Real-Time Implementation of the Sparse Multinomial Logistic Regression for Hyperspectral Image Classification on GPUs

Zebin Wu (吴泽彬), Member, IEEE, Qicong Wang (王启聪), Antonio Plaza, Fellow, IEEE, Jun Li (李军), Member, IEEE, Le Sun (孙乐), Member, IEEE, and Zhihui Wei (韦志辉)

Abstract—In this letter, a real-time implementation of the logistic regression via variable splitting and augmented Lagrangian (LORSAL) algorithm for sparse multinomial logistic regression is presented on commodity graphics processing units (GPUs) using Nvidia’s compute unified device architecture. The proposed parallel method properly exploits the GPU architecture at the low level, including its shared memory, and takes full advantage of the computational power of GPUs to achieve real-time classification performance of hyperspectral images for the first time in the hyperspectral imaging literature. Our experimental results reveal remarkable acceleration factors and real-time performance, while retaining exactly the same classification accuracy with regard to the serial and multicore versions of the classifier.

Index Terms—Graphics processing units (GPUs), hyperspectral image classification, parallel.

I. INTRODUCTION

The spectral signatures collected by hyperspectral imaging instruments can be utilized to effectively discriminate various land-cover classes [1]. The extremely high dimensionality of the data, together with the limited number of training samples generally available, pose important challenges for hyperspectral image classification. Various supervised and unsupervised methods have been recently developed for this purpose [2]. Discriminative approaches based on machine learning, such as support vector machines (SVMs) [3] and multinomial logistic regression (MLR) [4], have gained great popularity for supervised classification. These methods can deal with high-dimensional data and limited training samples. MLR-based approaches learn the class distributions themselves in a Bayesian learning framework, thus supplying a degree of plausibility of such classes [5]. The sparse MLR framework (SMLR) has been shown to be able to deal with difficult classification problems [6]. The logistic regression via variable splitting and augmented Lagrangian (LORSAL) algorithm [7], which learns a sparse regressor with Laplace prior distribution for SMLR, has been proven to be a very competitive pixel-based approach for hyperspectral image classification. However, its execution is computationally far from real-time performance, which is very important for swift decisions in time-critical scenarios [8], [10], such as onboard military reconnaissance, environmental quality surveillance, monitoring of chemical contamination, wildfire tracking, biological threat detection, and so on.

Recent advances in high-performance computing with graphics processing units (GPUs) offer a tremendous potential to bridge the gap toward real-time analysis of hyperspectral images, including algorithms such as vertex component analysis [8], sparse unmixing by variable splitting and augmented Lagrangian (SUNSAL) [9], pixel purity index [10], and constrained sparse nonnegative matrix factorization method [12]. Nevertheless, there are very few GPU implementations of hyperspectral image classification algorithms.

In this letter, a real-time implementation of the LORSAL algorithm for hyperspectral image classification is presented on GPUs, using Nvidia’s compute unified device architecture (CUDA). The proposed parallel method properly exploits the GPU architecture at the low level, including its shared memory, and takes full advantage of the computational power of GPUs.

II. CLASSIFICATION USING LORSAL

Let us assume that we have a classification scenario with $K$ distinct classes $\{1, 2, \ldots, K\}$, including $J$ training samples in total. Let $x = [x_1, x_2, \ldots, x_N] \in \mathbb{R}^{d \times N}$ be a hyperspectral image with $N$ pixels, and let $y = [y_1, y_2, \ldots, y_N]$ be an image of class labels, where $x_i \in \mathbb{R}^d$ is an $L$-dimensional hyperspectral pixel observation, and $y_i = [y_i^{(1)}, y_i^{(2)}, \ldots, y_i^{(K)}]$ denote a “1-of-$K$” encoding of the $K$ classes, where $\sum_{k=1}^{K} y_i^{(k)} = 1$, $y_i^{(k)} \in \{0, 1\}$ for hard labels and $y_i^{(k)} \in [0, 1]$ for soft labels.
for soft labels. Let us also assume that \( w^{(k)} \) is a feature weight vector for the \( k \)th class. According to the MLR model in [4], given a sample \( x_i \) of the \( k \)th class, the probability density \( p(y_i|x_i, w) \) can be modeled as

\[
p_i^{(k)} = p(y_i^{(k)} = 1|x_i, w) = \frac{\exp(w^{(k)T}h(x_i))}{\sum_{j=1}^{N} \exp(w^{(j)T} h(x_i))}
\]  

(1)

where \( w = [w^{(1)}, \ldots, w^{(K)}] \) and \( p_i = [p_i^{(1)}, \ldots, p_i^{(K)}] \) are feature weight vectors, \( k \in \{1, 2, \ldots, K\} \), \( i \in \{1, 2, \ldots, N\} \), and the notation \( (\cdot)^T \) stands for vector transposed. Similarly, \( h(x) = [h_1(x), \ldots, h_l(x)] \) is a vector of \( l \) fixed functions of the input, which are often termed features. In general, \( h(x) \) can be chosen as a kernel function \( h(x) = [K(x, x_1), \ldots, K(x, x_n)]^T \), where \( K(\cdot, \cdot) \) is a symmetric kernel function. In this letter, we use the radial basis function (RBF) \( K(x_i, x_j) = \exp(-\|x_i - x_j\|^2/2\sigma^2) \). If we assume that regressor \( w \) is sparse, the prior of \( w \) can be modeled as a Laplacian density function

\[
p(w) \propto \exp(-\lambda \|w\|_1) \tag{2}
\]

where \( \lambda \) is a regularization parameter that controls the sparsity level of \( w \).

The maximum a posteriori probability of the regressors can then be given by

\[
\hat{w} = \arg\max_w L(w) = \arg\max_w [l(w) + \log P(w)] \tag{3}
\]

where \( l(w) = \log \prod_i p(y_i|x_i, w) \) is a log-likelihood function. Model (3) is nonconvex and hard to solve. However, the log-likelihood function can be approximated to a quadratic lower bound. Based on this lower bound, model (3) can be converted into a convex optimization problem. The LORSAL algorithm [7] converts model (3) into an \( l_0 \) optimization problem by the method of quadratic lower bound approximation and efficiently solves it by the variable splitting and augmented Lagrangian method with computational complexity of \( O(L^2(K - 1)) \). The LORSAL based on SMLR (LORSAL_S) is summarized in Algorithm 1.

Algorithm 1 Serial LORSAL for hyperspectral Classification (LORSAL_S)

Input: Set of training samples \( A \in R^{L \times J} \), number of classes \( K \), Set of test samples \( x = [x_1, x_2, \ldots, x_N] \in R^{L \times N} \);

Initialization: Set \( \lambda > 0, \beta > 0, m = 0 \), maximum iterations \( M \), initialize \( w = w_0, v = v_0, b = b_0, \hat{A} = h(A), \hat{x} = h(x), h(x) = \) RBF kernel function;

Step 1. \( u = -\frac{1}{2}(I - \frac{1}{K}11^T) \), \( 1 \in \{1, 1, \ldots, 1\}^T \), perform the eigendecomposition of \( u, [\xi_u, \lambda_u] = \text{eig}(u) \), \( \lambda_u \) is the square matrix whose main diagonal elements are eigenvalues of \( u, \xi_u \) is the square matrix consists of eigenvectors;

Step 2. \( r = \hat{A} \cdot \hat{A}^T \), perform the eigen-decomposition of \( r, [\xi_r, \lambda_r] = \text{eig}(r) \);

Step 3. \( s = 1/(1 \cdot \text{diag}(\lambda_r))^T \cdot \lambda_u - \beta 11^T \), where \( \text{diag}(\lambda_r) \) is a row vector consists of the main diagonal elements of \( \lambda_r \);

Do:

Step 4. \( p_1^{(k)} = \frac{\exp(w^{(k)T} \hat{A}_i)}{\sum_{j=1}^{K} \exp(w^{(j)T} \hat{A}_i)}, i \in \{1, J\}, k \in \{1, K\} \)

\[
p = \begin{pmatrix}
p_1^{(1)} & p_2^{(1)} & \cdots & p_K^{(1)} \\
p_1^{(2)} & p_2^{(2)} & \cdots & p_K^{(2)} \\
p_1^{(K)} & p_2^{(K)} & \cdots & p_K^{(K)}
\end{pmatrix},
\]

\[
d = r \cdot w \cdot u^T - \hat{A} \cdot (Y - p)^T, 
\]

\[
z = d - \beta (v + b);
\]

Step 5. \( w = s \cdot (\xi_u^T \cdot z \cdot \xi_u), w = \xi_r \cdot w \cdot \xi_u^T; \)

Step 6. \( v = \text{soft}(w - b, \lambda/\beta), \) where \( \text{soft}(\cdot) \) is a soft threshold function \( \text{soft}(z, \mu) = \text{sign}(z) \max(|z| - \mu, 0) \);

Step 7. \( b = b - (w - v) \);

Step 8. \( \beta = 1.05 \cdot \beta, \quad s = 1/((1 \cdot \text{diag}(\lambda_r))^T \cdot \lambda_u - \beta 11^T) \);

Step 9. \( m = m + 1; \)

While \( m < M \)

Step 10.

\[
p_1^{(k)} = \frac{\exp(w^{(k)T} \hat{A}_i)}{\sum_{j=1}^{K} \exp(w^{(j)T} \hat{A}_i)}, i \in \{1, N\}, k \in \{1, K\},
\]

\[
p = \begin{pmatrix}
p_1^{(1)} & p_2^{(1)} & \cdots & p_K^{(1)} \\
p_1^{(2)} & p_2^{(2)} & \cdots & p_K^{(2)} \\
p_1^{(K)} & p_2^{(K)} & \cdots & p_K^{(K)}
\end{pmatrix};
\]

Step 11. \( p_{\text{max}} = \max(p) \), where \( p_{\text{max}} \in R^{1 \times N} \) consists of the maximum elements of each column of \( p, y \in N^{1 \times N} \) consists of the indices of the maximum values of \( p \).

Output: \( y \), the class labels of \( x \).

End

III. GPU IMPLEMENTATION

Our GPU implementation of LORSAL is developed using Nvidia CUDA and the CUDA linear algebra library (CULA) [13]. The key stage of the LORSAL_S in Algorithm 1 is the loop iterative solution of regressor \( w \). Since there are dependencies among some iteration steps (e.g., the calculation of \( w \) and \( p \)), it is hard to parallelize the algorithm as a whole. Our parallel implementation is based on three optimizations: 1) reconstructing the iteration steps with tight coupling; 2) parallelizing the iteration steps with loose coupling; 3) realizing the parallelization at kernel level by CUDA streams.

Furthermore, bearing in mind that the cost of input/output (I/O) communication between the host (CPU) and the device (GPU) is quite expensive, we minimized the data transfers between the host and the device in our implementation. In the parallel optimization process, the data are stored in the local GPU memory as much as possible, and the storage space for intermediate variables of the iterative process is allocated in advance. In general, there are mainly three occasions in which the I/O interactions between host and device take place: 1) at the beginning, we allocate memory on the device, and transfer original data from the CPU to the GPU; 2) when updating
Fig. 1. GPU implementation of LORSAL for hyperspectral classification.

Table I

<table>
<thead>
<tr>
<th>Specification</th>
<th>Platform 1</th>
<th>Platform 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor Number</td>
<td>Intel Xeon E5-2609</td>
<td>Intel Xeon E5-2620 v2</td>
</tr>
<tr>
<td>Processor Base Frequency</td>
<td>2.4 GHz</td>
<td>2.10 GHz</td>
</tr>
<tr>
<td>Number of Cores</td>
<td>8 (2 CPUs)</td>
<td>24 (2 CPUs)</td>
</tr>
<tr>
<td>main memory</td>
<td>32 GB</td>
<td>16 GB</td>
</tr>
<tr>
<td>Model</td>
<td>Tesla C2075</td>
<td>Tesla K20C</td>
</tr>
<tr>
<td>Frequency of CUDA Cores</td>
<td>1.15 GHz</td>
<td>0.71 GHz</td>
</tr>
<tr>
<td>Number of CUDA cores</td>
<td>448</td>
<td>2496</td>
</tr>
<tr>
<td>Double Precision Floating Point Performance (Peak)</td>
<td>515 Gflops</td>
<td>1.17 Tflops</td>
</tr>
<tr>
<td>Single Precision Floating Point Performance (Peak)</td>
<td>1.03 Tflops</td>
<td>3.52 Tflops</td>
</tr>
<tr>
<td>Dedicated Memory</td>
<td>6 GB</td>
<td>5 GB</td>
</tr>
<tr>
<td>CUDA Compute capability</td>
<td>2.0</td>
<td>3.5</td>
</tr>
<tr>
<td>Software</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS</td>
<td>Windows 7 64 bit</td>
<td>Windows 7 64 bit</td>
</tr>
<tr>
<td>CUDA version</td>
<td>5.5</td>
<td>6.0</td>
</tr>
<tr>
<td>CULA library</td>
<td>R17</td>
<td>R17</td>
</tr>
<tr>
<td>MKL version</td>
<td>11.2</td>
<td>11.2</td>
</tr>
<tr>
<td>OpenMP</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>Compiler</td>
<td>Visual C++ 2010</td>
<td>Visual C++ 2010</td>
</tr>
</tbody>
</table>

For steps 1 and 2 in Algorithm 1, the main diagonal elements of \( u \) are \(- (K - 1)/2K\), and other elements of \( u \) are \(1/2K\). Since there are no dependencies among the elements, \( u \) is allocated on GPU global memory, and a kernel function \( u_{\text{kernel}} \) is designed to launch a block with the same size of \( u \), in which every thread is in charge of calculating one element of \( u \), and the parallelization is maximized. Both \( u \) and \( r \) are symmetric matrices; hence, we allocate them on global memory and use function \texttt{culaDeviceDsyev} (included in CULA) to perform the eigenvalue decomposition on the GPU. Moreover, the multiplication of the big matrix \( \tilde{A} \) and \( \tilde{A}^T \) is efficiently realized by using function \texttt{culaDeviceDgemm} in CULA.

For steps 3 and 8 in Algorithm 1, the calculations of \( s \) are the same. To reduce the amount of calculation in the iterative process as much as possible, the solution of \( s \) is decomposed into two steps. One is \( s = (1 \cdot \text{diag}(\lambda_r))^T \cdot \lambda_u \), which remains unchanged in the iteration process and can be calculated and stored outside the loop. Thus, step 8 only needs to compute the other one \( s = 1/(s_t - \beta 11^T) \). \( s_t \) and \( s \) are both stored on GPU global memory, and the temporary variables are stored on shared memory to minimized the latency of global memory access. A kernel function named \( s_{t\text{kernel}} \) is defined to compute \( s_t \) on the GPU. It launches a block with the same size of \( s \), in which every thread calculates one element of \( s_t \), i.e., \( s_{t(i,j)} = \sum \lambda_{r(i,i)} \cdot \lambda_{u(i,j)} \). By these designs, the iterative processes are reconstituted to maximize the parallel optimization. As Fig. 2 shows, the block size is set to BlockSize*BlockSize. Since subC\(=\)subA\(+\)subB, both subA and subB are segmented into several square matrices (e.g., subPartitionA, subPartitionB) of BlockSize*BlockSize, every time one block stores one subpartition matrix from both subA and subB on the shared memory of this block, and every thread in this block computes the multiplication of one element from subPartitionA and one element from subPartitionB, storing

parameters or determining the termination conditions in the loop iterative process, data are transferred between host and device; 3) at the end of the algorithm, we transfer data back from GPU to CPU and free the device memory when it is no longer needed. With the aforementioned issues in mind, a parallel implementation of LORSAL (LORSAL_P) has been developed, as illustrated in Fig. 1. In the following, we describe the most relevant steps of the parallelization and architecture-related optimizations carried out in the development of the GPU version of LORSAL_P algorithm. Motivated by the GPU hardware characteristics (listed in Table I), the kernel functions launch 32 \( \times \) 32 threads in each CUDA block by default, if not otherwise specified.
the results in subC. In this way, the shared local memory is efficiently used to save the global memory bandwidth. After that, a kernel function \( s_{\text{kernel}} \) is realized to compute \( s = 1/(s_1 - \beta \cdot 11^T) \), which starts a grid with the same size of \( s \), and each thread calculates one element of \( s \) on the GPU.

For steps 4 and 10, function cudaDeviceDgemm in CUDA is first invoked to compute \( w^T \cdot A \) on the GPU. Then, the rest of the calculations for \( p \) are encapsulated in a kernel function \( p_{\text{kernel}} \). \( p \) is allocated on GPU global memory. We organize the grid size according to the size of \( p \). In order to minimize the startup times of the kernel functions, \( Y - p \) in the calculation of \( d \) is merged into \( p_{\text{kernel}} \). After that, the remaining calculations of \( d \) and \( z \) are implemented by a kernel function \( z_{\text{kernel}} \), in which the computational data are transferred to shared memory to process, while the matrix multiplication operations are realized via CUDA function cudaDeviceDgemm. These optimizations combine the iterative steps of tight coupling together and realize the parallelization of the function level inside the iteration.

For step 5, the calculation of \( w \) is decomposed into \( w = \xi^T \cdot z \cdot \xi \) and \( w = s \cdot w \). The latter is defined as a kernel function \( w_{\text{kernel}} \), which transfers the computation data to shared memory, and launches a grid with the same size as \( w \). The rest of the computations are matrix multiplications, which can be efficiently realized by the function cudaDeviceDgemm.

For steps 6 and 7, since \( v \) and \( b \) have the same size, they are allocated on GPU global memory, and their calculations are encapsulated together in a kernel function \( \text{Soft}_{\text{kernel}} \), where a grid is created according to the matrix size, and each thread implements the computation of \( v = \text{soft}(w - b, \lambda/\beta) \) and \( b = b - (w - v) \) for a matrix element. The launch times of kernel functions are minimized by merged refactoring.

### IV. Performance Evaluation

Here, we assess the efficiency and performance of the proposed parallel implementation of the LORSAL algorithm (LORSAL_P) with two widely used hyperspectral images (Pavia University and Pavia Center) collected by the Reflective Optics System Imaging Spectrometer (ROSIS) in 2001. Both of them have 115 spectral bands, ranging from 430 to 860 nm, and a spatial resolution of 1.3 m per pixel. There are 103 bands after removing the 12 noisiest bands. The former image is centered at the University of Pavia (with a size of 610 \( \times \) 340 pixels), in the city of Pavia, Italy. The latter image was collected over the center of the Pavia city (with a size of 1096 \( \times \) 492 pixels). There are nine ground-truth classes in both scenes. In order to achieve a fair benchmark in terms of classification accuracy and execution performance with respect to the parallel versions, a C++ implementation of Algorithm 1 is realized as a basis for the subsequent parallel implementations. For the purpose of distinguishing the performance improvements between the parallel implementations on a multicore CPU platform and our considered GPU platforms, a multicore implementation of LORSAL (LORSAL_M) has also been carried out using OpenMP Application Program Interface (API) and the Intel Math Kernel Library (MKL) [14]. On the one hand, OpenMP is adopted to explicitly address multithreaded and shared-memory parallelism, in which we specify the regions corresponding to the kernels in GPU version for parallel implementation. On the other hand, the multithreading implementations of Basic Linear Algebra Subprograms (BLAS) in MKL library are invoked to realize the time-consuming operations of the big matrix calculations. Two different platforms are used to perform the tests (see Table I). The corresponding serial version (LORSAL_S) is executed on one core of the CPUs, and the multicore version (LORSAL_M) is run on all the cores of the two CPUs. All of the measurements obtained are achieved by the adopted classifiers after ten runs.

According to [6], the parameters of the classifier are set to \( \lambda = 0.001, \beta = 0.0001, \) and \( \delta = 0.8 \) (RBF kernel parameter) for the ROSIS Pavia Center image, and \( \delta = 0.45 \) for the ROSIS Pavia University image, respectively.

First of all, we test the classification accuracies of the three implementations, which are measured by the overall accuracy (OA), average accuracy (AA), and kappa statistic \((k)\) [15]. Forty labeled pixels are randomly chosen from each class as training samples, and the remaining labeled pixels are used as test samples. Although the training samples are randomly selected, we use exactly the same training-test sets for the three considered classifiers, in order to compare their accuracies. The classification accuracies are quantitatively shown in Tables II and III. These results indicate that the implementations of LORSAL_S, LORSAL_M, and LORSAL_P obtain exactly the same classification accuracies.

### TABLE II

**Classification Accuracy (%) for the Pavia Center Image**

<table>
<thead>
<tr>
<th>class</th>
<th>Training Samples</th>
<th>Test Samples</th>
<th>LORSAL_S</th>
<th>LORSAL_M</th>
<th>LORSAL_P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Water</td>
<td>40</td>
<td>65238</td>
<td>99.55</td>
<td>99.55</td>
<td>99.55</td>
</tr>
<tr>
<td>Trees</td>
<td>40</td>
<td>6467</td>
<td>90.85</td>
<td>90.85</td>
<td>90.85</td>
</tr>
<tr>
<td>Meadow</td>
<td>40</td>
<td>2851</td>
<td>95.12</td>
<td>95.12</td>
<td>95.12</td>
</tr>
<tr>
<td>Bricks</td>
<td>40</td>
<td>2100</td>
<td>91.99</td>
<td>91.99</td>
<td>91.99</td>
</tr>
<tr>
<td>Soil</td>
<td>40</td>
<td>6509</td>
<td>93.81</td>
<td>93.81</td>
<td>93.81</td>
</tr>
<tr>
<td>Asphalt</td>
<td>40</td>
<td>7485</td>
<td>95.90</td>
<td>95.90</td>
<td>95.90</td>
</tr>
<tr>
<td>Bitumen</td>
<td>40</td>
<td>7247</td>
<td>85.12</td>
<td>85.12</td>
<td>85.12</td>
</tr>
<tr>
<td>Tile</td>
<td>40</td>
<td>3082</td>
<td>98.93</td>
<td>98.93</td>
<td>98.93</td>
</tr>
<tr>
<td>Shadows</td>
<td>40</td>
<td>2125</td>
<td>99.92</td>
<td>99.92</td>
<td>99.92</td>
</tr>
<tr>
<td>OA (%)</td>
<td>360</td>
<td>103104</td>
<td>97.08</td>
<td>97.08</td>
<td>97.08</td>
</tr>
<tr>
<td>AA(%)</td>
<td>94.59</td>
<td>94.59</td>
<td>94.59</td>
<td>94.59</td>
<td>94.59</td>
</tr>
<tr>
<td>Kappa</td>
<td>94.98</td>
<td>94.98</td>
<td>94.98</td>
<td>94.98</td>
<td>94.98</td>
</tr>
</tbody>
</table>

### TABLE III

**Classification Accuracy (%) for the Pavia University Image**

<table>
<thead>
<tr>
<th>class</th>
<th>Training Samples</th>
<th>Test Samples</th>
<th>LORSAL_LS</th>
<th>LORSAL_M</th>
<th>LORSAL_LP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asphalt</td>
<td>40</td>
<td>6591</td>
<td>74.32</td>
<td>74.32</td>
<td>74.32</td>
</tr>
<tr>
<td>Meadows</td>
<td>40</td>
<td>18609</td>
<td>85.10</td>
<td>85.10</td>
<td>85.10</td>
</tr>
<tr>
<td>Gravel</td>
<td>40</td>
<td>2059</td>
<td>79.00</td>
<td>79.00</td>
<td>79.00</td>
</tr>
<tr>
<td>Trees</td>
<td>40</td>
<td>3024</td>
<td>91.43</td>
<td>91.43</td>
<td>91.43</td>
</tr>
<tr>
<td>Metal sheets</td>
<td>40</td>
<td>1305</td>
<td>98.28</td>
<td>98.28</td>
<td>98.28</td>
</tr>
<tr>
<td>Bare soil</td>
<td>40</td>
<td>4989</td>
<td>89.32</td>
<td>89.32</td>
<td>89.32</td>
</tr>
<tr>
<td>Bitumen</td>
<td>40</td>
<td>1290</td>
<td>91.64</td>
<td>91.64</td>
<td>91.64</td>
</tr>
<tr>
<td>Bricks</td>
<td>40</td>
<td>3642</td>
<td>82.86</td>
<td>82.86</td>
<td>82.86</td>
</tr>
<tr>
<td>Shadows</td>
<td>40</td>
<td>907</td>
<td>99.19</td>
<td>99.19</td>
<td>99.19</td>
</tr>
<tr>
<td>OA (%)</td>
<td>360</td>
<td>42416</td>
<td>84.82</td>
<td>84.82</td>
<td>84.82</td>
</tr>
<tr>
<td>AA(%)</td>
<td>87.80</td>
<td>87.80</td>
<td>87.80</td>
<td>87.80</td>
<td>87.80</td>
</tr>
<tr>
<td>Kappa</td>
<td>0.8055</td>
<td>0.8055</td>
<td>0.8055</td>
<td>0.8055</td>
<td>0.8055</td>
</tr>
</tbody>
</table>
Now, let us consider the Nvidia Tesla C2075, as an example, to theoretically analyze the computational performance. For the kernel functions, such as st_kernel, s_kernel, p_kernel, z_kernel, w_kernel, and Soft_kernel, the block size is set to be 32 × 32. Since every streaming multiprocessor in Tesla C2075 can launch at most 1536 threads, there is one block to be launched for every kernel. Therefore, the theoretical occupancies are 1024/1536 = 66.7%. In addition, the experimental results show that the practical occupancies are 66.4%, 66.4%, 66.5%, 66.5%, 66.4%, 66.4%, respectively.

Table IV provides an experimental evaluation in terms of computational performance. The LORSAL_P achieved significant speedups, with regard to the serial version LORSAL_S and also with regard to LORSAL_M, on both platforms. The data transfers between host and device were obtained after profiling the implementation using the CUDA visual profiler tool distributed by Nvidia. In all the cases, less than one-third of the total solver time is consumed by data transfers. The classification can be completed in less than 1 s. Taking into consideration that ROSIS is a pushbroom instrument with a maximum sampling rate of 62 Hz [16], one line of full-pixel vectors can be collected in 16 ms. This means that, if we would like to classify the Pavia Center scene (103 104 test samples) as fast as the data are acquired, we would need to do it in less than 3 s. Similarly, the Pavia University scene (42 416 test samples) would have to be classified in less than 1 s. This is our understanding of real-time processing in this work.

### Table IV

<table>
<thead>
<tr>
<th>Platform 1</th>
<th>Univ. Pavia</th>
<th>Pavia Center</th>
</tr>
</thead>
<tbody>
<tr>
<td>LORSAL_S</td>
<td>30.71</td>
<td>92.32</td>
</tr>
<tr>
<td>LORSAL_M</td>
<td>5.31</td>
<td>13.35</td>
</tr>
<tr>
<td>Total</td>
<td>0.6606</td>
<td>1.00</td>
</tr>
<tr>
<td>IO(Host to Device)</td>
<td>0.0551</td>
<td>(8.34%)</td>
</tr>
<tr>
<td>IO Device to Host</td>
<td>0.0099</td>
<td>(1.50%)</td>
</tr>
<tr>
<td>Speed up(X)</td>
<td>LORSAL_S/ LORSAL_P</td>
<td>46.49</td>
</tr>
<tr>
<td></td>
<td>LORSAL_M/ LORSAL_P</td>
<td>8.04</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Platform 2</th>
<th>Univ. Pavia</th>
<th>Pavia Center</th>
</tr>
</thead>
<tbody>
<tr>
<td>LORSAL_S</td>
<td>28.98</td>
<td>88.04</td>
</tr>
<tr>
<td>LORSAL_M</td>
<td>5.74</td>
<td>14.78</td>
</tr>
<tr>
<td>Total</td>
<td>0.4544</td>
<td>0.7898</td>
</tr>
<tr>
<td>IO(Host to Device)</td>
<td>0.0909</td>
<td>(20.00%)</td>
</tr>
<tr>
<td>IO Device to Host</td>
<td>0.0126</td>
<td>(2.77%)</td>
</tr>
<tr>
<td>Speed up(X)</td>
<td>LORSAL_S/ LORSAL_P</td>
<td>63.78</td>
</tr>
<tr>
<td></td>
<td>LORSAL_M/ LORSAL_P</td>
<td>12.63</td>
</tr>
</tbody>
</table>

### V. Conclusion

In this letter, a real-time implementation of the LORSAL algorithm for hyperspectral image classification has been presented. Experimental results show the effectiveness of the proposed GPU implementation, not only in terms of classification accuracy but also in terms of computational performance. The implementation achieved significant speedups compared to the serial and multicore versions, which are encouraging to employ this technique in time-critical scenarios.

### References