Efficient Classification of Hyperspectral Images on Commodity GPUs using ELM-based Techniques

Javier López-Fandiño, Dora B. Heras, Francisco Argüello

Abstract—Hyperspectral image processing algorithms are computationally very costly, which makes them good candidates for parallel and, specifically, GPU processing. Extreme Learning Machine (ELM) is a recently proposed classification algorithm very suitable for its implementation on GPU platforms. In this paper we propose an efficient GPU implementation of an ELM-based classification strategy for hyperspectral images. ELM can be expressed in terms of matrix operations that can take maximum advantage of the GPU architecture. Regarding the classification accuracy, the proposed algorithm achieves competitive results as compared to a traditional SVM. In this paper we present an optimized CUDA (Compute Unified Device Architecture) GPU implementation of an ELM-based algorithm to classify hyperspectral datasets in real time. We also discuss different approaches in order to combine the classifiers through a majority vote mechanism, in order to improve accuracy results. The implementation keys are the exploitation of the thousands of threads available in the GPU architecture and the adequate use of the memory hierarchy. The GPU algorithm is formulated in terms of matrix operations that are efficiently executed in blocks by an optimized linear algebra library.

The next sections of this paper are organized as follows: Sect. 2.1 explains the ELM mechanism to classify hyperspectral images, while the majority vote mechanism is presented in Sect. 2.2. We introduce some GPU and CUDA fundamentals in Sect. 3. The GPU implementations are described in Sect. 4. Sect. 5 is devoted to the discussion of the experimental results. Finally, Sect. 6 summarizes the conclusions of this work.

2. Hyperspectral Image Classification Using ELM-based Techniques

In this section, we present the ELM and V-ELM algorithms, whose GPU implementation will be studied in Sect. 4.

2.1 ELM-based Classification

The raw pixel-wise ELM algorithm was proposed as an efficient learning algorithm for single-hidden layer feedforward neural networks (SLFNs) [8].

The output function of a SLFN with $L$ hidden nodes and $m$ output nodes, and being $x$ the input vector (see Fig.1) can be written as

$$f_L(x) = \sum_{i=1}^{L} \beta_i G(a_i, b_i, x), \quad x \in \mathbb{R}^d, \quad \beta_i \in \mathbb{R}^m,$$

where $a_i$ and $b_i$ are the input and output weights, respectively, $G(a_i, b_i, x)$ is the activation function, and $\beta_i$ are the output weights.
where $G(a_i, b_i, x)$ denotes the output function of the $i$th hidden node, being $a_i, b_i$ the hidden node parameters and $\beta_i$ the weight vector connecting the $i$th hidden node to the output nodes. For the case of additive nodes with activation function $g$, it can be expressed as

$$G(a_i, b_i, x) = g(a_i \cdot x + b_i), \quad a_i \in \mathbb{R}^d, \quad b_i \in \mathbb{R}, \quad (2)$$

A SLFN with $L$ hidden nodes can approximate $N$ arbitrary distinct samples and targets $(x_i, t_i) \in \mathbb{R}^d \times \mathbb{R}^m$, if the following equation system can be solved:

$$H\beta = T \quad (3)$$

where

$$H = \begin{bmatrix} h(x_1) \\ \vdots \\ h(x_N) \end{bmatrix} = \begin{bmatrix} G(a_1, b_1, x_1) & \ldots & G(a_L, b_L, x_1) \\ \vdots & \ddots & \vdots \\ G(a_1, b_1, x_N) & \ldots & G(a_L, b_L, x_N) \end{bmatrix}_{N \times L} \quad (4)$$

$$\beta = \begin{bmatrix} \beta_1^T \\ \vdots \\ \beta_L^T \end{bmatrix}_{L \times m}, \quad \text{and} \quad T = \begin{bmatrix} t_1^T \\ \vdots \\ t_N^T \end{bmatrix}_{N \times m} \quad (5)$$

$H$ is called the hidden layer output matrix of the neural network. Huang et al. [9], [4] have proven that a SLFN with randomly generated additive or RBF nodes in the hidden layer can universally approximate any continuous target function over any compact subset $\chi \subset \mathbb{R}^d$. For the case of additive nodes, the activation function $g$ can be any infinitely differentiable function, including sigmoidal functions, and also the radial basis, sine, cosine and exponential functions among others.

Once they are randomly generated, hidden node parameters $(a_i, b_i)$ remain fixed and training a SLFN is equivalent to finding a least-squares solution $\beta$ of the linear system $H\beta = T$, i.e.:

$$\beta = H^+T, \quad (6)$$

where $H^+$ is the Moore-Penrose generalized inverse of matrix $H$ [10].

So, ELM can be summarized as follows [9], [4]:

**Algorithm ELM**: Given a training set $\{(x_i, t_i)|x_i \in \mathbb{R}^d, t_i \in \mathbb{R}^m, i = 1, \ldots, N\}$, hidden node output function $G(a_i, b_i, x)$, and hidden node number $L$,

1) Randomly generate hidden node parameters $(a_i, b_i), \quad i = 1, \ldots, L$ where $a_i$ and $b_i$ are the input weight and bias values.
2) Calculate the hidden layer output matrix $H$.
3) Calculate the output weight vector, $\beta = H^+T$.

As it was stated in [11], ELM requires less human intervention than SVM because a single parameter, the number of neurons in the hidden layer, needs to be optimized, since all the other parameters are randomly initialized. It also achieves similar or better generalization performance for binary and multiclass classification cases. In addition, ELM has better scalability and it runs at much faster learning speed than SVM.

In our hyperspectral image case, each training sample represents a random selected pixel on the image and each component of the sample a spectral band of the pixel. The output that we obtain after the classification phase is the predicted class for each pixel of the image.

### 2.2 Voting-based ELM

In this work we follow an approach based on ensembles [7], [12], i.e., computing a number of independent ELMs with the same number of hidden nodes and the same activation function in each hidden node and combining the results obtained by the different classifiers. The individual ELMs are trained with the same dataset and the learning parameters of each ELM are randomly initialized independently.

Majority vote is the simplest method to implement among all the combination methods because it does not assume prior knowledge of the behaviour of the individual classifiers and it does not require training [13]. We use a democratic majority vote method where each classifier vote counts equal to the others and the final decision for each sample is the most repeated vote for the sample.

### 3. CUDA GPU Programming Model

Nowadays, commodity GPUs provide massively parallel processing capabilities based on their data parallel architecture. CUDA is a hardware/software platform that enables NVIDIA GPUs to execute programs invoking parallel functions called kernels that execute across many parallel threads [14]. These threads are organized into blocks so that each thread executes an instance of the kernel following a SIMD programming model. The blocks are arranged in a grid that is mapped to a hierarchy of CUDA cores in the GPU.

A streaming multiprocessor (called SM) contains plenty of CUDA cores and executes one or more thread blocks. The threads are executed in groups of 32 threads called warps. If all the threads in a warp execute the same code and access memory with nearby addresses the performance will be greatly improved.
Threads can access data from multiple memory spaces. First, each thread has a private local memory and registers. Each block of threads has a shared memory visible exclusively to the threads within the block with a lifetime that is equal to the block lifetime. Finally, all threads access the same global memory space (DRAM) which is persistent across kernel launches by the same application. The lower the memory level, the faster the read/write access to the data. Shared memory lifetime makes it difficult to share data among thread blocks because it implies the use of global memory whose access is slower than shared memory access.

The GPU architecture named Kepler includes a two level cache hierarchy. There are 64 KB of on-chip memory for each SM, which can be configured as half each for the shared memory and the L1 cache, 48 KB of shared memory and 16 KB of L1 cache or vice versa. There is also a unified L2 cache of 1536 KB that is shared among all the SM units. In the most recent Maxwell architecture [14] there are 64 KB of dedicated shared memory since the L1 cache is placed together with the texture memory.

Different performance optimization strategies have been applied in this work in order to optimize the computational performance:

- **Maximize the parallel execution.** It is important to organize the algorithm in computational blocks that can be independently executed minimizing communications among them.
- **Improve the efficiency in the use of the memory hierarchy.** Trying to maximize both the spatial and temporal locality in the data access, thus reducing data movement among different memory levels. It is essential to perform the maximum number of computations on data already stored in shared memory, therefore minimizing the data transfer between global and shared memory.
- **Exploit the available optimized CUDA libraries.** There are different efficient CUDA libraries for FFT, image processing, or linear algebra, among others, that can improve the performance of the code.

## 4. ELM GPU Implementation

In this section we present the CUDA implementation of the ELM algorithm used for the classification of hyperspectral datasets. We use the MAGMA library [15] to achieve optimal GPU linear algebra operations. This library has proven to be more efficient than others like CUBLAS [16] [15].

### 4.1 ELM-based Classification

In this section we briefly describe the GPU implementation of the V-ELM algorithm introduced in Sect. 2 for K independent ELMs. The algorithm consists of three main phases: preprocessing, training and test. The pseudocode in Fig. 2 shows the algorithm that has been implemented, including host and device codes. The kernels executed in GPU are placed between <> symbols. The pseudocode also includes the GM and SM acronyms to indicate kernels executed only in global memory and kernels that only use shared memory, respectively.

First, all the data are scaled in the range [0:1] (line 1 in the pseudocode). Given that ELM is a supervised learning algorithm and ground truths of the datasets are available, the pixels (pixel vectors) of each dataset are randomly distributed between two non overlapping sets: training and test. These two sets are stored in the matrices $\mathbf{X}_{\text{train}}$ and $\mathbf{X}_{\text{test}}$, respectively, where each row represents a sample and each column a spectral band (lines 3 and 4 in the pseudocode). Data matrices are converted to column major format in order to be used by the MAGMA library (line 5 in the pseudocode). The ground truth labels are also split into two target matrices, $\mathbf{Y}_{\text{train}}$, which is used during the learning phase, and $\mathbf{Y}_{\text{test}}$, which will be used to check the accuracy results. Finally, the training and test target matrices are processed so that each row represents a sample and each column a class, where a value of 1 indicates membership to a class and a value of -1 is assigned otherwise (line 6 in the pseudocode). The preprocessing phase is computed in CPU and the results are stored in the global memory of the GPU. All the remaining steps will be computed in GPU.

The training phase starts by generating random weights and biases (line 7 in the pseudocode). The weights matrix has a number of rows equal to the number of neurons in the hidden layer. The number of columns is the same as the number of neurons in the input layer (equal to the spectral band number). This matrix takes values in the range [-1:1]. The biases vector takes values in the range [0:1] and its size is equal to the number of neurons in the hidden layer. These two matrices are then stored in the global memory of the GPU. Then, we have to calculate the hidden layer output matrix $\mathbf{H}$. To do this, we first multiply the transpose of the weights matrix by the training matrix, then we add the biases vector to the result and, finally, we apply an activation function to each element of the matrix (lines 8, 9 and 10 in the pseudocode corresponding to (2)). In our case, a sigmoid function ($f(x) = 1 / (1 + e^{-x})$) is applied through a CUDA kernel with each thread operating over a single element of the matrix. The final training step consists in calculating the output weights multiplying the transpose of the pseudoinverse of matrix $\mathbf{H}$ by the training targets matrix (line 12 in the pseudocode corresponding to (6)). In the next section we will detail how to calculate the pseudoinverse of a matrix.

The test phase (lines 13 to 17 in the pseudocode) starts by multiplying the transpose of the previous generated weights matrix by the data matrix $\mathbf{X}_{\text{test}}$. Then, the biases vector is added and the activation function is applied just like in the training phase to obtain the test hidden layer matrix $\mathbf{H}$. These operations are analogous to those of lines 8 to 10 in the training phase. Then, the output matrix $\mathbf{Y}$ is calculated multiplying $\mathbf{H}^T$ by the output weights matrix $\beta$ obtained in the training phase. Finally, the estimated output label $\hat{\mathbf{T}}_i$ is calculated as the argument that maximizes $\mathbf{Y}_{i,c}$ for each sample,

$$\hat{T}_i = \arg\max_{c=1,\ldots,C} Y_{i,c}$$ (7)
4.2 Moore-Penrose Inverse of a Matrix

In this section we explain how to efficiently calculate the pseudoinverse of a matrix with CUDA using the MAGMA library. It is the most computationally costly operation in the training phase and the one that involves more steps (line 11 in the pseudocode). We implement it as described in [10].

We first check the dimensions of the input H matrix in order to know if the number of rows is lower than the number of columns. If this is the case, we use MAGMA to compute a matrix A like the multiplication of the original matrix by its transpose, otherwise we compute A as the multiplication of the transpose matrix by the original matrix. This operation ensures that A is a symmetric positive definite matrix.

The next step consists in calculating the Cholesky factorization of A with the MAGMA `dpotrf` function and then applying a kernel to nullify the upper triangle of the factorized matrix obtaining the L matrix. Unlike the other steps, this last kernel is launched in global memory.

Afterwards, an M matrix is calculated multiplying L\textsuperscript{T} by L and then we compute the inverse of this matrix with the MAGMA `dgetri` and `dgetrf` functions.

Finally, once all of these matrices have been calculated, we obtain the inverse of the original H matrix with a set of consecutive multiplications computed using the MAGMA `dgemm` function. If the dimension check of the H matrix at start resulted in that the row number is lower than the column number, we compute \( H^\dagger = H^\text{T} \times L \times M \times M \times L^\text{T} \), otherwise we compute \( H^\dagger = L \times M \times M \times L^\text{T} \times H^\text{T} \).

4.3 Voting-based ELM

The voting algorithm used in this work, as it was explained in Sect. 2.2, comprises a set of independent ELMs whose outputs are stored in a matrix. After all the computations we will obtain an \( N \times C \) matrix (being \( N \) the number of pixels and \( C \) the number of classes in the dataset) containing the vote of each ELM for every pixel of the image. This phase of the algorithm is not needed if a single ELM is launched.

The majority vote phase is shown in Fig. 3. For each sample, a vector \( S_i \) is used to store the vote of the \( K \) ELMs and then, the final label (\( \text{mvv} \text{T}_i \)) is calculated as the most repeated output value produced by the different ELMs for the sample. A CUDA kernel is launched and computed in global memory where each thread computes the majority vote for one pixel of the image.

5. Results

We have evaluated the proposed algorithm on a PC with a quad-core Intel Core i5-3470 at 3.20 GHz and 8 GB of RAM. The code has been compiled using the gcc 4.6.3 version with OpenMP 3.0 support under Linux using 4 threads. Regarding the GPU implementation, CUDA codes run on an NVIDIA GeForce GTX Titan with 14 SMXs and 192 CUDA cores each. The CUDA code has been compiled using nvcc with version 5.5 of the toolkit under Linux.

The accuracy results are expressed in terms of overall accuracy (OA) average accuracy (AA) and kappa coefficient [17]. The performance results are expressed in terms of

<table>
<thead>
<tr>
<th>Require: hyperspectral dataset ( X ), label set ( T ), ( K ): number of ELMs, ( L ): number of neurons in the hidden layer, ( C ): number of classes, ( N ): number of samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Scale hyperspectral dataset in [0:1]</td>
</tr>
<tr>
<td>2: for each ELM ( k ) in V-ELM (( k = 1, \ldots, K )) do</td>
</tr>
<tr>
<td>3: Randomly choose the training points</td>
</tr>
<tr>
<td>4: Take the remaining points for test</td>
</tr>
<tr>
<td>5: Store data in column major order matrices ( X_{\text{train}} ) and ( X_{\text{test}} )</td>
</tr>
<tr>
<td>6: Process target matrices ( T_{\text{train}} ) and ( T_{\text{test}} )</td>
</tr>
<tr>
<td>7: (&lt;\text{Generate random weights (a\textsubscript{i}) and biases (b\textsubscript{i}), i=1,...,L}&gt;)</td>
</tr>
<tr>
<td>8: (&lt;\text{Transpose the weight matrix and multiply by } X_{\text{train}} &gt;)</td>
</tr>
<tr>
<td>9: (&lt;\text{Add the biases}&gt;)</td>
</tr>
<tr>
<td>10: (&lt;\text{Apply activation (sigmoid) function to obtain } H &gt;)</td>
</tr>
<tr>
<td>11: (&lt;\text{Calculate } H^\dagger \text{ as he Moore-Penrose pseudoinverse of } H &gt;)</td>
</tr>
<tr>
<td>12: (&lt;\text{Calculate output as } \beta = H^\dagger \times T_{\text{train}} &gt;)</td>
</tr>
<tr>
<td>13: (&lt;\text{Transpose weight matrix and multiply by } X_{\text{test}} &gt;)</td>
</tr>
<tr>
<td>14: (&lt;\text{Add the biases}&gt;)</td>
</tr>
<tr>
<td>15: (&lt;\text{Apply activation (sigmoid) function to obtain } H &gt;)</td>
</tr>
<tr>
<td>16: (&lt;\text{Calculate output as } Y = (H)^T \times \beta &gt;)</td>
</tr>
<tr>
<td>17: (&lt;\text{Calculate the estimated output label}&gt;)</td>
</tr>
<tr>
<td>18: end for</td>
</tr>
<tr>
<td>19: (&lt;\text{Calculate the output as the majority vote of the } K \text{ estimated labels for each sample}&gt;)</td>
</tr>
</tbody>
</table>

Fig. 2 - Pseudocode for the V-ELM algorithm in GPU.

| 1: for each sample \( i \) (\( i = 1, \ldots, N \)) do |
| 2: for each ELM \( k \) (\( k = 1, \ldots, K \)) do |
| 3: \( S_i(T_i^k) = S_i(T_i^k) + 1 \) |
| 4: end for |
| 5: \( \text{mvv}T_i = \arg \max S_i \) |
| 6: end for |

Fig. 3 - Voting phase of the V-ELM algorithm.
running times and speedups compared to an OpenMP CPU optimized version of the ELM parallelized using 4 threads and whose algebra operations are accelerated with the LAPACK library [18] together with GoToBLAS2 [19]. For comparison purposes, speedups of the ELM implementations are also calculated comparing to an optimized SVM implementation with the parameter values and number of training samples taken from [20]. The running times include the training and test phases, completely executed in GPU, therefore they do not include CPU-GPU data transfers.

The test were run on two hyperspectral airborne datasets [21]: A 103-band ROSIS image of the University of Pavia (Pavia Univ.) with a spatial dimension of 610 x 340 pixels and a 220-band AVIRIS image of 145 x 145 pixels taken over Northwest Indiana (Indian Pines).

We compare three different configurations using ELM:

1) A single ELM trained with 200 samples for each class (ELM).
2) A V-ELM comprising 8 ELMs trained with a total of 200 samples for each class equally spread (with bootstrap) among the ELMs. This way the number of training points used by the 8 ELMs is the same as those used by the single ELM of the previous configuration (V-ELM-1).
3) A V-ELM comprising 8 ELMs trained with 200 samples for each class for each one of the ELMs, so that each ELM is the same as in the first configuration (V-ELM-2).

The number of training samples employed are 200 per class, or half the number of samples in the class if there are not enough samples. These samples are randomly chosen and all the remaining samples are used for test. The number of hidden layer neurons employed are 500 for Pavia Univ. and 950 for Indian Pines in all the cases [2]. These configurations were chosen in order to achieve the best accuracy [3].

Table 1 shows accuracy results for the Pavia Univ. and Indian Pines images in terms of OA, AA, and kappa.

The first thing to highlight in the results is that both the ELM and V-ELM-2 configurations obtain acceptable accuracy results, being best result slightly better than the SVM for both images. Regarding the V-ELM-1 configuration, as was expected, it offers in all the cases a lower accuracy than a single ELM. This is due to the very low number of samples to train each class in each ELM, so an overfitting is produced resulting in poor generalization capabilities. This is supported by the fact that every ELM in this configuration obtains 100% accuracy in the training phase but much lower accuracy in the later test phase.

For the Pavia Univ. image, the V-ELM-2 configuration clearly improves the ELM configuration while for the Indian Pines image both configurations obtain similar results, being the ELM configuration only slightly better. Figures 4 and 5 help in understanding the results. They represent the ground truth and false color classification maps for both datasets. The class specific accuracies for the SVM and the best ELM classification method applied to each image are shown in Tables 2 and 3. It can be observed that ELM clearly improves SVM in certain classes. In the case of Pavia Univ., the best improvements are achieved for the biggest class called meadows with an accuracy of 92.78% obtained by ELM while SVM obtains 70.79%. For the case of the Indian Pines dataset, the best ELM improvements are achieved for the classes corn-mintill (76.72% obtained by ELM and 69.64% by SVM) and bldg-grass-tree-drives (89.00% obtained by ELM and 61.52% by SVM). For these classes, more homogeneous areas are observed in the ELM classification maps of the Figures 4 and 5.

Table 4 shows performance results in terms of running times and speedups calculated over the OpenMP multicore implementations for the Pavia Univ. and Indian Pines datasets. It can be observed that the CUDA GPU implementation is faster than the OpenMP CPU one for all the configurations in both datasets (up to 10.1 × for SVM and 8.6 × for V-ELM-1 for the Pavia Univ. image). It also can be checked that the single ELM configuration is the fastest one for both datasets. Regarding the comparison to the SVM implementation, Table 5 shows that, for both images, the single ELM configuration is faster than SVM, 8.6 times faster for the Pavia Univ. image in CPU and 6.5 times for the same image in GPU.

Running times also indicate that V-ELM-2 is better than V-ELM-1 because V-ELM-1 only saves time against V-ELM-2 in the training phase. This represents a small part of the total
Table 1

Classification accuracy as percentages for the Pavia Univ. and Indian Pines images. The ELMs contained 500 and 950 nodes in the hidden layer, respectively.

<table>
<thead>
<tr>
<th></th>
<th>Pavia Univ.</th>
<th>Indian Pines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OA</td>
<td>AA</td>
</tr>
<tr>
<td>SVM</td>
<td>81.01</td>
<td>88.25</td>
</tr>
<tr>
<td>ELM</td>
<td>86.68</td>
<td>89.46</td>
</tr>
<tr>
<td>V-ELM-1</td>
<td>79.20</td>
<td>77.29</td>
</tr>
<tr>
<td>V-ELM-2</td>
<td>90.31</td>
<td>91.78</td>
</tr>
</tbody>
</table>

Fig. 4

From left to right, Pavia Univ. ground truth, SVM classification map, and best ELM classification map in terms of accuracy (V-ELM-2).

Fig. 5

From left to right, Indian Pines ground truth, SVM classification map, and best ELM classification map in terms of accuracy (ELM).

Table 4

Performance results for the Pavia Univ. and Indian Pines images.

<table>
<thead>
<tr>
<th></th>
<th>Pavia Univ.</th>
<th>Indian Pines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SVM</td>
<td>ELM</td>
</tr>
<tr>
<td>OpenMP CPU</td>
<td>19.9844s</td>
<td>2.3304s</td>
</tr>
<tr>
<td>CUDA GPU</td>
<td>1.9802s</td>
<td>0.3063s</td>
</tr>
<tr>
<td>speedup CPU-GPU</td>
<td>10.1×</td>
<td>7.6×</td>
</tr>
<tr>
<td>Indian Pines</td>
<td>SVM</td>
<td>ELM</td>
</tr>
<tr>
<td>OpenMP CPU</td>
<td>2.6980s</td>
<td>1.1653s</td>
</tr>
<tr>
<td>CUDA GPU</td>
<td>0.4548s</td>
<td>0.3096s</td>
</tr>
<tr>
<td>speedup CPU-GPU</td>
<td>5.9×</td>
<td>3.8×</td>
</tr>
</tbody>
</table>
time, resulting in a final time slightly higher for V-ELM-2 while its accuracy is better as we explained before.

The V-ELM-2 configuration provides more stable results than a single ELM in exchange for a higher execution time. The V-ELM-2 configuration is more interesting when the dataset size is large because if the dataset is too small (as in the case of Indian Pines) there are not enough samples to take advantage of the voting. Besides, in big datasets, the ELM algorithm has larger speedups against SVM allowing the voting configurations to be executed in almost the same time as a single SVM, as shown in Table 5.

Summarizing, on the one hand, the raw ELM algorithm described in this paper is significantly faster than SVM and, on the other hand, the V-ELM-2 algorithm always approaches or improves the raw ELM accuracy.

6. Conclusions

In this paper we have presented an ELM-based GPU implementation to efficiently classify hyperspectral datasets exploiting efficiently the hundreds of threads available, using shared memory to make an effective use of the memory hierarchy, and exploiting a linear algebra library. Different ensemble configurations were also considered to achieve better classification accuracies.

Results have shown that commodity GPUs like the GTX Titan used in this work are good candidates to reduce computation times in order to achieve real-time hyperspectral processing. For the raw ELM and the Pavia Univ. and Indian Pines datasets, speedups of 7.6× and 3.8× are achieved, respectively, compared to the ELM CPU classification. Results also show that the hyperspectral dataset classification using ELM is faster than the SVM one, up to 8.6× faster in CPU and 6.5× in GPU.

Acknowledgements

This work was supported in part by the Ministry of Science and Innovation, Government of Spain, cofounded by the FEDER funds of European Union, under contract TIN 2010-17541, and by Xunta de Galicia, Program for Consolidation of Competitive Research Groups ref. 2010/28. Javier acknowledges financial support from the Xunta de Galicia, under a predoctoral grant.

References


