertex component analysis: A fast algorithm to unmix hyperspectral data, IEEE Transactions on Geoscience & Remote Sensing 43 (4) (2005) 898–910.

[22] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, E. J. Kelmelis, Cula: hybrid gpu accelerated linear algebra routines 1 (1) (2010) 456–9003.