CiTiUS
Centro Singular de Investigación en Tecnoloxías da Información

# Clasificación Supervisada de Imágenes de Sensado Remoto en GPU


Javier López-Fandiño
Centro Singular de Investigación en Tecnoloxías da Información
javier.lopez.fandino@usc.es
http://citius.usc.es

Dora B. Heras
Centro Singular de Investigación en Tecnoloxías da Información
dora.blanco@usc.es
http://citius.usc.es

Francisco Argüello
Centro Singular de Investigación en Tecnoloxías da Información
francisco.arguello@usc.es
http://citius.usc.es



Congreso JPXXV2014 17/09/2014

Gracias a SARTECO por su permiso para publicar este documento


CAMPUS VIDA
USC UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

**Resumen**

Los algoritmos de procesado de imágenes hiperespectrales son muy costosos computacionalmente, lo que los convierte en buenos candidatos para el procesamiento paralelo y, en particular, el procesamiento en GPU. Extreme Learning Machine (ELM) es un algoritmo de clasificación propuesto recientemente muy adecuado para su implementación en plataformas GPU. En este artículo proponemos una implementación eficiente en GPU de una estrategia de clasificación para imágenes hiperespectrales basada en ELM. ELM puede expresarse en términos de operaciones matriciales que pueden aprovechar al máximo la arquitectura de una GPU. En lo referente a la precisión de clasificación, el algoritmo propuesto alcanza resultados competitivos comparado con una estrategia tradicional basada en SVM con tiempos de ejecución significativamente inferiores. Además, se considera el uso de un mecanismo de voto para mejorar los resultados de precisión.

**Palabras clave:** Imágenes hiperespectrales, redes neuronales, ELM, SVM, GPU, CUDA, MAGMA.

# 1 Introducción

Actualmente, los datasets hiperespectrales están disponibles en variedad de contextos gracias a los recientes avances en la tecnología de sensores. Estas imágenes proveen información de cientos de bandas espectrales a diferentes longitudes de onda para cada píxel, permitiendo discriminar entre diferentes materiales y objetos. Mediante el procesado de datasets hiperespectrales pueden abordarse distintos problemas: clasificación, segmentación, detección de objetivos, entre otros.

Métodos tradicionales como las redes neuronales RBF (funciones de base radial) o KNN (K vecinos más próximos) han sido usados para la clasificación de imágenes hiperespectrales. Sin embargo, otros algoritmos como SVM [1] son más adecuados para procesar toda la información almacenada en los canales espectrales de este tipo de imágenes.

A pesar de esto, para alcanzar procesado en tiempo real, son necesarios métodos más rápidos que SVM, incluso si se utilizan sistemas de computación de altas prestaciones. Las *Extreme Learning Machine* (ELM) son un método novedoso que mejora los tiempos de ejecución de SVM y que ha sido utilizado previamente en el sensado remoto [2, 3]. Además, ELM es un algoritmo adecuado para su implementación en GPU porque las operaciones requeridas son en su mayoría de tipo matricial, que pueden ser computadas en bloques sin dependencias de datos entre ellos.

Distintas técnicas basadas en ELM han sido propuestas [4]: *online sequential* ELM, que es adecuado cuando los datos se reciben en fragmentos, *incremental* ELM, en el cual los nodos son añadidos de uno en uno, o *pruned* ELM, que empieza con una red grande de la que posteriormente se eliminan los nodos ocultos que tienen poca relevancia para las etiquetas de clase. Una conocida técnica para mejorar fácilmente los resultados de un método de clasificación consiste en el uso de conjuntos, es decir, combinar los resultados obtenidos mediante diferentes técnicas de clasificación o mediante la misma técnica entrenada con diferentes datasets [5]. Un mecanismo simple para combinar los resultados de los conjuntos es el voto de la mayoría, es decir, asignar al valor final de una muestra el valor de etiqueta más repetido en el agrupamiento. *Ensemble Based Extreme Learning Machine* (EN-ELM)

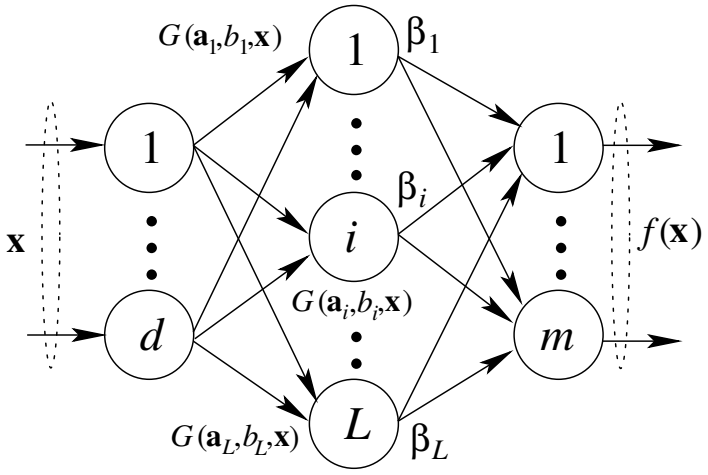


Figure 1: SLFN tal como es usada por ELM.

[6], y *Voting-based Extreme Learning Machine* (V-ELM) [7] son ejemplos de esta técnica aplicada a ELM.

En este artículo presentamos una implementación GPU CUDA (*Compute Unified Device Architecture*) eficiente de un algoritmo basado en ELM para clasificar datasets hiperespectrales en tiempo real, así como diferentes estrategias para combinar estos clasificadores mediante un mecanismo de voto de la mayoría, también implementado en en GPU, para mejorar la precisión de los resultados. Las claves para la implementación son la explotación de los miles de *threads* disponibles en la arquitectura GPU y el uso adecuado de la jerarquía de memoria. El algoritmo GPU está formulado en términos de operaciones matriciales que se ejecutan en bloques de forma eficiente mediante una librería de álgebra lineal optimizada.

Las restantes secciones de este artículo se organizan de la siguiente manera: La sección 2.1 explica el mecanismo ELM para clasificar imágenes hiperespectrales, mientras que el mecanismo de voto de la mayoría para combinar clasificadores se presenta en la sección 2.2. Introducimos algunos fundamentos sobre GPU y CUDA en la sección 3. Las implementaciones en GPU de los algoritmos son descritas en la sección 4. La sección 5 está dedicada a la discusión de los resultados experimentales. Por último, la sección 6 resume las conclusiones de este trabajo.

# 2 Clasificación de Imágenes Hiperespectrales Mediante Técnicas basadas en ELM

En esta sección, presentamos los algoritmos ELM y V-ELM, cuya implementación en GPU será estudiada en la sección 4.

## 2.1 Clasificación basada en ELM

El algoritmo ELM píxel a píxel ha sido propuesto como un algoritmo de aprendizaje eficiente para las redes neuronales de una única capa oculta (SLFNs) [8].

La función de salida de una SLFN con $L$ nodos ocultos y $m$ nodos de salida y siendo $\boldsymbol{x}$ el vector de entrada (ver Fig. 1) puede escribirse como

$$f_L(\boldsymbol{x}) = \sum_{i=1}^{L} \boldsymbol{\beta}_i G(\boldsymbol{a}_i, b_i, \boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{R}^d,\ \boldsymbol{\beta}_i \in \mathbb{R}^m, \tag{1}$$

donde $G(\boldsymbol{a}_i, b_i, \boldsymbol{x})$ denota la función de salida del $i$-ésimo nodo oculto, siendo $\boldsymbol{a}_i$, $b_i$ los parámetros del nodo oculto y $\boldsymbol{\beta}_i$ el vector de pesos que conecta el $i$-ésimo nodo oculto a los nodos de salida. Para el caso de nodos aditivos con función de activación $g$, puede expresarse como

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

Una SLFN con $L$ nodos ocultos puede aproximar $N$ distintas muestras arbitrarias $(\boldsymbol{x}_i, \boldsymbol{t}_i) \in \mathbb{R}^d \times \mathbb{R}^m$, siendo $\boldsymbol{t}_i$ la etiqueta de clase esperada, si el siguiente sistema de ecuaciones puede ser resuelto:

$$\mathbf{H}\boldsymbol{\beta} = \mathbf{T} \tag{3}$$

donde

$$\mathbf{H} = \begin{bmatrix} \boldsymbol{h}(\boldsymbol{x}_1) \\ \vdots \\ \boldsymbol{h}(\boldsymbol{x}_N) \end{bmatrix} = \begin{bmatrix} G(\boldsymbol{a}_1, b_1, \boldsymbol{x}_1) & \dots & G(\boldsymbol{a}_L, b_L, \boldsymbol{x}_1) \\ \vdots & \dots & \vdots \\ G(\boldsymbol{a}_1, b_1, \boldsymbol{x}_N) & \dots & G(\boldsymbol{a}_L, b_L, \boldsymbol{x}_N) \end{bmatrix}_{N \times L}, \tag{4}$$

$$\boldsymbol{\beta} = \begin{bmatrix} \boldsymbol{\beta}_1^T \\ \vdots \\ \boldsymbol{\beta}_L^T \end{bmatrix}_{L \times m}, \quad \text{y} \quad \mathbf{T} = \begin{bmatrix} \boldsymbol{t}_1{}^T \\ \vdots \\ \boldsymbol{t}_N{}^T \end{bmatrix}_{N \times m}. \tag{5}$$

Se denomina $\mathbf{H}$ a la matriz de salida de la capa oculta de la red neuronal. Huang et al. [9, 4] han probado que una SLFN con nodos aditivos o RBF aleatoriamente generados en la capa oculta puede aproximar universalmente cualquier función objetivo continua sobre cualquier subconjunto compacto $\chi \subset \mathbb{R}^d$. Para el caso de nodos aditivos, la función de activación $g$ puede ser cualquier función infinitamente diferenciable, incluyendo funciones sigmoidales, y también funciones de base radial, sinusoidales, cosenoidales y exponenciales, entre otras.

Una vez generados aleatoriamente, los parámetros de los nodos ocultos $(\boldsymbol{a}_i, b_i)$ permanecen fijados. Entrenar una SLFN equivale a encontrar la solución de mínimos cuadrados $\hat{\boldsymbol{\beta}}$ del sistema linear $\mathbf{H}\boldsymbol{\beta} = \mathbf{T}$, es decir:

$$\hat{\boldsymbol{\beta}} = \mathbf{H}^{\dagger}\mathbf{T}, \tag{6}$$

donde $\mathbf{H}^{\dagger}$ es la *inversa generalizada Moore-Penrose* de la matriz $\mathbf{H}$ [10].

Por lo tanto, ELM puede resumirse de la siguiente manera [9, 4]:

**Algoritmo ELM**: Dado un conjunto de entrenamiento $\{(\boldsymbol{x}_i, \boldsymbol{t}_i) | \boldsymbol{x}_i \in \mathbb{R}^d, \boldsymbol{t}_i \in \mathbb{R}^m, i = 1, \dots, N\}$, función de salida de los nodos ocultos $G(\boldsymbol{a}_i, b_i, \mathbf{x})$,y número de nodos ocultos $L$,

1. Generar aleatoriamente los parámetros de los nodos ocultos $(\boldsymbol{a}_i, b_i),\ i = 1, \ldots, L$ donde $\boldsymbol{a}_i$ y $b_i$ son los valores de los pesos y desviaciones de entrada.
2. Calcular la matriz de salida de la capa oculta $\mathbf{H}$.
3. Calcular el vector de pesos de salida, $\boldsymbol{\beta} = \mathbf{H}^{\dagger}\mathbf{T}$.

Como se indica en [11], ELM requiere menos intervención humana que SVM porque se requiere optimizar un único parámetro, el número de neuronas en la capa oculta, ya que los demás parámetros se inicializan con valores aleatorios. ELM también alcanza precisiones similares o mejores que SVM tanto para clasificaciones binarias como multiclase. Además, ELM es más escalable y se ejecuta en mucho menos tiempo que SVM.

Para nuestro caso de imágenes hiperespectrales, cada muestra de entrenamiento representa un píxel seleccionado aleatoriamente en la imagen y cada componente de la muestra una banda espectral de dicho píxel. La salida que se obtiene tras la fase de clasificación es la predicción de clase para cada píxel de la imagen.

## 2.2 Voting-based ELM

En este trabajo seguimos una aproximación basada en conjuntos [7, 12], es decir, computar un número de ELMs independientes con el mismo número de nodos ocultos y la misma función de activación en cada nodo oculto y combinar los resultados obtenidos por los distintos clasificadores. Los distintos ELMs son entrenados con el mismo dataset y los parámetros de aprendizaje de cada ELM son generados de forma aleatoria e independiente.

El voto de la mayoría es el mecanismo de implementación más simple entre los métodos de combinación porque no asume conocimiento a priori del comportamiento de los clasificadores individuales y no requiere entrenamiento [13]. Utilizamos un mecanismo de voto de la mayoría democrático donde el voto de cada clasificador tiene el mismo peso que los demás y la decisión final para cada muestra es el voto más repetido para dicha muestra.

# 3 Modelo de programación CUDA GPU

En la actualidad, las GPU proporcionan capacidades de procesamiento masivamente paralelo basado en su arquitectura de paralelismo de datos. CUDA es una plataforma que combina hardware y software que permite a las GPUs de NVIDIA ejecutar programas invocando funciones paralelas llamadas kernels que se ejecutan en multitud de *threads* en paralelo [14]. Estos *threads* están organizados en bloques de forma que cada *thread* ejecuta una instancia del kernel siguiendo un modelo de programación SIMD (Single Instruction Multiple Data). Los bloques se agrupan en un grid que se mapea a una jerarquía de núcleos CUDA en la GPU.

Un multiprocesador de streaming (llamado SM) contiene un elevado número de núcleos CUDA y ejecuta uno o más bloques de *threads*. Los *threads* se ejecutan en grupos de 32 llamados warps. Si todos los *threads* de un warp ejecutan el mismo código y acceden a posiciones de memoria con direcciones próximas el rendimiento mejora considerablemente.

Los *threads* pueden acceder a datos de múltiples espacios de memoria. En primer lugar, cada *thread* dispone de una memoria privada local y de registros. Cada bloque de *threads* tiene una memoria compartida visible exclusivamente para los *threads* pertenecientes a ese

bloque y cuyo tiempo de vida coincide con el tiempo de vida del bloque. Finalmente, todos los *threads* acceden a un mismo espacio de memoria global (DRAM) que es persistente entre llamadas a kernel por la misma aplicación. Cuanto más bajo sea el nivel de memoria empleado, más rápidos serán los accesos para lectura/escritura de datos. El tiempo de vida de la memoria compartida dificulta compartir datos entre bloques de *threads* porque implica el uso de memoria global, cuyo acceso es más lento que el acceso a memoria compartida.

En la arquitectura de GPU Kepler, se incluye una jerarquía de caché de dos niveles. Cada SM dispone de 64 KB de memoria que puede configurarse a partes iguales como memoria compartida y caché L1, 48 KB de memoria compartida y 16 KB de caché L1 o viceversa. También se dispone de una caché L2 unificada de 1536 KB compartida entre todos los SM. En la más reciente arquitectura Maxwell [14] se dispone de 64 KB dedicados para la memoria compartida ya que la caché L1 se ubica en la misma unidad que la memoria de texturas.

Para optimizar el rendimiento computacional, se han aplicado distintas estrategias de optimización:

- **Maximizar la ejecución en paralelo.** Es importante organizar el algoritmo en bloques computacionales que puedan ejecutarse de forma independiente minimizando las comunicaciones entre ellos.
- **Mejorar la eficiencia en el uso de la jerarquía de memoria.** Tratando de maximizar tanto la localidad espacial como temporal en los accesos a datos, reduciendo el movimiento de datos entre diferentes niveles de memoria. Es esencial realizar el mayor número de computaciones en datos ya almacenados en memoria compartida, minimizando las transferencias entre memoria global y compartida.
- **Explotar las librerías CUDA optimizadas disponibles.** Existen diferentes librerías CUDA para FFT, procesado de imagen, o álgebra lineal, entre otras, que pueden ser usadas.

# 4 Implementación de ELM en GPU

En esta sección presentamos la implementación CUDA del algoritmo ELM utilizado para la clasificación de datasets hiperespectrales. Usaremos la librería MAGMA para optimizar la ejecución de operaciones de álgebra lineal en GPU. Esta librería a probado ser más eficiente que otras como CUBLAS [15] [16].

## 4.1 Clasificación basada en ELM

En esta sección describimos brevemente la implementación GPU del algoritmo V-ELM itroducido en la sección 2 para $K$ ELMs independientes. El algoritmo tiene tres fases principales: preprocesado, entrenamiento y test. El pseudocódigo en la Fig. 2 muestra el algoritmo que ha sido implementado, incluyendo los códigos del anfitrión y el dispositivo. Los kernels ejecutados en GPU se indican entre símbolos $<>$. El pseudocódigo también incluye los acrónimos GM y SM para indicar kernels ejecutados en memoria global y kernels que utilizan memoria compartida, respectivamente.

**Require:** hyperspectral dataset $\mathbf{X}$, label set $\mathbf{T}$, $K$: number of ELMs, $L$: number of neurons in the hidden layer, $C$: number of classes, $N$: number of samples

1: Scale hyperspectral dataset in [0:1] ▷ Preprocessing phase
2: **for each** ELM $k$ in V-ELM ($k = 1, \ldots, K$) **do**
3: Randomly choose the training points
4: Take the remaining points for test
5: Store data in column major order matrices $\mathbf{X}_{train}$ and $\mathbf{X}_{test}$
6: Process target matrices $\mathbf{T}_{train}$ and $\mathbf{T}_{test}$
▷ Training phase
7: <Generate random weights ($\mathbf{a}_i$) and biases ($b_i$), $i$=1,...,$L$> ▷ GM
8: <Transpose the weight matrix and multiply by $\mathbf{X}_{train}$ > ▷ SM
9: <Add the biases> ▷ SM
10: <Apply activation (sigmoid) function to obtain $\mathbf{H}$ > ▷ GM
11: <Calculate $\mathbf{H}^{\dagger}$ as he Moore-Penrose pseudoinverse of $\mathbf{H}$ > ▷ GM-SM
12: <Calculate output as $\boldsymbol{\beta} = \mathbf{H}^{\dagger} \times \mathbf{T}_{train}$ > ▷ SM
▷ Test phase
13: <Transpose weight matrix and multiply by $\mathbf{X}_{test}$ > ▷ SM
14: <Add the biases> ▷ SM
15: <Apply activation (sigmoid) function to obtain $\mathbf{H}$ > ▷ GM
16: <Calculate output as $\mathbf{Y} = (\mathbf{H})^T \times \boldsymbol{\beta}$ > ▷ SM
17: <Calculate the estimated output label> ▷ GM
18: **end for**
▷ Majority vote phase
19: <Calculate the output as the majority vote of the $K$ estimated labels for each sample> ▷ GM
▷ GM: Global Memory, SM: Shared Memory

Figure 2: Pseudocódigo para el algoritmo V-ELM.

En primer lugar, todos los datos son escalados en el rango [0:1] (línea 1 en el pseudocódigo). Dado que ELM es un algoritmo de aprendizaje supervisado y que se dispone de clasificaciones de referencia de los datasets, los vectores de píxeles de cada dataset son distribuidos aleatoriamente en dos conjuntos mutuamente exclusivos: entrenamiento y test. Estos dos conjuntos son almacenados en las matrices $\mathbf{X}_{train}$ y $\mathbf{X}_{test}$, respectivamente, donde cada fila representa un píxel de muestra y cada columna una banda espectral (líneas 3 y 4 en el pseudocódigo). Las matrices de datos se pasan a almacenar por columnas para poder ser usadas en la librería MAGMA (línea 5 en el pseudocódigo). Las etiquetas de la clasificación de referencia también se dividen en dos matrices de objetivos: $\mathbf{T}_{train}$, utilizada durante la fase de aprendizaje y $\mathbf{T}_{test}$ que se utilizará para comprobar la precisión de los resultados. Finalmente, las matrices de objetivos de entrenamiento y test se procesan de forma que cada fila represente una muestra y cada columna una clase, donde un valor de 1 indica pertenencia a una clase y un valor de -1 es asignado en otro caso (línea 6 en el pseudocódigo). La fase de preprocesado se computa en CPU y los resultados se almacenan en la memoria global de la GPU. Todos los pasos restantes serán computados en GPU.

La fase de entrenamiento empieza generando aleatoriamente pesos y desviaciones (línea 7 en el pseudocódigo). La matriz de pesos debe tener valores en el rango [-1:1] y sus dimensiones se determinan por el número de neuronas en la capa oculta y el número de neuronas de entrada (igual al número de bandas espectrales). El vector de desviaciones tendrá valores en el rango [0:1] y tamaño igual al número de neuronas en la capa oculta. Estas dos matrices se almacenan posteriormente en la memoria global de la GPU. A continuación, se calcula la matriz de salida de la capa oculta $\mathbf{H}$. Para hacer esto, en primer lugar se multiplica la traspuesta de la matriz de pesos por la matriz de entrenamiento, después se añade el vector de desviaciones al resultado y finalmente se aplica la función de activación a cada elemento de la matriz (líneas 8, 9 y 10 en el pseudocódigo, correspondientes con (2)). En nuestro caso, se aplica una función sigmoide ($f(x) = 1/(1 + e^{-x})$) mediante un

kernel CUDA en el que cada *thread* opera sobre un elemento de la matriz. El último paso del entrenamiento consiste en calcular los pesos de salida multiplicando la traspuesta de la pseudoinversa de la matriz $\mathbf{H}$ por la matriz de objetivos de entrenamiento (línea 12 en el pseudocódigo, correspondiente a (6)). En la siguiente sección detallaremos como calcular la pseudoinversa de una matriz.

La fase de test (líneas 13 a 17 en el pseudocódigo) empieza multiplicando la traspuesta de la matriz de pesos generada previamente por la matriz de datos $\mathbf{X}_{test}$. A continuación, se suma el vector de desviaciones y se aplica la función de activación de la misma manera que en la fase de entrenamiento, obteniendo la matriz de la capa oculta de test $\mathbf{H}$. Estas operaciones son análogas a las realizadas en las líneas 8 a la 10 en la fase de entrenamiento. Después, la matriz de salida $\mathbf{Y}$ se calcula multiplicando $\mathbf{H}^T$ por la matriz de pesos de salida $\beta$ obtenida en la fase de entrenamiento. Finalmente, las etiquetas de salida estimadas $\mathbf{T}_i$ se calculan como el argumento que maximiza $\mathbf{Y}_i$ para cada muestra,

$$\hat{\mathbf{T}}_i = \underset{c=1,\ldots,C}{\arg\max}\, \mathbf{Y}_{i,c} \tag{7}$$

## 4.2 Inversa Moore-Penrose de una Matriz

En esta sección explicamos como calcular eficientemente la pseudoinversa de una matriz utilizando CUDA y la librería MAGMA puesto que es la operación más costosa computacionalmente de la fase de entrenamiento así como la que implica más pasos (línea 11 en el pseudocódigo). La implementación se realiza de la forma descrita en [10].

En primer lugar comprobamos las dimensiones de la matriz de entrada $\mathbf{H}$ para saber si el número de filas es menor que el número de columnas. Si ese es el caso, se utiliza MAGMA para computar la matriz $\mathbf{A}$ como la multiplicación de la matriz original por su transpuesta, en otro caso, se computa $\mathbf{A}$ como la multiplicación de la traspuesta de la matriz por la matriz original. Esta operación asegura que $\mathbf{A}$ es una matriz simétrica definida positiva.

El siguiente paso consiste en calcular la factorización Cholesky de $\mathbf{A}$ con la función *dpotrf* de MAGMA y a continuación aplicar un kernel que ponga a cero el triangulo superior de la matriz factorizada obteniendo así la matriz $\mathbf{L}$. A diferencia de los demás pasos, este último kernel es ejecutado en memoria global.

Después, la matriz $\mathbf{M}$ se calcula multiplicando $\mathbf{L}^T$ por $\mathbf{L}$ y a continuación se computa la inversa de esta matriz con las funciones *dgetrf* y *dgetri* de MAGMA.

Finalmente, la inversa de la matriz original $\mathbf{H}$ se obtiene mediante un conjunto de multiplicaciones consecutivas utilizando la función *dgemm* de MAGMA. Si la comprobación de dimensiones de la matriz $\mathbf{H}$ realizada al comienzo resultó en que el número de filas es menor que el número de columnas, $\mathbf{H}^\dagger$ se computa como,

$$\mathbf{H}^\dagger = \mathbf{H}^T \times \mathbf{L} \times \mathbf{M} \times \mathbf{M} \times \mathbf{L}^T,$$

en otro caso se computa

$$\mathbf{H}^\dagger = \mathbf{L} \times \mathbf{M} \times \mathbf{M} \times \mathbf{L}^T \times \mathbf{H}^T.$$

## 4.3 Voting-based ELM

El algoritmo de voto utilizado en este trabajo, como se explicó en la sección 2.2, comprende un conjunto de ELMs independientes cuyas salidas se almacenan en una matriz. Después

1: **for each** sample $i$ $(i = 1, \ldots, N)$ **do**
2: **for each** ELM $k$ $(k = 1, \ldots, K)$ **do**
3: $\mathbf{S}_i(\hat{\mathbf{T}}_i^k) = \mathbf{S}_i(\hat{\mathbf{T}}_i^k) + 1$
4: **end for**
5: $\mathbf{mvT}_i = \underset{c=1,\ldots,C}{\arg\max}\, \mathbf{S}_{i,c}$
6: **end for**

Figure 3: Fase de voto del algoritmo V-ELM.

de que todas las computaciones se realicen, obtenemos una matriz $\mathbb{R}^N \times \mathbb{R}^C$ (siendo *N* el número de píxeles y *C* el número de clases en el dataset) con el voto de cada ELM para cada uno de los píxeles de la imagen. Esta fase del algoritmo no es necesaria si se ejecuta un único ELM.

La fase de voto de la mayoría se muestra en la Fig. 3. Para cada muestra, se utiliza un vector $\mathbf{S}_i$ para almacenar el voto de los *K* ELMs y, a continuación, la etiqueta final ($\mathbf{mvT}_i$) se calcula como el valor de salida más repetido producido por los diferentes ELMs para esa muestra. Un kernel CUDA se lanza y computa en memoria global de forma que cada *thread* computa el voto de la mayoría para un píxel de la imagen.

# 5 Resultados

Hemos evaluado el algoritmo propuesto en un PC con un Intel Core i5-3470 quad-core con 4 *threads* a 3.20 Ghz y 8 GB de RAM. El código ha sido compilado usando la versión 4.6.3 de gcc con soporte para OpenMP 3.0 bajo Linux. En lo referente a la implementación GPU, los códigos CUDA se ejecutan en una NVIDIA GeForce GTX Titan con 14 SMXs y 192 núcleos CUDA cada uno. El código CUDA se ha compilado usando nvcc con la versión 5.5 de toolkit bajo Linux.

Los resultados de precisión están expresados en términos de precision global (OA), precisión media (AA) y coeficiente *kappa* [17]. Los resultados de rendimiento están expresados en términos de tiempo de ejecución y *speedups* comparados con una versión CPU optimizada de ELM paralelizada mediante OpenMP utilizando 4 *threads* y cuyas operaciones algebraicas están aceleradas con la librería LAPACK [18]. Para propósitos de comparación, los *speedups* de las implementaciones de ELM también están calculados en comparación con una versión optimizada de SVM con parámetros y número de muestras de entrenamiento tomados de [19].

Los test fueron ejecutados en dos datasets hiperespectrales de imagen aérea [20]: Una imagen ROSIS de 103 bandas de la Universidad de Pavia (Pavia Univ.) cuyas dimensiones espaciales son 610 x 340 píxeles y una imagen AVIRIS de 220 bandas de 145 x 145 píxeles tomada sobre el noroeste de Indiana (Indian Pines).

Hemos comparado tres configuraciones diferentes de ELM:

1. Un ELM individual entrenado con 200 muestras para cada clase (ELM).

2. Un V-ELM conteniendo 8 ELMs entrenados con un total de 200 muestras para cada clase repartidas de forma equitativa (con reposición) entre los ELMs. De esta forma

Table 1: Precisión de clasificación, en porcentaje, para las imágenes Pavia Univ. e Indian Pines. Los ELMs contienen 500 y 950 nodos en la capa oculta, respectivamente.

| | Pavia Univ. | | | Indian Pines | | |
|---|---|---|---|---|---|---|
| | OA | AA | *kappa* | OA | AA | *kappa* |
| SVM | 81.01 | 88.25 | 75.86 | 78.17 | 85.97 | 75.33 |
| ELM | 86.68 | 89.46 | 82.52 | 80.81 | 86.20 | 77.81 |
| V-ELM-1 | 79.20 | 77.29 | 72.62 | 63.14 | 75.66 | 58.67 |
| V-ELM-2 | 90.31 | 91.78 | 85.78 | 79.59 | 80.42 | 72.13 |

el número de puntos de entrenamiento utilizados por los 8 ELMs es el mismo que el usado en el ELM individual de la configuración previa (V-ELM-1).

3. Un V-ELM conteniendo 8 ELMs entrenados con 200 muestras de cada clase para cada uno de los ELMs, de forma que cada ELM tiene las mismas características que el de la primera configuración (V-ELM-2).

El número de muestras de entrenamiento empleado es de 200 por clase, o la mitad de las muestras disponibles para la clase si no hay suficientes. Estas muestras son seleccionadas aleatoriamente y todas las demás muestras son utilizadas para test. El número de neuronas en la capa oculta es 500 para Pavia Univ. y 950 para Indian Pines en todos los casos [2]. Estas configuraciones fueron elegidas para alcanzar la mejor precisión [3].

La tabla 1 muestra los resultados de precisión para las imágenes Pavia Univ. e Indian Pines en términos de OA, AA, y *kappa*. El primer aspecto a destacar en los resultados es que tanto la configuración ELM como la configuración V-ELM-2 obtienen resultados de precisión aceptables, siendo el mejor resultado ligeramente mejor que para SVM en ambas imágenes. En lo referente a la configuración V-ELM-1, como se esperaba, ofrece en todos los casos peores precisiones que un ELM individual. Esto se debe al hecho de que esta configuración deja muy pocos puntos para entrenar cada clase en cada ELM y se produce un sobreaprendizaje resultando en malas capacidades de generalización. Esto se comprueba por el hecho de que cada ELM en esta configuración obtiene un 100% de precisión en la fase de entrenamiento pero una precisión mucho peor en la posterior fase de test.

Para la imagen Pavia Univ., la configuración V-ELM-2 mejora con claridad la configuración ELM mientras que para la imagen Indian Pines ambas configuraciones obtienen resultados similares, siendo la configuración ELM ligeramente mejor. Las figuras 4 y 5 muestran los mapas de clasificación en falso color obtenidos para ambas imágenes.

La tabla 2 muestra los resultados de rendimiento en términos de tiempos de ejecución y speedups calculados sobre la implementación multicore OpenMP para las imágenes Pavia Univ. e Indian Pines. La tabla 3 muestra que, para ambas imágenes, el ELM individual es más rápido que SVM, alcanzando, en el mejor caso, un *speedup* de 8.6× para la imagen Pavia Univ. en CPU y 6.5× para la misma imagen en GPU.

Los tiempos de ejecución también indican que V-ELM-2 es más adecuado que V-ELM-1 ya que V-ELM-1 únicamente ahorra tiempo de ejecución respecto a V-ELM-2 en la fase de entrenamiento. Esto representa un pequeño porcentaje del tiempo total, resultando un tiempo final ligeramente mayor para V-ELM-2 mientras que la precisión es mejor, como explicamos anteriormente.

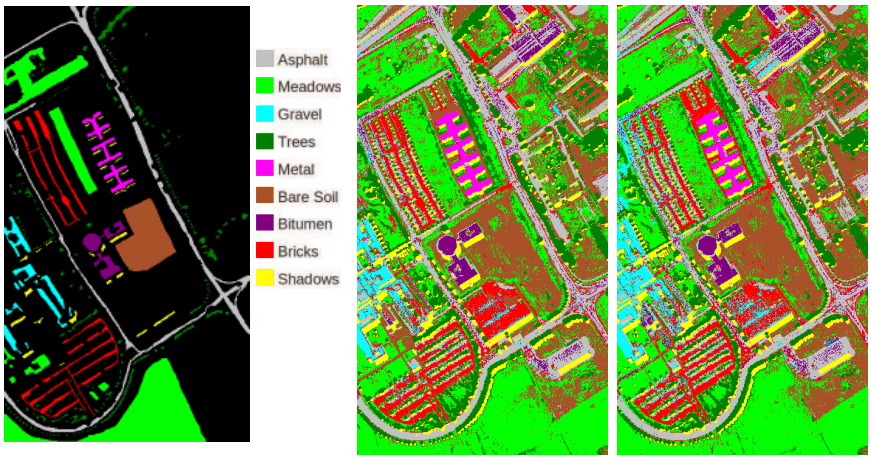


Figure 4: Mapa de referencia (izquierda), mapa de clasificación SVM (centro) y mapa de clasificación V-ELM-2 (derecha) para la imagen Pavia Univ.


Figure 5: Mapa de referencia (izquierda), mapa de clasificación SVM (centro) y mapa de clasificación V-ELM-2 (derecha) para la imagen Indian Pines.

Table 2: Resultados de rendimiento para las imágenes Pavia Univ. e Indian Pines.

| **Pavia Univ.** | SVM | ELM | V-ELM-1 | V-ELM-2 |
|---|---|---|---|---|
| OpenMP CPU | 19.9844s | 2.3304s | 17.2394s | 18.9022s |
| CUDA GPU | 1.9802s | **0.3063s** | 1.9960s | 2.4501s |
| speedup CPU-GPU | 10.1x | 7.6x | 8.6x | 7.7x |
| **Indian Pines** | SVM | ELM | V-ELM-1 | V-ELM-2 |
| OpenMP CPU | 2.6980s | 1.1653s | 4.5749s | 9.6903s |
| CUDA GPU | 0.4548s | **0.3096s** | 0.8032s | 2.6058s |
| speedup CPU-GPU | 5.9x | 3.8x | 5.7x | 3.7x |

Table 3: Speedups frente SVM para las imágenes Pavia Univ. e Indian Pines.

| | Pavia Univ. | | | Indian Pines | | |
|---|---|---|---|---|---|---|
| | ELM | V-ELM-1 | V-ELM-2 | ELM | V-ELM-1 | V-ELM-2 |
| OpenMP CPU | **8.6x** | 1.2x | 1.1x | 2.3x | 0.6x | 0.3x |
| CUDA GPU | **6.5x** | 1.0x | 0.8x | 1.5x | 0.6x | 0.2x |

La configuración V-ELM-2 proporciona resultados más estables que un ELM individual a cambio de un tiempo de ejecución mayor. La configuración V-ELM-2 es interesante cuando el dataset es grande ya que si es demasiado pequeño (como en el caso de Indian Pines) no

hay suficientes muestras de entrenamiento para sacar ventaja del voto de la mayoría para mejorar los resultados de precisión. Además, en datasets grandes, el algoritmo ELM tiene mejores *speedups* frente a SVM lo que permite a las configuraciones con voto de la mayoría ejecutarse en casi el mismo tiempo que un único SVM, como se muestra en la tabla 3.

En resumen, por un lado, el algoritmo ELM descrito en este artículo es significativamente más rápido que SVM y, por otro lado, el algoritmo V-ELM-2 siempre se aproxima o mejora la precisión del ELM individual. Esta última es también una buena configuración si queremos priorizar los tiempos de ejecución.

# 6 Conclusiones

En este artículo hemos presentado una implementación GPU basada en ELM para clasificar eficientemente datasets hiperespectrales tomando ventaja de los cientos de *threads* disponibles, usando memoria compartida para hacer un uso efectivo de la jerarquía de memoria y explotando una librería de álgebra lineal para tomar ventaja de la arquitectura de la GPU. También se han considerado distintas configuraciones de agrupamientos para mejorar las precisiones de clasificación.

Los resultados han demostrado que las GPUs de consumo como la GTX Titan usada en este trabajo son buenos candidatos para reducir los tiempos de computación con el objetivo de conseguir procesado hiperespesctral en tiempo real. Para el ELM individual y los datasets Pavia Univ. e Indian Pines, se han alcanzado *speedups* de 7.6× y 3.8×, respectivamente, comparando con la clasificación ELM en CPU. Los resultados también muestran que la clasificación de datasets hiperespectrales usando ELM es más rápida que utilizando SVM, hasta 8.6× más rápida en CPU y 6.5× en GPU.

# Agradecimientos

Este trabajo ha sido financiado en parte por el Ministerio de Ciencia e Innovación, Gobierno de España, cofinanciado por los fondos FEDER de la Unión Europea, bajo contrato TIN 2010-17541, y por la Xunta de Galicia, Programa para la Consolidación de Grupos de Investigación Competitivos ref. 2010/28. Javier agradece el soporte económico de la Xunta de Galicia, bajo una beca predoctoral.

# Bibliografía

[1] Farid Melgani and Lorenzo Bruzzone, "Classification of hyperspectral remote sensing images with support vector machines," *IEEE Trans. Geosci. Remote Sens.*, vol. 42, no. 8, pp. 1778–1790, 2004.

[2] Mahesh Pal, "Extreme-learning-machine-based land cover classification," *International Journal of Remote Sensing*, vol. 30, no. 14, pp. 3835–3841, 2009.

[3] Dora B Heras, Francisco Argüello, and Pablo Quesada-Barriuso, "Exploring elm-based spatial–spectral classification of hyperspectral images," *International Journal of Remote Sensing*, vol. 35, no. 2, pp. 401–423, 2014.

[4] Guang-Bin Huang, Dian Hui Wang, and Yuan Lan, "Extreme learning machines: a survey," *International Journal of Machine Learning and Cybernetics*, vol. 2, no. 2, pp. 107–122, 2011.

[5] Yuan Lan, Yeng Chai Soh, and Guang-Bin Huang, "Ensemble of online sequential extreme learning machine," *Neurocomputing*, vol. 72, no. 13, pp. 3391–3395, 2009.

[6] Nan Liu and Han Wang, "Ensemble based extreme learning machine," *IEEE Signal Process. Lett.*, vol. 17, no. 8, pp. 754–757, 2010.

[7] Jiuwen Cao, Zhiping Lin, Guang-Bin Huang, and Nan Liu, "Voting based extreme learning machine," *Information Sciences*, vol. 185, no. 1, pp. 66–77, 2012.

[8] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew, "Extreme learning machine: theory and applications," *Neurocomputing*, vol. 70, no. 1, pp. 489–501, 2006.

[9] Guang-Bin Huang, Lei Chen, and Chee-Kheong Siew, "Universal approximation using incremental constructive feedforward networks with random hidden nodes," *IEEE Trans. Neural Netw.*, vol. 17, no. 4, pp. 879–892, 2006.

[10] Pierre Courrieu, "Fast computation of moore-penrose inverse matrices," *arXiv preprint arXiv:0804.4809*, 2008.

[11] Guang-Bin Huang, Hongming Zhou, Xiaojian Ding, and Rui Zhang, "Extreme learning machine for regression and multiclass classification," *IEEE Trans. Syst., Man, Cybern. B*, vol. 42, no. 2, pp. 513–529, 2012.

[12] Mark van Heeswijk, Yoan Miche, Erkki Oja, and Amaury Lendasse, "Gpu-accelerated and parallelized elm ensembles for large-scale regression," *Neurocomputing*, vol. 74, no. 16, pp. 2430–2437, 2011.

[13] Louisa Lam and Ching Y Suen, "Application of majority voting to pattern recognition: an analysis of its behavior and performance," *IEEE Trans. Syst., Man, Cybern. A*, vol. 27, no. 5, pp. 553–568, 1997.

[14] Nvidia, *NVIDIA GeForce GTX 750 Ti Whitepaper*, 2012.

[15] Nvidia, *CUBLAS Library User Guide*, 2013.

[16] Stanimire Tomov, Rajib Nath, Peng Du, and Jack Dongarra, "Magma users' guide," *ICL, UTK (November 2009)*, 2011.

[17] John A Richards and Xiuping Jia, *Remote sensing digital image analysis*, vol. 3, Springer, 1999.

[18] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, Philadelphia, PA: Society for Industrial and Applied Mathematics, third edition, 1999.

[19] Yuliya Tarabalka, Jocelyn Chanussot, and Jón Atli Benediktsson, "Segmentation and classification of hyperspectral images using minimum spanning forest grown from automatically selected markers," *IEEE Trans. Syst., Man, Cybern. B*, vol. 40, no. 5, pp. 1267–1279, 2010.

[20] Computational Intelligence Group from the Basque University (UPV/EHU), "Hyperspectral remote sensing scenes," 2014.