GPU IMPLEMENTATION OF HYPERSONSPECTRAL IMAGE CLASSIFICATION BASED ON WEIGHTED MARKOV RANDOM FIELDS

Zebin Wu\(^1\), Qicong Wang\(^1\), Antonio Plaza\(^2\), Jun Li\(^3\), Jie Wei\(^1\), and Zhihui Wei\(^1\)

\(^1\)School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing 210094, China; \(^2\)Department of Technology of Computers and Communications, University of Extremadura, Cáceres E-10003, Spain; \(^3\)School of Geography and Planning, Sun Yat-sen University, Guangzhou, 510275, China.

ABSTRACT

The dimensionality of hyperspectral data is very high, and spectral-spatial hyperspectral classification techniques are quite demanding from a computational viewpoint. In this paper, we present a computationally efficient implementation of a spectral-spatial classification method based on weighted Markov random fields. The method learns the spectral information from a sparse multinomial logistic regression (SMLR) classifier, and the spatial information is characterized by modeling the potential function associated with a weighted Markov random field (MRF) as a spatially adaptive vector total variation function. The parallel implementation has been carried out using commodity graphics processing units (GPUs) and the NVIDIA’s compute unified device architecture (CUDA), thus exploiting the massively parallel nature of GPUs to achieve significant acceleration factors with regards to the serial version of the same classifier on an NVIDIA Tesla C2075 platform.

Index Terms—Hyperspectral imaging, spectral-spatial classification, sparse multinomial logistic regression (SMLR), Markov Random Fields (MRFs), graphics processing units (GPUs).

1. INTRODUCTION

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].

In order to efficiently exploit the spatial structure and contextual information to improve the classification accuracy in hyperspectral image classification, in [2] we proposed a novel spectral-spatial approach based on weighted Markov random fields. In this approach, the spectral information is learnt by a 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 (TV) 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 in the original data. This classifier has the potential to outperform other spectral-spatial approaches, as shown in [2]. Meanwhile, the utilization of spatial information leads to a significant computational burden, and its execution is computationally very expensive and far from real-time performance, which compromises its application in time-critical scenarios.

Fortunately, GPU-based parallel computing offers a tremendous potential to bridge the gap towards real-time analysis of hyperspectral images, owing to its capacity to perform compute-intensive, massively parallel computations [3].

In this paper, we develop an efficient implementation of a spectral-spatial classification method based on weighted MRFs on commodity GPUs. The proposed parallel method has been developed using NVIDIA’s compute unified device architecture (CUDA), as well as the cuFFT library, taking full advantage of the computational power of GPUs. We conduct an evaluation of the method not only in terms of classification accuracy, but also in terms of computational performance, using a GPU platform by NVIDIA: Tesla C2075. The experimental results, conducted on real hyperspectral images, reveal remarkable acceleration factors while retaining exactly the same classification accuracy achieved by the corresponding serial version.

2. CLASSIFICATION BASED ON WEIGHTED MRFS

Let us assume that \( x = [x_1, x_2, \ldots, x_N] \in \mathbb{R}^{N \times L} \) is a hyperspectral image with \( N \) pixels of \( L \) bands (features), \( S = \{1, 2, \ldots, N\} \) denotes the set of indexes of the \( N \) pixels, and \( y = [y_1, y_2, \ldots, y_N] \in \mathbb{K}^N \) is an image of class labels, where \( \mathbb{K} = \{1, 2, \ldots, K\} \) 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 (\( y_i^{(j)} \in \{0,1\}, \text{for } j \in K \)).

By introducing a spectral data fidelity term learned from the sparse multinomial logistic regression (SMLR) [4] classifier and a weighted Markov random field (MRF) prior in the hidden field, the spatial-spectral hyperspectral image classification model can be modeled as follows [2]:

\[
\hat{q} = \arg \min_q \left\{ \left\| q - p_c \right\|_2 + \mu_i \cdot a(x_i) \sum_{j=0} \left\| q_j - q_j^i \right\| \right\} \quad (1)
\]

s.t. \( q \geq 0, q_{\lambda_i} = y_{\lambda_i}, 1^T q = 1, i = 1, 2, \ldots, N', \)

where \( p_c = [p_1, p_2, \ldots, p_N] \in R^{K \times N} \) is the probability matrix obtained through the SMLR, \( p_i = [p_i^{(1)}, p_i^{(2)}, \ldots, p_i^{(K)}] \), \( p_i^{(k)} \) is the probability that a given sample \( x_i \) belongs to class \( K \), and \( q = [q_1, q_2, \ldots, q_N] \in R^{K \times N} \) is the implicit marginal matrix, where \( q_i = [q_i^{(1)}, q_i^{(2)}, \ldots, q_i^{(K)}] \), and \( \left\| \cdot \right\| \) denotes the Frobenius norm, and \( \Lambda_i \) is the index set of training samples. The parameter \( \mu_i \) tunes the degree of homogeneity of each region in the image, 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:

\[
a(x_i) = \frac{1}{1 + \sum_{j=1}^N (\nabla^x x_j^i)^2 + (\nabla^y x_j^i)^2 \right)} \quad (2)
\]

where \( \nabla^x x_j^i \) and \( \nabla^y x_j^i \) are the horizontal and vertical first-order gradients of \( x_i \) at the \( j \)th band.

By defining \( Hq = \left[ H_1 q \ H_2 q \ \ldots \ H_N q \right] \), \( \lambda = \mu_i \cdot a(x_i) \), and introducing variables \( V_1, V_2, V_3, V_4, \) and \( V_5 \), model (1) can be rewritten as:

\[
\hat{q} = \arg \min_{q_1, q_2, q_3, q_4, q_5} \left\{ \frac{1}{2} \left\| V_1 - p_c \right\|_2 + \lambda \left( \left\| V_2 \right\|_1 + l_{\delta}(V_3) + l_{\delta}(V_4) \right) \right\} \quad (3)
\]

s.t. \( V_1 = q, V_2 = q, V_3 = q, V_4 = q, V_5 = HV_4 \),

where \( H \) is a convolution operator, \( q_i \) is the \( i \)-th column of \( q \) corresponding to pixel \( i \), \( \delta \) controls the size of the neighborhood, \( \left\| \cdot \right\|_1 = \sum_{j=1}^5 \left\| x_j^i \right\| \), \( x_i \) is the \( i \)-th column of \( x \), and \( l_{\delta}(x) = \left\{ \begin{array}{ll} 0, & x \in S \\ +\infty, & x \notin S \end{array} \right. \)

Then, the spatial-spectral classification model (3) can be efficiently solved by the alternating direction method of multipliers (ADMM) [5, 6] as follows.

---

**Algorithm 1** Serial algorithm of hyperspectral classification based on weighted MRFS (HCWMRFS)

**Input:** Training samples set \( A \in R^{L \times N}, \) class labels of training samples \( Y_i \in R^{K \times N}, \) test samples set \( X \in R^{K \times N} \)

**Initialization:** Set \( \lambda > 0, \lambda_\epsilon > 0, \beta > 0, \mu > 0, t = 0, \)

\( M=\text{MaxIteration}, \tilde{A} = h(A), \tilde{X} = h(X), \ h(x) \) 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.** \( p_c = \text{LORSAL}(\tilde{A}, Y_i, \tilde{X}, \lambda_\epsilon, \beta) \)

**Step 2.** Calculate \( q \)

**Do:**

**Step 2.1.**

\( q^{(0)} = \frac{1}{4} (V_1^{(0)} + D_1^{(0)} + V_2^{(0)} + D_2^{(0)} + V_3^{(0)} + D_3^{(0)} + V_4^{(0)} + D_4^{(0)}) \)

\( q_i^{(0)} = y_i \)

**Step 2.2.**

\( V_1^{(s+1)} = \frac{1}{1 + \mu} (p_c + \mu (q^{(s)} - D_1^{(s)})) \)

**Step 2.3.**

\( q^{(s+1)} = \text{max}(q^{(s)} - D_2^{(s)}, 0) \)

**Step 2.4.**

\( s = q^{(s)} - D_2^{(s)} \)

**Step 2.5.**

\( b_{i,j} = (1 - \sum_{t=1}^K s_{i,t}) / K, j \in (1, \ldots, N) \)

**Step 2.6.**

\( V_3^{(s+1)} = s + I \cdot b, I = [1, 1, \ldots, 1]^T \)

**Step 2.7.**

\( V_4^{(s+1)} = (H^T H + I)^{-1} (q^{(s)} - D_3^{(s)} + H^T (V_5^{(s)} + D_5^{(s)})) \)

**Step 2.8.**

\( V_5^{(s+1)} = \text{soft}(D_3^{(s)} - HV_4^{(s)}, \lambda / \mu_\epsilon) \)

**Step 2.9.** Update multipliers

\( D_1^{(s+1)} = D_1^{(s)} - (q^{(s)} - V_1^{(s+1)}) \)

\( D_2^{(s+1)} = D_2^{(s)} - (q^{(s)} - V_2^{(s+1)}) \)

\( D_3^{(s+1)} = D_3^{(s)} - (q^{(s)} - V_3^{(s+1)}) \)

\( D_4^{(s+1)} = D_4^{(s)} - (q^{(s)} - V_4^{(s+1)}) \)

\( D_5^{(s+1)} = D_5^{(s)} - (HV_4^{(s)} - V_5^{(s+1)}) \)

**Step 2.10.** \( t = t + 1 \)

While \( t > M \)

**Step 3.** \( [q_{\text{max}}, Y_i] = \text{max}(q), q_{\text{max}} \in R^{K \times N} \) is a vector that consists of the maximum entry of each column in \( q \).

\( Y_i \in R^{K \times N} \) is the corresponding row subscripts of the maximum entries.

**Output:** \( Y_i \), the Class labels of training samples \( X \).

---

**3. GPU IMPLEMENTATION**

The proposed parallel implementation (HCWMRFS_P) has been developed following the philosophy that the most
time-consuming operations are always carried out by GPU. In the following, we describe the most relevant parallelization steps and the architecture-related optimizations conducted in the GPU for developing a computationally efficient implementation of HCWMRF, as Fig.1 shows.

(1) First and foremost, we need to properly arrange the data storage and input/output (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, we store the data in the GPU memory as much as possible, and allocate the storage space for the intermediate variables of the iterative process in advance, thus maximizing global 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, we transfer the data back from GPU to CPU and free the device memory when it is no longer needed.

(2) 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 \( q_{\text{kernel}} \), 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 training samples, a kernel \( fixq_{\text{kernel}} \) is realized to perform the execution of \( q^{(t)}_v = Y_v \).

(3) Taking into consideration that the calculation of \( V_1 \), \( V_2 \), \( s \) are loosely coupled and involve matrices of the same dimensionality, and that there are no data dependences among the matrix elements, steps 2.2, 2.3 and 2.4 in Algorithm1 are merged and encapsulated into a single CUDA kernel function \( VS_{\text{kernel}} \) to minimize the startup times of the kernel functions. This kernel launches as many threads as elements in \( V_j \). Then, we define the CUDA kernel functions \( b_{\text{kernel}} \) and \( V3_{\text{kernel}} \) to carry out the computation of steps 2.5 and 2.6 in Algorithm 1, respectively. The former kernel launches \( J \) threads, and each of them sums a column of \( s \) and gets one element of \( b \) using the GPU. The latter kernel launches a \( K \times N \) dimensional grid of threads on the GPU, where each thread performs the calculations for one matrix element.

(4) The calculation of \( V_4 \) and \( V_5 \) is very time-consuming, and this represents a bottleneck of the algorithm. However, since \( H \) acts only on the spatial domain, it can be independently handled in band by band fashion. As a result, for each band, we need to perform a convolution operation, which can be solved by means of discrete Fourier transform (DFT) diagonalization. As a result, it is crucial to efficiently implement the DFT on the GPU. In this paper, we accomplish this important step by means of the cuFFT library\(^2\), which is one of the fine-tuned libraries for NVIDIA GPUs which is included in the CUDA Toolkit\(^3\), providing a simple interface for computing fast Fourier transforms (FFTs) on GPUs. This allows users to quickly leverage the floating-point power and parallelism of the GPU in a highly optimized and tested FFT library. Specifically, a simple configuration mechanism called a plan in cuFFT is applied to optimize the transform for the given configuration and the particular GPU hardware selected by using internal building blocks. Then, the actual transform takes place following the plan of execution by calling the execution function. Therefore, once the user creates a plan, the library retains whatever state is needed to execute the plan multiple times, without the need to recalculate the configuration. This is a key aspect to the performance of our GPU parallel implementation. Although different kinds of FFTs require different thread configurations and GPU resources, the plan interface provides a simple way of reusing configurations. After the configuration of the plan, the function \text{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 \text{cufftPlanMany} function.

---

\(^2\) http://docs.nvidia.com/cuda/pdf/CUFFT_Library.pdf

\(^3\) https://developer.nvidia.com/cuda-toolkit
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).

(5) After that, a CUDA kernel called $V_4$ kernel is implemented to compute in parallel the operations of matrix addition and subtraction. Then, a kernel function called Soft_kernel is launched, where the number of threads equals the size of the matrix $V_5$, and each thread implements the operation $V_5(t) = \text{soft}(D(t) - HV_4(t), \lambda / \mu)$ for a matrix element. The updating of the multipliers is accomplished in the GPU by means of a specific CUDA kernel function, called D_kernel. In this way, we can minimize the launch times of the kernel functions by merged refactoring.

(6) The parallel algorithm now repeats from step 2.1 to 2.10 until a maximal number of iterations is reached. The remaining operations are implemented in the CPU since they have much lower computation costs, and can be realized very quickly in the CPU without the need for parallelization.

4. EXPERIMENTAL RESULTS

The experimental platform used to evaluate the proposed algorithm is the NVIDIA Tesla C2075, which features 448 processor cores operating at clock frequency of 1150MHz, with total dedicated memory of 6 GB, 1500MHz memory (with 384-bit GDDR5 interface) and memory bandwidth of 144 GB/s. The GPU is connected to two Intel Xeon E5-2609 CPUs at 2.4 GHz with 4 cores (8 cores in total), with 32GB RAM. The real hyperspectral datasets used for tests, are the AVIRIS Indian Pines dataset (Dataset1) and the ROSIS Pavia University dataset (Dataset2) [8]. The former has 16 ground-truth classes of interest, while nearly 10% of the labeled pixels were used as test samples. The latter has 9 classes, with 95.36% of the labeled pixels were used as test samples, and the remaining labeled pixels were used as test samples.

In our experiments, we focus on evaluating both the classification accuracy and the computational performance of our GPU-based HCWMRF_P algorithm, as compared with a corresponding serial version. According to [2] and our repeated experiments, the parameters were empirically set to $\lambda_c = 0.001$, $\beta = 0.0001$, $\mu = 0.05$, $\mu_c = 2$, $\delta = 0.35$ (RBF kernel parameter) and $\gamma = 1$. For each value reported in experiments, ten Monte Carlo runs were performed and the mean values are reported.

Table 1 summarizes the classification accuracies, the timing results, and the acceleration factors (speedups) measured after processing the datasets on the considered platforms. It is obvious from Table 1 that the HCWMRF_S, and HCWMRF_P obtain exactly the same classification accuracies on the considered platforms, leading to very smooth classification maps.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Training</th>
<th>Test</th>
<th>HCWMRF_S</th>
<th>HCWMRF_P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OA (%)</td>
<td>OA (%)</td>
<td>Time (sec)</td>
<td>Acceleration factor (X)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>95.36</td>
<td>1043.03</td>
<td>51.05</td>
</tr>
<tr>
<td>Dataset2</td>
<td>98.32</td>
<td>98.32</td>
<td>0.9808</td>
<td>0.9808</td>
</tr>
<tr>
<td>Dataset2</td>
<td>96.36</td>
<td>96.36</td>
<td>201.14</td>
<td>3.94</td>
</tr>
<tr>
<td>Dataset2</td>
<td>20.20</td>
<td>20.20</td>
<td>1431.03</td>
<td>70.84</td>
</tr>
</tbody>
</table>

It is worth noting that the HCWMRF_P achieves remarkable acceleration factors, as compared to the serial version. This is because the HCWMRF_P benefits from an efficient exploration of the parallel capacities of the GPU, as well as from the utilization of the highly efficient GPU-accelerated CUDA Toolkit. Specifically, the parallel version achieved significant acceleration factors of more than 51x, with regards to the serial version in the considered GPU platforms.

5. REFERENCES