GPU Parallel Implementation of Spatially Adaptive Hyperspectral Image Classification

Zebin Wu, Member, IEEE, Linlin Shi, Jun Li, Member, IEEE, Qicong Wang, Le Sun, Member, IEEE, Zhihui Wei, Javier Plaza, Senior Member, IEEE, and Antonio Plaza, Fellow, IEEE

Abstract—Image classification is a very important tool for remotely sensed hyperspectral image processing. Techniques able to exploit the rich spectral information contained in the data, as well as its spatial-contextual information, have shown success in recent years. Due to the high dimensionality of hyperspectral data, spectral-spatial classification techniques are quite demanding from a computational viewpoint. In this paper, we present a computationally efficient parallel implementation for a spectral-spatial classification method based on spatially adaptive Markov random fields (MRFs). The method learns the spectral information from a sparse multinomial logistic regression classifier, and the spatial information is characterized by modeling the potential function associated with a weighted MRF as a spatially adaptive vector-tototal variation function. The parallel implementation has been carried out using commodity graphics processing units (GPUs) and the NVIDIA’s Compute Unified Device Architecture. It optimizes the work allocation and input/output transfers between the central processing unit and the GPU, taking full advantages of the computational power of GPUs as well as the high bandwidth and low latency of shared memory. As a result, the algorithm exploits the massively parallel nature of GPUs to achieve significant acceleration factors (higher than 70x) with regards to the serial and multicore versions of the same classifier on an NVIDIA Tesla K20C platform.

Index Terms—Graphics processing units (GPUs), hyperspectral image, parallel, sparse multinomial logistic regression (SMLR), spatially adaptive Markov random fields (MRFs), spectral-spatial classification.

I. INTRODUCTION

Hyperspectral image classification intends to assign the pixel vectors of a scene into a set of predefined classes. Different from panchromatic and multispectral remote sensing images, hyperspectral images contain hundreds of narrow spectral bands, spanning the visible to infrared spectrum, and exhibit a wealth of information in the spectral domain. In the literature, it has been shown that techniques able to exploit both the spectral and the spatial information contained in the scene represent successful approaches to hyperspectral image classification [1].

However, a remaining challenge is to take full advantage of the spectral and spatial information contained in the hyperspectral data. Since each pixel is given by a high-dimensional vector, supervised and semisupervised classification requires a sufficient number of training samples, which is difficult to satisfy in many remote sensing applications [2]. Recently, many (supervised and unsupervised) methods have been presented for hyperspectral image classification [1]–[3], among which those based on machine learning methods such as the support vector machine (SVM) [4] or the multinomial logistic regression (MLR) [5], [6] have been proven to be able to deal with limited training samples in a robust way.

Another important challenge in hyperspectral image classification is the extremely high dimensionality of hyperspectral data cubes. The dimensionality and volume of hyperspectral data is ever increasing, and high-speed classification is very important for applications with time-critical constraints, such as target detection for military purposes, monitoring of chemical contamination, and wildfire tracking. Fortunately, recent advances in high performance computing [7]–[15] have allowed for the exploitation of specialized hardware devices such as field-programmable gate arrays (FPGAs) [11], Beowulf clusters and distributed computers [12], multicore central processing units (CPUs) [13], and graphics processing units (GPUs) [13]–[17] in hyperspectral imaging applications. Specifically, it is possible to greatly accelerate the hyperspectral image processing on a GPU-based parallel computing platform, benefit from its capacity of performing compute-intensive, massively...
parallel computations [9]. For example, a GPU implementation of a sparse MLR (SMLR) classifier has been recently presented as one of the first techniques achieving real-time classification performance [7].

Nevertheless, the SMLR is a pixel-based classifier that only exploits the spectral information contained in the scene. To further improve the classification accuracy and robustness [18], the spatial-contextual information has been successfully included in many classification techniques, such as composite kernels (CK) [19], [20], graph kernels [21], morphological filters [22], [23], partitional clustering [24], [25], and joint sparse representation [26], [27]. Among these techniques, Markov random fields (MRFs) [28]–[31] have achieved great popularity and effectiveness in the task of incorporating the spatial information to spectral information under a Bayesian inferring framework.

High classification accuracy is extremely important for proper decision-making in many critical scenarios [3], including, for instance, precision agriculture, urban planning, or military reconnaissance. Take into consideration that neighboring pixels in natural scenes usually comprise materials with similar spectral characteristics, especially in homogeneous regions, many techniques have included the spatial-contextual information in addition to spectral-based analysis in order to improve the classification accuracy and robustness [18]. This is, for instance, the case of techniques such as CK [19], [20], graph kernels [21], morphological filters [22], [23], partitional clustering [24], [25], and joint sparse representation [26], [27]. Among these techniques, MRFs [28]–[31] have achieved great popularity and effectiveness in the task of incorporating the spatial information to spectral information under a Bayesian inference framework.

In order to further exploit the spatial structure and contextual information to improve the classification accuracy in hyperspectral image classification, Sun et al. [32] proposed a novel spectral-spatial approach based on spatially adaptive MRFs. In this approach, the spectral information is learnt by an SMLR classifier, and the spatial information is characterized by modeling the potential function associated with a weighted MRF as a spatially adaptive vector total variation function, which is defined on the real-valued hidden marginal probabilities of the posterior distribution, where the weights are calculated by the gradients of the original hyperspectral image to model the spatial structure of the original data. This classifier has the potential to outperform other spectral-spatial approaches, as shown in [32]. Meanwhile, the utilization of spatial information leads to a significant computational burden, and its execution is computationally too expensive to achieve real-time performance, which compromises its application in time-critical scenarios.

In this paper, we develop an efficient parallel implementation of a spatial-spectral classification method based on spatially adaptive MRFs, implemented on commodity GPUs. The proposed parallel implementation has been developed using NVIDIA’s Compute Unified Device Architecture (CUDA), as well as the cuFFT library. It optimizes the work allocation and input/output (I/O) transfers between the CPU and the GPU, taking full advantages of the computational power of GPUs as well as the high bandwidth and low latency of shared memory. Both classification accuracy and computational performance are evaluated, using two different GPU platforms by NVIDIA: Tesla C2075 and Tesla K20C. The experimental results, conducted on two real hyperspectral images, reveal remarkable acceleration factors while retaining exactly the same classification accuracy achieved by the corresponding serial and multiversion cores.

The remainder of this paper is organized as follows. Section II briefly describes the spatially adaptive hyperspectral image classification method based on weighted MRFs. Section III describes its GPU parallelization in detail. Section IV evaluates the proposed parallel implementation in terms of both classification accuracy and computational performance. Section V concludes with some remarks and hints at plausible future research lines.

II. HYPSPECTRAL IMAGE CLASSIFICATION BASED ON SPATIALLY ADAPTIVE MRFs

Let us assume that \( \mathbf{x} = [x_1, x_2, \ldots, x_N] \in \mathbb{R}^{L \times N} \) is a hyperspectral image with \( N \) pixels and \( L \) bands (features), \( \mathcal{S} \equiv \{1, 2, \ldots, N\} \) denotes the set of the indexes of the \( N \) pixels, and \( \mathbf{y} = [y_1, y_2, \ldots, y_N] \in \mathbb{R}^N \) is an image of class labels, where \( K \equiv \{1, 2, \ldots, K\} \) denotes a set of \( K \) class labels, \( x_i \in \mathbb{R}^L \) is an \( L \)-dimensional hyperspectral pixel observation, and each \( y_i = [y_{i}^{(1)}, y_{i}^{(2)}, \ldots, y_{i}^{(K)}] \) denotes a “1-of-\( K \)” encoding of the \( K \) classes \( \{y_{i}^{(j)}\in \{1, 0\}, \text{for } j \in K\} \).

In a Bayesian framework, hyperspectral classification can be interpreted as estimating \( \mathbf{y} \) given the observed image \( \mathbf{x} \). The maximum a posteriori (MAP) classification [32], [33] can be modeled as

\[
\hat{y}_i = \arg \max_{y_i \in \mathcal{K}} \log p(y_i | x_i)
\]

\[
= \arg \max_{y_i \in \mathcal{K}} \left\{ \sum_{i=1}^{N} \left( \log p(y_i | x_i) - \log p(y_i) \right) + \log p(y_i) \right\}.
\]

The densities \( p(y_i | x_i) \) are learnt by a probabilistic classifier, and the density \( p(y_i) \) is usually modeled to impose a spatial prior on the labels \( y_i \). The classification task can be defined by the MAP marginal solution [34],

\[
\hat{y}_i \equiv \max_{k} q_{i}^{(k)} = \max_{k} p(y_i = k | x_j, j \in N_i), i \in \mathcal{S}.
\]

In our previous work [32], we used SMLR [35] to model \( p(y_i | x_i) \) as a spectral data fidelity term, and modeled the spatial prior on the implicit marginal probability of the posterior distribution \( q \), instead of using a Gibbs distribution on the discrete labels \( y \).

Let us denote by \( \mathbf{p}_C \equiv [p_1, p_2, \ldots, p_N] \in \mathbb{R}^{K \times N} \) the probability matrix obtained through the SMLR, where \( p_i = [p_i^{(1)}, p_i^{(2)}, \ldots, p_i^{(K)}]^T \), and let us denote by \( \mathbf{q} = [q_1, q_2, \ldots, q_N] \in \mathbb{R}^{K \times N} \) the implicit marginal matrix, where \( q_i = [q_i^{(1)}, q_i^{(2)}, \ldots, q_i^{(K)}]^T \). Since the SMLR classifier does not utilize the spatial-contextual information, it often leads to classification outliers in homogeneous areas. Here we model the SMLR outliers as additive Gaussian white noise, and define the posterior probability \( p(q | \mathbf{p}_C) \) to infer \( q \) from \( \mathbf{p}_C \) as follows:

\[
q = \arg \max_{q} p(q | \mathbf{p}_C) = \arg \max_{q} \frac{p(\mathbf{p}_C | q)p(q)}{p(\mathbf{p}_C)}.
\]
whose solution is equivalent to
\[
q = \arg \max_q p(q|p_C) = \arg \min_q (-\log p(p_C|q) - \log p(q)).
\]

(4)

Bearing in mind that modeling the spatial structure is important for classification purposes [36], a spatially adaptive MRF [32] is used here to model the spatial prior term, taking full advantage of the spatial-contextual information of the hyperspectral image as follows:

\[
p(q) = \frac{1}{Z} \exp \left\{ -\mu_s \cdot a(x_i) \sum_{|i-j| < \delta} \| q_i - q_j \|_1 \right\}
\]

(5)

where \( Z \) is a normalization constant for the density, \( \mu_s \) tunes the degree of homogeneity of each region in the hyperspectral image, \( |i - j| < \delta \) indicates that pixel \( i \) and \( j \) are a pair of neighbors in spatial sense, and \( a(x_i) \) is a spatially adaptive regularization parameter imposing the edge structure information and adjusting the power of the spatial smoothness in different pixel locations as follows:

\[
a(x_i) = \frac{1}{1 + \sum_{j=1}^{L} \left( \| \nabla_h x^j_i \|_2^2 + \| \nabla_v x^j_i \|_2^2 \right)}
\]

(6)

where \( \nabla_h x^j_i \) and \( \nabla_v x^j_i \) are the horizontal and vertical first-order gradients of \( x_i \) at the \( j \)th band. Moreover, the true labels of the training samples used for the learning stage can be fixed as an additional constraint to characterize their spatial structure by spreading the class information to their neighbors. The probability distribution \( q \) must be nonnegative, and its columns must be sum-to-1. Then, the proposed hyperspectral image classification model based on spatially adaptive MRFs can be summarized as follows:

\[
\hat{q} = \arg \min_q \left\{ \| q - p_C \|_F^2 + \mu_s \cdot a(x_i) \sum_{|i-j| < \delta} \| q_i - q_j \|_1 \right\}
\]

s.t. \( q \geq 0, q_{\Lambda_i} = y_{\Lambda_i}, 1^T q_i = 1, i = 1, 2, \ldots, N \)

(7)

where \( \| \cdot \|_F \) denotes the Frobenius norm, and \( \Lambda_i \) is the index set of training samples.

By defining \( Hq = \left[ H_h q, H_v q \right] = \sum_{|i-j| < \delta} q_i - q_j \|_1 \) and \( \lambda = \mu_s \cdot a(x_i) \), the model can be transformed into

\[
\hat{q} = \arg \min_q \frac{1}{2} \| q - p_C \|_F^2 + \lambda Hq
\]

s.t. \( q \geq 0, q_{\Lambda_i} = y_{\Lambda_i}, 1^T q_i = 1, i = 1, 2, \ldots, N \)

(8)

where \( \| \cdot \|_1 \) denotes the \( l_1 \) norm, \( q_i \) is the \( i \)-th column of \( q \) corresponding to pixel \( i \), \( \delta \) controls the size of the neighborhood, and \( H \) is a convolution operator. \( H_h \) and \( H_v \) denote linear operators computing the horizontal and vertical differences, respectively, between the components of \( q \) corresponding to neighboring pixels [37]. Let us take \( H_h \) for example, \( H_h q = [d_1, d_2, \ldots, d_N] \), where \( d_i = q_i - q_{i_0} \), \( i \) and \( i_0 \) denote a pixel and its horizontal neighbor. By introducing variables \( V_1, V_2, V_3, V_4, \) and \( V_5 \), model (8) can be rewritten as

\[
\hat{q} = \arg \min_{V_1, V_2, V_3, V_4, V_5} \left\{ \frac{1}{2} \| V_1 - p_C \|_F^2 + \lambda \| V_5 \|_1,1 \right\}
\]

+ \lambda \| V_5 \|_1,1 \right\}
\]

(9)

s.t. \( V_1 = q V_2 = q V_3 = q V_4 = q V_5 = HV_1 \)

where \( \| \cdot \|_1,1 = \sum_{i=1}^{N} \| x_i \|_1, x_i \) is the \( i \)-th column of \( x \), and

\[
l_S(x) = \begin{cases} 0, & x \in S \\ +\infty, & x \notin S \end{cases}
\]

At this point, the alternating direction method of multipliers [38], [39] can be utilized to efficiently solve model (9). In terms of the probability matrix \( p_C \), we use the logistic regression via variable splitting and augmented Lagrangian (LORSAL) algorithm [40] to solve the SMLR model. A detailed algorithmic description of our spatially adaptive hyperspectral image classification algorithm based on weighted MRF (referred to hereinafter as SAHIC) is given in Algorithm 1.

III. GPU PARALLEL IMPLEMENTATION BASED ON CUDA

The proposed parallel implementation has been developed following the philosophy that the most time-consuming operations are always carried out by GPU, and that it is particularly important to optimize the memory allocation and to minimize the I/O transfers between the CPU (host) and the GPU (device).

First, we analyzed Algorithm 1 in detail to determine the most time-consuming parts that should be implemented in parallel. After analyzing Algorithm 1, the following observations are in place.

1) Algorithm 1 requires serial iterative loop executions for calculating the implicit marginal matrix \( q \). In every loop, \( D_1, \ldots, D_5, V_1, \ldots, V_5, \) \( q \) need to be updated, each of which is a big matrix of size \( K \times N \).

2) The algorithm involves many big matrix operations. Every read-write and arithmetic operation involving these big matrices is very time consuming on the CPU, and the multistep iterative process increases the computational burden even more. As a result, it is necessary to particularly optimize the operations relevant to these big matrices on the GPU.

3) The calculation of \( V_5 \) is the most time-consuming step with computational complexity of \( O(LN \log N) \), and the convolution operations in steps 2.7 and 2.8 are the most expensive parts (computationally) of the algorithm. Therefore, we mainly concentrate on the efficient parallelization of these operations in our GPU implementation.

With the aforementioned issues in mind, a GPU parallel implementation of SAHIC (SAHIC_P) has been designed as illustrated by the flowchart in Fig. 1.
Algorithm 1: Serial version of SAHIC (SAHIC_S).

**Input:** Training samples set $A \in \mathbb{R}^{K \times J}$, class labels of training samples $Y_A \in \mathbb{R}^K$, test samples set $X \in \mathbb{R}^{L \times N}$

**Initialization:** Set $\lambda > 0, \lambda_C > 0, \beta > 0, \mu > 0, t = 0, M = \text{MaxIteration}$, $\tilde{A} = h(A)$, $\tilde{X} = h(X)$, $h(x)$ is the radial basis function (RBF), initialize $V_1^{(0)}, V_2^{(0)}, V_3^{(0)}$, $V_4^{(0)}, V_5^{(0)}, D_1^{(0)}, D_2^{(0)}, D_3^{(0)}, D_4^{(0)}, D_5^{(0)}$

**Step 1.** $P_C = \text{LORSAL} (\tilde{A}, Y_A, \tilde{X}, \lambda_C, \beta)$

**Step 2.** Calculate $q$

Do:

**Step 2.1.**

$$q^{(t+1)} = \frac{1}{4} (V_1^{(t)} + D_1^{(t)} + V_2^{(t)} + D_2^{(t)} + V_3^{(t)} + D_3^{(t)})$$

$$q^{(t+1)}_{n_i} = Y_{n_i}$$

**Step 2.2.** $s = q^{(t+1)} - D_3^{(t)}$

**Step 2.3.** $b_{(j)} = \left( 1 - \frac{K}{\sum_{(i,j)} s_{(i,j)}} \right) / K, j \in \{1, \ldots, N\}$

**Step 2.4.** $V_1^{(t+1)} = \frac{1}{\lambda} p_C + \mu q^{(t+1)} - D_1^{(t)}$

**Step 2.5.** $V_2^{(t+1)} = \max (q^{(t+1)} - D_2^{(t)}, 0)$

**Step 2.6.** $V_3^{(t+1)} = s + 1 \cdot b, b = [1, \ldots, 1]^T$

$$a = q^{(t+1)} - D_4^{(t)}$$

**Step 2.7.** $V_4^{(t+1)} = (H^T H + I)^{-1}$

$$a + H^T (V_5^{(t)} + D_5^{(t)})$$

**Step 2.8.** $V_5^{(t+1)} = \text{soft}(D_5^{(t)} - HV_4^{(t+1)}, \lambda / \mu_s)$

**Step 2.9.** Update multipliers

$$D_1^{(t+1)} = D_1^{(t)} - (q^{(t+1)} - V_1^{(t+1)})$$

$$D_2^{(t+1)} = D_2^{(t)} - (q^{(t+1)} - V_2^{(t+1)})$$

$$D_3^{(t+1)} = D_3^{(t)} - (q^{(t+1)} - V_3^{(t+1)})$$

$$D_4^{(t+1)} = D_4^{(t)} - (q^{(t+1)} - V_4^{(t+1)})$$

$$D_5^{(t+1)} = D_5^{(t)} - (HV_4^{(t+1)} - V_5^{(t+1)})$$

**Step 2.10.** $t = t + 1$

While $t > M$

Step 3. $[q_{\text{max}}, Y_x] = \text{max} (q), q_{\text{max}} \in \mathbb{R}^{L \times N}$ is a vector that consists of the maximum entry of each column in $q$, $Y_x \in \mathbb{R}^{L \times N}$ is the corresponding row subscripts of the maximum entries.

**Output:** $Y_x$, the class labels of training samples $X$

In the following, we describe the optimizations related to low level architecture and the most relevant parallelization steps conducted in the GPU for developing the computationally efficient parallel implementation of SAHIC in Algorithm 1.

### A. Optimization of the Memory Allocation and I/O Transfer

First and foremost, we need to properly arrange the data storage and I/O communication between the host (CPU) and the device (GPU), to minimize the cost of data transfers. The training samples and test samples are stored by columns, and then manually transferred from the CPU to the local GPU memory after initialization. During the parallel optimization process, the data is stored in the GPU memory as much as possible, and the storage space for the intermediate variables of the iterative process is allocated in advance. By taking full advantage of the shared memory to achieve an efficient interaction with low latency, we maximize the memory bandwidth and optimizing accesses. In addition, the I/O communication between the host and the device mainly takes place when updating parameters or determining the termination conditions in the loop iterative process. After the iterative process is completed, the data will be transferred back from device to host, and the device memory will be set free when it is no longer needed.

### B. Parallel Optimization for Calculating $V_1, V_2,$ and $V_3$

Once the data set is loaded into the GPU memory, we execute the GPU version of the variable splitting and augmented Lagrangian algorithm for sparse multinomial logistic regression (LORSAL_P) in [7] to obtain the probability matrix $P_C$. After that, we define a CUDA kernel function called $q_kernel$, configured with a grid that consists of as many threads as the size of $q$. Here every thread is responsible for calculating an element of $q$.

To fix the true labels of the training samples, a kernel $fixq_kernel$ is realized to perform the execution of $q^{(t+1)}_{n_i} = Y_{n_i}$.

After that, we encapsulate the calculation of $s$ and $b$ into a single CUDA kernel function $nu_{aux3}\_sum$ kernel (see Fig. 2), and optimize the calculation of $b$ by means of a column batch sum. The kernel function $nu_{aux3}\_sum$ kernel is designed to launch $N$ blocks ($n = (N + ThreadNum - 1) / ThreadNum$), and each block includes $ThreadNum$ threads and a shared memory with the size of $ThreadNum$ and data type of Double. Bearing in mind that the shared memory in the device has much lower latency, higher bandwidth, and smaller memory size than the global memory, we develop a strategy to perform the column batch sum to efficiently calculate the $b$, as Fig. 3 shows, making full use of the shared memory.

Taking into consideration that the calculation of $V_1, V_2, V_3$ are loosely coupled and involve matrices of the same dimensionality, and that there are no data dependences among the matrix elements, steps 2.4, 2.5, and 2.6 in Algorithm 1 are merged and encapsulated into a single CUDA kernel function $Vs\_kernel$ to minimize the startup times of the kernel functions. This kernel launches as many threads as elements in $V_1$, and each thread performs the calculations for one matrix element, thus making better use of the intrinsic concurrency of the CUDA blocks.

### C. Parallel Optimization for Calculating $V_4, V_5$

The calculation of $V_4$ and $V_5$ is very time-consuming, and this represents a bottleneck of the algorithm. First of all, we compute $L = (H^T H + I)^{-1}$ and store it outside the loop, since its
Fig. 1. Flowchart of the proposed GPU parallel implementation (SAHIC_P).

Fig. 2. CUDA codes for kernel nu_aux3_sum_kernel.

Fig. 3. Diagram that illustrates our efficient strategy to perform the column batch sum of a matrix.

computation remains unchanged in the iterative process. As a result, we define a kernel function named IL_kernel to compute $L$, which launches a block with the same size of $H$ on the GPU, and every thread is in charge of calculating one element of $L$, taking full advantages of parallelism of the GPU device.

After that, we further analyze the steps 2.7 and 2.8. Since $H$ acts only on the spatial domain, it can be independently handled in band-by-band fashion. For each band, we need to perform a convolution operation, which can be solved by means of a discrete Fourier transform (DFT) diagonalization. Therefore, it is crucial to efficiently implement the DFT on the GPU. In this paper, we accomplish this important step using the cuFFT library [41] in the CUDA Toolkit, which provides a simple interface for computing fast Fourier transforms (FFTs) on GPUs [42]. By employing the cuFFT library and applying the configuration mechanism plan, the transform is optimized for the particular GPU hardware. Then, the actual transform takes place following the plan of execution by calling the execution function. This is a key aspect to the performance of our GPU parallel implementation. It is worth noting that, by the plan interface, it is very easy to reuse the thread configurations and GPU resources for different kinds of FFTs. After the configuration of the plan, the function cufftExec* is invoked to conduct the FFT calculation in the GPU. Since the convolution operation on each band is carried out independently, the FFT calculation is configured by means of a cufftPlanMany function, for which the size of the batch is set to the number of bands (i.e., $K$) in the original hyperspectral image, and every branch performs the two-dimensional Fourier transform with the same scale (i.e., the spatial size of the image), thus leveraging the parallelism of the GPUs and the floating-point power.

Then, a CUDA kernel called V4_kernel is implemented to compute in parallel the operations of matrix addition and subtraction (see Fig. 4). A kernel function called V5_kernel is launched next (see Fig. 5), where the number of threads equals the size of the matrix $V_5$, and each thread implements the operation $V_5(t+1) = \text{softmax}((D_5^{(t)} - H V_4(t+1)) \lambda/\mu)$ and the updating of the multipliers for a matrix element. In this way, we can minimize the launch times of the kernel functions by merged refactoring.

The parallel algorithm now repeats from step 2.1 to 2.10 until a maximal number of iterations is reached. We implement the remaining operations in the CPU, since they have much lower computation costs, and can be realized very quickly in the CPU without the need for parallelization.
Fig. 4. CUDA codes for kernel V4_kernel.

```
__global__ void V4_kernel(cuDoubleComplex* mid, double *Vmat3,
int VmatCols, cuDoubleComplex* inner, int rows, int cols, int k)
{
    __shared__ double block[BLOCK_SIZE][BLOCK_SIZE + 1];
    int rowld = blockIdx.y * blockDim.y + threadIdx.y;
    int colld = blockIdx.x * blockDim.x + threadIdx.x;
    if (rowld < rows & & colld < cols)
    {
        int idx = rowld * cols + colld;
        int size = rows * cols;
        block[idx][0] = size;
        block[idx][1] = inner[idx].x;
        mid[idx] = temp;
        mid[idx + size] = temp;
    }
    __syncthreads();
    rowld = blockIdx.y * blockDim.y + threadIdx.y;
    colld = blockIdx.x * blockDim.x + threadIdx.x;
    if (rowld < rows & & colld < cols)
    {
        int idx = rowld * cols + colld;
        Vmat3[idx] = block[threadIdx.x][threadIdx.y];
    }
    __syncthreads();
}
```

Fig. 5. CUDA codes for kernel V5_kernel.

```
__global__ void V5_kernel(double *VcellMat1, double *VcellMat2,
double *DcellMat1, double *DcellMat2, cuDoubleComplex* mid, double param, int rows, int cols)
{
    int idx = rowld * cols + colld;
    int size = rows * cols;
    int mididx = idx;
    double auxx = mid[mididx] / size;
    double auxx = mid[mididx] / size;
    double DcellMat1 = DcellMat1[idx];
    double DcellMat2 = DcellMat2[idx];
    double mid1 = auxx - DcellMat1;
    double mid2 = auxx - DcellMat2;
    double temp = max(sqrt(mid1 * mid1 + mid2 * mid2) - param, 0.0);
    VcellMat[0] = mid1 * (auxx - DcellMat1);
    VcellMat[1] = mid1 * (auxx - DcellMat2);
    VcellMat[2] = mid2 * (auxx - VcellMat[1][idx]);
    VcellMat[2] = mid2 * (auxx - VcellMat[1][idx]);
}
```

Fig. 6. Detailed illustration of our GPU parallel hyperspectral image classification algorithm based on spatially adaptive MRFs (SAHIC_P).

**IV. EXPERIMENTAL RESULTS**

**A. GPU Platforms and Experimental Datasets**

In order to evaluate the performance of our GPU parallel algorithm on different high performance computing architectures, two GPU platforms of different CUDA Compute Capacities (i.e., NVIDIA Tesla C2075 and Tesla K20C) have been used in our experiments. The hardware specifications and computing capabilities of these GPUs and the corresponding CPU platforms, as well as the software specifications, are described in Tables I and II, respectively.

Two widely used hyperspectral images collected by two different imaging spectrometers: the Airborne Visible Infrared Imaging Spectrometer (AVIRIS) and the Reflective Optics Spectrographic Imaging System (ROSIS), are used to assess the proposed algorithm. A summary of these images, respectively, collected over the Indian Pines region in NW Indiana USA and the Urban area of Pavia University, Pavia, Italy, is given in Table III, as well as in Figs. 7 and 8.

**B. Performance Evaluation**

In our experiments, we focus on evaluating both the classification accuracy and the computational performance of our GPU-based SAHIC algorithm, as compared with the corresponding serial version and a multicore version. The serial ver-
Algorithm 2: GPU parallel version of SAHIC (SAHIC_P).

Input: Training samples set $A \in R^{L\times J}$, class labels of training samples $Y_A \in R^{N \times J}$, test samples set $X \in R^{L \times N}$.

Initialization: Set $\lambda > 0$, $\lambda_C > 0$, $\beta > 0$, $\mu > 0$, $t = 0$, $M = \text{MaxIteration}$, $\tilde{A} = h(A)$, $\tilde{X} = h(X)$, $h(x)$ is the radial basis function (RBF), initialize $V_1^{(0)}$, $V_2^{(0)}$, $V_3^{(0)}$, $V_4^{(0)}$, $V_5^{(0)}$, $D_1^{(0)}$, $D_2^{(0)}$, $D_3^{(0)}$, $D_4^{(0)}$, $D_5^{(0)}$.

Step 1. Copy data from host to device.

Step 2. Invoke $LORSAL_P(A, Y_A, \tilde{A}, \lambda_C, \beta)$ to calculate $p_C$.

Step 3. Calculate $q$ on GPU

Step 3.1. Invoke $ILP_{\text{kernel}}$ to compute $L$.

Do:

Step 3.2. Invoke $q_{\text{kernel}}$ to calculate $q$, and invoke $\text{fixq}_{\text{kernel}}$ to perform the execution of $q^{(t+1)} = Y_{h_l}$.

Step 3.3. Invoke $nu_{\text{aux3 sum kernel}}$ to calculate $s$ and $b$ by means of column batch sum.

Step 3.4. Invoke $V_s_{\text{kernel}}$ to consolidate the calculation of $V_1$, $V_2$, $V_3$.

Step 3.5. Invoke $V_4_{\text{kernel}}$ to compute in parallel the operations of matrix addition and subtraction in calculating $V_4$.

Step 3.6. Perform the DFT on the GPU by employing the cuFFT library to calculate the $HV_4^{(t+1)}$.

Step 3.7. Invoke $V_5_{\text{kernel}}$ to perform the operation $V_5^{(t+1)} = \text{soft}(D_5^{(t)} - HV_4^{(t+1)}, \lambda/\mu)$, and update the multipliers.

While $t > M$

Step 4. Copy $q$ from device to host.

Step 5. Compute $[q_{\text{max}}, Y_x] = \max(q)$ on CPU.

Output: $Y_X$, the class labels of training samples $X$.

| TABLE I | HARDWARE SPECIFICATIONS AND COMPUTING CAPABILITIES OF THE CONSIDERED PLATFORMS |
|------------------|---------------------------------|---------------------------------|
| Specification | NVIDIA Tesla C2075 Platform | NVIDIA Tesla K20C Platform |
| Processor Number | Intel Xeon | Intel Xeon |
| CPU Processor Base Frequency | 2.4 GHz | 2.10 GHz |
| Number of Cores | 8 in total | 24 in total |
| Main Memory | 32 GB | 16 GB |
| Architecture | Fermi | Kepler |
| Frequency of CUDA Cores | 1150 MHz | 706 MHz |
| Number of CUDA Cores | 448 | 2496 |
| Double Precision Floating Point Performance (Peak) | 0.515 Tflops | 1.17 Tflops |
| GPU Single Precision Floating Point Performance (Peak) | 1.03 Tflops | 3.52 Tflops |
| Dedicated Memory | 6 GB | 5 GB |
| Memory Interface | 384-bit | 320-bit |
| Memory Bandwidth | 144 GB/sec | 208 GB/sec |
| CUDA Compute Capability (version) | 2.0 | 3.5 |

TABLE II
SOFTWARE SPECIFICATIONS OF THE CONSIDERED PLATFORMS

<table>
<thead>
<tr>
<th>Specification</th>
<th>OS</th>
<th>CUDA version</th>
<th>CUDA library</th>
<th>MKL version</th>
<th>OpenMP</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA Tesla C2075 Platform</td>
<td>Windows 7 64bit</td>
<td>5.5</td>
<td>R17</td>
<td>11.2</td>
<td>2.0</td>
<td>Visual C++ 2010</td>
</tr>
<tr>
<td>NVIDIA Tesla K20C Platform</td>
<td>Windows 7 64bit</td>
<td>6.0</td>
<td>R17</td>
<td>11.2</td>
<td>2.0</td>
<td>Visual C++ 2010</td>
</tr>
</tbody>
</table>

TABLE III
DETAILS OF THE TWO EXPERIMENTAL HYPERSPECTRAL IMAGES USED IN OUR EXPERIMENTS

<table>
<thead>
<tr>
<th>Scenes</th>
<th>Indian Pines Dataset</th>
<th>Pavia University Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruments</td>
<td>AVIRIS</td>
<td>ROSIS</td>
</tr>
<tr>
<td>Image spatial size</td>
<td>145x145</td>
<td>610x340</td>
</tr>
<tr>
<td>Spectral bands</td>
<td>200 (220 in total, and 20 noise and water absorption bands are removed)</td>
<td>103 (115 in total, and 12 noise bands are removed)</td>
</tr>
<tr>
<td>Spectral resolution</td>
<td>10 nm</td>
<td>4 nm</td>
</tr>
<tr>
<td>Spatial resolution</td>
<td>20 m</td>
<td>1.3 m</td>
</tr>
<tr>
<td>Ground-truth classes</td>
<td>16 Details are shown in Fig. 7(b)</td>
<td>9 Details are shown in Fig. 8(b)</td>
</tr>
<tr>
<td>Labeled pixels</td>
<td>10366</td>
<td>43923</td>
</tr>
</tbody>
</table>

Fig. 7. AVIRIS Indian Pines hyperspectral dataset. (a) False color composition. (b) Ground truth as a collection of mutually exclusive classes.
1) Overall accuracy (OA), computed as the ratio between the correctly classified test samples and the total number of test samples.

2) Average accuracy (AA), the mean of the accuracies across the different classes.

3) Kappa statistic (Kappa), computed by weighting the measured accuracies. It incorporates both of the diagonal and off-diagonal entries of the confusion matrix and is a robust measure in terms of the degree of agreement.

According to Sun et al. [32] and our repeated experiments, the parameters were empirically set to $\lambda_C = 0.001$, $\beta = 0.0001$, $\mu = 0.05$, $\mu_s = 2$, $\delta = 0.8$ (RBF kernel parameter), and $\lambda = 2$ for the AVIRIS Indian Pines scene, and $\delta = 0.35$ and $\lambda = 1$ for ROSIS Pavia University image. For each value reported in experiments, ten Monte Carlo runs were performed and the average values were reported.

We first evaluated the classification performance using the ROSIS Pavia University dataset. We randomly chose 40 labeled pixels from each class as training samples, and used the remaining labeled pixels as test samples. Moreover, we used exactly the same training-test sets for the three considered versions of SAHIC when compared the achieved classification accuracies in a fair way. Table IV summarizes the classification accuracies (OA, AA, and Kappa), measured after processing the ROSIS Pavia University dataset on the two considered platforms. Some of the obtained classification maps are given in Fig. 9. It is worth noting that the proposed SAHIC_S, SAHIC_M, and SAHIC_P obtain exactly the same classification accuracies (when using exactly the same training and test sets) on the considered platforms, leading to very smooth maps of classification as depicted in Fig. 9. We find that the time they need to complete their calculations is the only difference between the serial and parallel versions. The corresponding timing results (in seconds), and the acceleration factors (speedups) are shown in Table V.

It is obvious from Table V that the parallel version SAHIC_P achieves remarkable acceleration factors on both platforms as compared to both the serial and multicore versions. This is because the SAHIC_P takes full advantages of the computational power of GPUs, the high bandwidth, as well as low latency of shared memory, and benefits from exploiting the massively parallel nature of GPUs. Specifically, the SAHIC_P achieves high speedups of about $81 \times$ and $121 \times$, respectively, with regards to the serial version in the two considered GPU platforms.

Now, we evaluate the performance of the considered classifiers using the AVIRIS Indian Pines dataset. In this test, we randomly chose nearly 10% of the labeled pixels of each class (1043 pixels in total) as training samples, and used the remaining labeled pixels as test samples. For illustrative purposes, Fig. 10 shows some of the classification maps and Table VI reports the
It shows that our proposed parallel implementation makes a significant improvement with regard to the versions of SAHIC_S and SAHIC_M. The best speedup obtained in our experiments is above 70 × (achieved on the NVIDIA Tesla K20C platform). These results exhibit the potential of GPUs for parallelizing the considered spectral-spatial classification algorithm.

To further evaluate the efficiency of the proposed parallel approach, we perform the tests on the Tesla K20C Platform, and compare the computation time of kernel functions in our GPU implementation with their corresponding serial versions. As can be seen from Table VIII, most of the kernel functions achieve remarkable acceleration factors, especially for the most time-consuming step of the calculation of $\mathbf{V}_5$, which is ideal in terms of parallel efficiency.

Let us now take a closer look at the performance of the parallel spatial-spectral classifier SAHIC_P as compared to the parallel spectral-only classifier LORSAL_P in [7]. For simplicity, we present an illustrative example using the AVIRIS Indian Pines dataset. Fig. 11 compares the classification accuracies and the computational performance of these two classifiers. It can be concluded from Fig. 9 that the proposed SAHIC_P greatly improves the classification accuracy with some additional computational burden, as expected. In particular, as noted by Tables VI and VII, the total processing time of SAHIC_P is less than 5 s on the NVIDIA Tesla K20C, and its accuracy reaches up to 98.32 %, which is very compelling for an efficient hyperspectral image classification method.

Fig. 12 shows the percentages of the data transfer time between the host and the device for the proposed parallel method, which is also an important issue for GPU parallelization. It can

### Table VI

**Classification Accuracies (%) Obtained for the AVIRIS Indian Pines Dataset**

<table>
<thead>
<tr>
<th>Class</th>
<th>Class Name</th>
<th>Training Samples</th>
<th>Test Samples</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alfalfa</td>
<td>6</td>
<td>48</td>
<td>91.67</td>
</tr>
<tr>
<td>2</td>
<td>Corn-no till</td>
<td>144</td>
<td>1290</td>
<td>97.05</td>
</tr>
<tr>
<td>3</td>
<td>Corn-min till</td>
<td>84</td>
<td>750</td>
<td>98.4</td>
</tr>
<tr>
<td>4</td>
<td>Corn</td>
<td>24</td>
<td>210</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>Grass/Forest</td>
<td>50</td>
<td>447</td>
<td>95.3</td>
</tr>
<tr>
<td>6</td>
<td>Grass/Trees</td>
<td>75</td>
<td>672</td>
<td>99.11</td>
</tr>
<tr>
<td>7</td>
<td>Grass/Pasture-mowed</td>
<td>3</td>
<td>23</td>
<td>78.26</td>
</tr>
<tr>
<td>8</td>
<td>Hay-windrowed</td>
<td>49</td>
<td>440</td>
<td>100</td>
</tr>
<tr>
<td>9</td>
<td>Oats</td>
<td>2</td>
<td>18</td>
<td>16.67</td>
</tr>
<tr>
<td>10</td>
<td>Soybeans-no till</td>
<td>97</td>
<td>871</td>
<td>97.7</td>
</tr>
<tr>
<td>11</td>
<td>Soybeans-min till</td>
<td>247</td>
<td>2221</td>
<td>99.77</td>
</tr>
<tr>
<td>12</td>
<td>Soybeans-clean till</td>
<td>62</td>
<td>552</td>
<td>99.82</td>
</tr>
<tr>
<td>13</td>
<td>Wheat</td>
<td>22</td>
<td>190</td>
<td>100</td>
</tr>
<tr>
<td>14</td>
<td>Woods</td>
<td>130</td>
<td>1164</td>
<td>100</td>
</tr>
<tr>
<td>15</td>
<td>Building-Grass-Trees-Drives</td>
<td>38</td>
<td>342</td>
<td>97.66</td>
</tr>
<tr>
<td>16</td>
<td>Stone-steel</td>
<td>10</td>
<td>85</td>
<td>74.12</td>
</tr>
<tr>
<td></td>
<td>Towers</td>
<td>1043</td>
<td>9323</td>
<td>98.32</td>
</tr>
<tr>
<td></td>
<td>QA (%)</td>
<td></td>
<td></td>
<td>90.35</td>
</tr>
<tr>
<td></td>
<td>Kappa</td>
<td></td>
<td></td>
<td>0.9808</td>
</tr>
</tbody>
</table>

### Table VII

**Execution Times and Acceleration Factors of the Kernel Functions**

<table>
<thead>
<tr>
<th>Kernels</th>
<th>Pavia University Dataset</th>
<th>Indian Pines Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial Time(s)</td>
<td>Parallel Time(s)</td>
</tr>
<tr>
<td>q_kernel</td>
<td>0.07368</td>
<td>0.00002</td>
</tr>
<tr>
<td>fix_kernel</td>
<td>0.00012</td>
<td>0.00003</td>
</tr>
<tr>
<td>nu_aux3_sum_kernel</td>
<td>0.00692</td>
<td>0.00003</td>
</tr>
<tr>
<td>V5_kernel</td>
<td>0.00672</td>
<td>0.00004</td>
</tr>
<tr>
<td>V4_kernel</td>
<td>0.02709</td>
<td>0.00006</td>
</tr>
<tr>
<td>V3_kernel</td>
<td>0.08223</td>
<td>0.00017</td>
</tr>
</tbody>
</table>

It is worth noting that our proposed GPU parallel implementation is efficient on different GPU computing architectures (for instance, Fermi and Kepler), and is scalable with different CUDA compute capabilities. Specifically, even with the CUDA compute capabilities of 2.0, the SAHIC_P can achieve significant speedup with regards to the serial version in the Tesla C2075 platform.
be observed from Fig. 12 that, in our GPU parallel implementation, the data transfer time is relatively low. In particular, our implementation uses more than 99% of the total GPU time for executing the kernels and less than 1.0% of the time for memory transfers in all cases. Therefore, for SAHIC_P, the most significant portion of the time is taken by pure computation steps, which can be defined as compute-bound. We emphasize that the performance of the algorithm is driven by the processing units themselves (not by data transfers) and, as a result, the algorithm is expected to scale very effectively for larger size problems.

Several conclusions can be obtained from the experiments reported in this section. First and foremost, our parallel SAHIC_P produces good results in terms of both classification accuracy and computational performance. The algorithm achieves significant acceleration factors in the two considered GPU architectures, while in all experiments the classification accuracies achieved by the parallel algorithm were shown to be exactly the same to those obtained by the serial and multicore versions of SAHIC. Additionally, the SAHIC_P offered significant improvements (in terms of classification accuracy and parallel performance) with regards to the GPU implementation of a spectral-based classifier (LORSAL_P). This fact results from the inclusion of spatial-contextual information, which is also accurately modeled and parallelized in the GPU. We believe that these performance improvements for the considered spatial-spectral classifier SAHIC can potentially enhance its applicability in classification problems with time-critical constraints, thus enhancing proper decision-making in these scenarios.

V. CONCLUSION AND FUTURE WORK

The incorporation of spatial-contextual information to spectral-based characterization is crucial for hyperspectral image classification. Despite the availability of many techniques
for spectral-spatial classification in the hyperspectral imaging literature, very few techniques have been effectively implemented in parallel for their application in time-critical scenarios. In this paper, we develop a new GPU parallel implementation of a spatial-spectral hyperspectral image classification method based on spatially adaptive MRFs, taking full advantage of the computational power offered by commodity GPUs. Our experimental assessment, conducted using two GPU platforms, demonstrated the superior performance of the proposed spectral-spatial classifier in terms of both classification accuracy and computational performance, using two widely used benchmark hyperspectral scenes. Specifically, our experiments demonstrate that the proposed method is more accurate and faster than several other available techniques for hyperspectral image classification, including the serial and multicore versions of the same algorithm. In future works, we will explore other parallelization frameworks for spectral-spatial classifiers using hardware platforms with lower power consumption ratios, such as FPGAs.

REFERENCES


Zebin Wu (M’13) was born in Zhejiang, China, in 1981. He received the B.Sc. and Ph.D. degrees in 2003 and 2007, respectively, all in computer science and technology, from Nanjing University of Science and Technology.

He is currently a Professor in the School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, China. He was a Visiting Scholar in the Department of Mathematics, University of California, Los Angeles, CA, USA, from August 2016 to September 2017. From June 2014 to June 2015, he was a Visiting Scholar in the Hyperspectral Computing Laboratory, Department of Technology of Computers and Communications, Escuela Politcnica, University of Extremadura, Cceres, Spain. His research interests include hyperspectral image processing, high performance computing, and computer simulation.

Qicong Wang was born in Henan, China, in 1988. He received the B.Sc. degree in computer science and technology from the School of Computer, Henan University of Science and Technology, Luoyang, China, in 2012. He is currently working toward the M.S. degree in computer application, with the School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, China. His research interests include hyperspectral image classification and high performance computing.

Le Sun (M’13) was born in Jiangsu, China, in 1987. He received the B.Sc. degree in applied mathematics from the School of Science, Nanjing University of Science and Technology, Nanjing, China, in 2009, where he is currently working toward the Ph.D. degree in pattern recognition, with the School of Computer Science and Engineering.

His research interests include the area of spectral unmixing, hyperspectral classification, image processing, sparse representation, and compressive sensing.
Zhihui Wei was born in Jiangsu, China, in 1963. He received the B.Sc. and M.Sc. degrees in applied mathematics and the Ph.D. degree in communication and information system all from the South East University, Nanjing, China, in 1983, 1986, and 2003, respectively.

He is currently a Professor and Doctoral Supervisor in Nanjing University of Science and Technology, Nanjing, China. His main research interests include partial differential equations, image processing, multiscale analysis, sparse representation, and compressed sensing.

Javier Plaza (M’09–SM’15) received the M.Sc. and Ph.D. degrees in computer engineering from the University of Extremadura, Cáceres, Spain, in 2004 and 2008, respectively.

He is currently a Member of the Hyperspectral Computing Laboratory, Department of Technology of Computers and Communications, University of Extremadura. He has authored more than 120 publications, including 36 JCR journal papers, ten book chapters, and 80 peer-reviewed conference proceeding papers. He has guest edited two special issues on hyperspectral remote sensing for different journals. His main research interests include hyperspectral data processing and parallel computing of remote sensing data.

Dr. Plaza is an Associate Editor for the IEEE GEOSCIENCE AND REMOTE SENSING LETTERS and an Associate Editor of the IEEE REMOTE SENSING CODE LIBRARY. He received the Outstanding Ph.D. Dissertation Award from the University of Extremadura in 2008. He received the Best Column Award of the IEEE SIGNAL PROCESSING MAGAZINE in 2015, the 2013 Best Paper Award of the JSTARS journal, and the most highly cited paper (during 2005–2010) in the Journal of Parallel and Distributed Computing. He received the Best Paper Awards in the IEEE International Conference on Space Technology and the IEEE Symposium on Signal Processing and Information Technology.

Antonio Plaza (M’05–SM’07–F’15) received the M.Sc. and Ph.D. degrees both in computer engineering from the University of Extremadura, Cáceres, Spain, in 1999 and 2002, respectively.

He is the Head of the Hyperspectral Computing Laboratory, Department of Technology of Computers and Communications, University of Extremadura, Cáceres, Spain. His main research interests include comprise hyperspectral data processing and parallel computing of remote sensing data. He has authored more than 600 publications, including 197 JCR journal papers (more than 140 in IEEE journals), 23 book chapters, and 285 peer-reviewed conference proceeding papers. He has guest edited ten special issues on hyperspectral remote sensing for different journals.

Prof. Plaza is a Fellow of the IEEE “for contributions to hyperspectral data processing and parallel computing of Earth observation data.” He received the Best Reviewers of the IEEE GEOSCIENCE AND REMOTE SENSING LETTERS (in 2009) and the Best Reviewers of the IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING (in 2010), for which he served as Associate Editor during 2007–2012. He is also an Associate Editor for IEEE ACCESS, and was a member of the Editorial Board of the IEEE GEOSCIENCE AND REMOTE SENSING NEWSLETTER (during 2011–2012) and the IEEE GEOSCIENCE AND REMOTE SENSING MAGAZINE (in 2013). He was also a member of the steering committee of the IEEE JOURNAL OF SELECTED TOPICS IN APPLIED EARTH OBSERVATIONS AND REMOTE SENSING (JSTARS). He received the Best Column Award of the IEEE SIGNAL PROCESSING MAGAZINE in 2015, the 2013 Best Paper Award of the JSTARS journal, and the most highly cited paper (during 2005–2010) in the Journal of Parallel and Distributed Computing. He received the Best Paper Awards in the IEEE International Conference on Space Technology and the IEEE Symposium on Signal Processing and Information Technology. He served as the Director of Education Activities for the IEEE Geoscience and Remote Sensing Society (GRSS) during 2011–2012, and as a President of the Spanish Chapter of IEEE GRSS during 2012–2016. He has reviewed more than 500 manuscripts for more than 50 different journals. He is currently serving as the Editor-in-Chief of the IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING.