POSGRADO EN INFORMÁTICA

PROYECTO FIN DE MÁSTER

UNIVERSIDAD DE ALMERIA

ESCUELA POLITÉCNICA SUPERIOR

MÁSTER EN INFORMÁTICA AVANZADA E INDUSTRIAL

"Optimización del método LRASR mediante arquitecturas heterogéneas"

Curso 2015/2016

Alumno:
Luis Ortega López

Directoras:
Gracia Ester Martín Garzón
Gloria Ortega López

Optimización del método LRASR mediante arquitecturas heterogéneas

Luis Ortega López
Directoras: Gracia Ester Martín Garzón y Gloria Ortega López

Resumen—**La detección de anomalías es un campo de gran importancia para el procesado de imágenes hiperespectrales. Este trabajo presenta varias aproximaciones para acelerar el método LRASR de detección de anomalías en imágenes hiperespectrales (HSI), con foco en aquellas secciones del método que consumen la mayor parte del tiempo de procesamiento (principalmente la operación de descomposición en valores singulares, SVD). La aceleración de la operación SVD se ha llevado a cabo utilizando diversas librerías que permiten explotar tanto plataformas GPU como heterogéneas (CPU-GPU). Se han llevado a cabo diversos experimentos sobre imágenes test para observar el impacto del uso de dichas librerías sobre el SVD. Dichos experimentos han demostrado que las librerías heterogéneas para computar el SVD presentan factores de aceleración interesantes cuando estamos trabajando con los mayores tamaños de las imágenes test.**

Palabras clave—**Optimización, computación GPU computing, LRASR, imágenes hiperespectrales, SVD.**

Abstract— **Anomaly detection is very important in hyperspectral images (HSI) processing. This work shows several strategies to accelerate the LRASR anomaly detection method for hyperspectral images, with focus on the most time-expensive sections of the algorithm (mainly the singular value decomposition operation, SVD). To achieve this acceleration, many libraries have been used to exploit the power of GPU and CPU-GPU heterogeneous architectures. Different experiments have been studied to observe the impact of the libraries over the test images and the SVD operation in their processing. These experiments have shown that heterogeneous libraries for computing the SVD present notable accelerations when working with the largest test images.**

Palabras clave—**Optimization, GPU computing, LRASR, HSI, SVD.**

I. Introducción

LAS IMÁGENES HIPERESPECTRALES (Hyperspectral images, HSIs) concentran abundante información sobre las características espectrales de los materiales, con cientos o incluso miles de bandas cubriendo longitudes de onda específicas [5]. El espectro de cada pixel hiperespectral puede ser visto como un vector, donde cada componente representa la luminosidad del valor de la reflectancia para cada banda espectral. Como se observa en la Fig. 1, todas las bandas de una imagen hiperespectral representan una misma escena, pero cada una de ellas contiene información de un rango de longitudes de onda diferente, que pueden cubrir tanto el espectro visible como el infrarrojo. El ancho de cada banda puede estar entre 5 y 10nm según el sensor utilizado. Cada material arroja un perfil de reflectancia distinto para el conjunto de las bandas. Así, para cada punto de la imagen disponemos de una curva espectral que nos proporciona una gran cantidad de información para el punto correspondiente en la escena observada.

Existen muchas aplicaciones que aprovechan la gran cantidad de información que nos proporcionan los sensores hiperespectrales. Esta gran cantidad de información disponible nos lleva a desarrollar nuevas técnicas de procesamiento. Además, muchas aplicaciones que trabajan con imágenes hiperespectrales requieren una respuesta rápida. Ejemplos de estas aplicaciones pueden ser las obtenidas en áreas del modelado y evaluación medioambiental, detección de objetivos militares o la prevención y respuesta a riesgos, como incendios forestales, operaciones de rescate, inundaciones o amenazas biológicas [10] [45].

La detección de anomalías sobre imágenes hiperespectrales es considerada como uno de estos grandes ámbitos de aplicación. La detección de anomalías es una tarea importante, con un gran interés en el campo del procesado de imágenes hiperespectrales. Podemos considerar que un algoritmo detector de anomalías es aquel que nos indica, partiendo de una imagen de entrada y sin ninguna otra información a priori, aquellos elementos espaciales que son espectralmente diferentes en relación a lo que le rodea espacialmente [48]. Tenemos que diferenciar este proceso de los algoritmos de detección de *targets* (objetivos) cuyo objetivo es localizar firmas espectrales específicas en una escena dada, requiriendo información adicional previa sobre la firma espectral de búsqueda [10][11][45].

Dadas las características de las imágenes hiperespectrales, que pueden llegar a ser de un gran tamaño, y la complejidad de los algoritmos de procesamiento, es imprescindible desarrollar maneras eficientes de tratar estos datos, así como desarrollar técnicas de procesado computacionalmente eficientes para manejar y extraer la información deseada de estos grandes volúmenes de datos, y que se pueda realizar en un periodo de tiempo razonable [59].

En la actualidad, y con el reciente aumento en la cantidad y en las dimensiones de los datos hiperespectrales, el procesamiento paralelo se postula como un requisito imprescindible en las aplicaciones destinadas al reconocimiento y tratamiento de estas imágenes. Más aún si lo que se persigue es el procesado de la imagen en tiempo real, es decir, procesar la imagen al mismo tiempo que se obtiene directamente desde el sensor [67].

Las arquitecturas de alto rendimiento actuales, ponen a disposición de la comunidad científica una serie de recursos computacionales que posibilitan la extensión y aceleración de las implementaciones de sus modelos. No

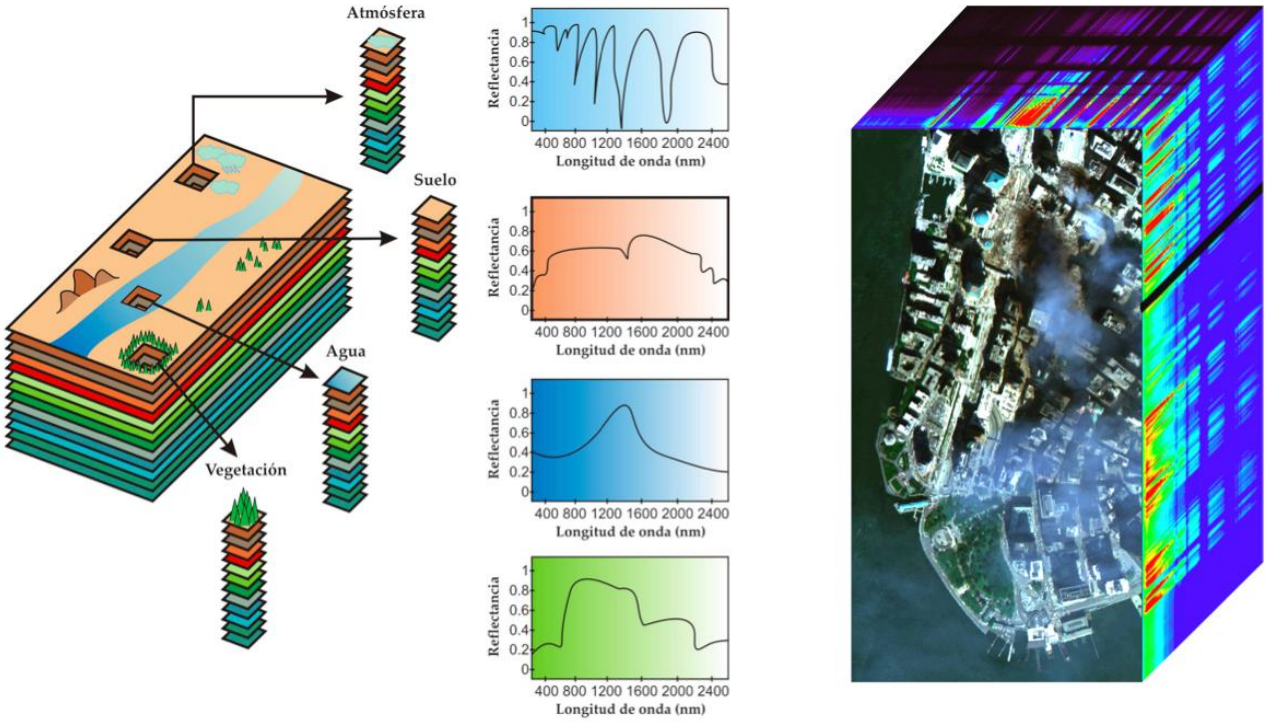


Fig. 1. Concepto de imagen hiperespectral. Cada elemento tiene un perfil de reflectancia en las distintas bandas. El conjunto de la información para cada banda define una imagen hiperespectral.[46]

obstante, es necesario un esfuerzo importante en la selección de las arquitecturas e interfaces de programación, de acuerdo con las características de estos y de los modelos matemáticos que se pretenden implementar.

Como plataformas de computación de altas prestaciones, nos centraremos en el uso y la explotación de arquitecturas y recursos que podemos encontrar en un ordenador de alta gama, sin necesitar el acceso a recursos de grandes centros de cálculo ni a grandes clústeres dedicados a supercomputación. En concreto la arquitectura multicore de los actuales procesadores y las tarjetas gráficas o GPUs son los recursos computacionales que pretendemos explotar [37][58].

Algunas ventajas de las arquitecturas multicore y de las tarjetas gráficas sobre otras plataformas son: su bajo coste, su bajo consumo energético en relación a la potencia y rendimiento que nos pueden ofrecer y su uso extendido. Sin embargo, una explotación eficiente de estas arquitecturas requiere su estudio y conocimiento, y en ocasiones la combinación de distintos modelos de programación como: modelos de paso de mensajes (MPI); de memoria compartida (OpenMP, PThreads, TBB, etc); o el uso de frameworks o APIs (Application Programming Interfaces) como OpenCL o CUDA, que permiten la programación de dispositivos de procesamiento gráfico o GPUs [31][53].

Este trabajo se centra en un método reciente de detección de anomalías en imágenes hiperespectrales, denominado método LRASR (*low-rank and sparse representation, LRASR*) [70]. Este método ha resultado ser muy prometedor, pero a pesar de obtener mejores rendimientos de detección, sus tiempos de procesamiento están muy por encima del resto de métodos clásicos. Este trabajo analizará el perfil computacional del método y estudiará diversas estrategias de aceleración para poder obtener unos tiempos de procesamiento menores. Otro de los objetivos de este trabajo es analizar el estado actual de las implementaciones paralelas sobre una de las operaciones más costosas del método, la operación SVD (*Singular Value Decomposition*).

Este documento se estructura en las siguientes secciones. En la Sección II A se presentan varios algoritmos para la detección de anomalías sobre imágenes hiperespectrales. En la Sección II B se describe el método LRASR. En la Sección III se explican las principales características de la computación de altas prestaciones y las diferentes arquitecturas de computadores. En la sección IV se detalla el proceso seguido para la caracterización del problema, que como veremos consume la mayor parte de su tiempo en la operación SVD, y se explican las particularidades de la aceleración de dicha operación. En la Sección V se describen las herramientas empleadas para los desarrollos realizados, cuya implementación se explica en la Sección VI. La Sección VII está dedicada a comentar los resultados obtenidos. Finalmente, en la Sección VIII se resumen las conclusiones y en la Sección IX se proponen líneas de trabajos futuros.

II. DETECCIÓN DE ANOMALÍAS: MÉTODO LRASR

En esta sección se detallan diversos métodos para la detección de anomalías y, finalmente, se destaca el método LRASR, cuya aceleración es uno de los objetivos de este trabajo.

A. Estado del arte de los métodos de detección de anomalías en HSI

Uno de los principales métodos de detección de anomalías es el algoritmo Reed-Xiaoli (RX) [63], que está basado en la suposición de que el fondo de la escena sigue una distribución normal multivariante. El detector RX utiliza la función de densidad de probabilidad de la distribución normal multivariante para medir la probabilidad de que un pixel de test dado forme parte del fondo.

Para la detección de anomalías es importante definir cómo cuantificamos lo anómalo que resulta un pixel dado. Para conocer esto, observamos que la solución de la relación de semejanza generalizada del test RX resulta ser la distancia de Mahalanobis entre los vectores espectrales del pixel de entrada y sus píxeles vecinos, según la siguiente expresión:

$$\delta^{RX}(\mathrm{x}) = (\mathrm{x}-\mu)^T \mathrm{K}^{-1}(\mathrm{x}-\mu)$$

donde x es un pixel hiperespectral o vector del tamaño de las bandas que contiene la imagen, μ es la media de la muestra de la imagen hiperespectral y K es la matriz de covarianza correspondiente a la muestra. Como se puede ver, $\delta^{RX}(\mathrm{x})$ es la conocida como distancia de Mahalanobis. Esta distancia es la magnitud que sirve para cuantificar lo anómalo que resulta un pixel respecto de sus pixeles vecinos. De esta forma, las imágenes generadas por el algoritmo RX son generalmente imágenes de escala de grises y las anomalías se pueden clasificar en términos de valor devuelto por $\delta^{RX}(\mathrm{x})$, de modo que el píxel con un mayor valor de $\delta^{RX}(\mathrm{x})$ puede ser considerado como la primera anomalía, y el siguiente valor la segunda anomalía y así sucesivamente.

Sin embargo, en un escenario hiperespectral real, una distribución normal multivariante es demasiado simple para describir un fondo de escenario complejo. Además, la existencia de ruido y otros píxeles anómalos puede provocar que la matriz de covarianza estimada y el vector normal no sean formas precisas de representación de un fondo complejo [70].

Para evitar esta limitación, existen algunos métodos mejorados respecto a RX. Por ejemplo, el método RX-regularizado regulariza la matriz de covarianza estimada para todos los píxeles de la HSI [50]. El método RX-segmentado es un método recientemente propuesto que utiliza agrupamiento de todos los píxeles de la imagen [49]. Los métodos RX-ponderado y RX-basado en filtro lineal se describen en [30], y buscan mejorar la estimación de la información del fondo. Además, los métodos basados en núcleo como el RX-núcleo [26], [36] y la descripción de los datos de los vectores de apoyo [2], [64] fueron propuestos basándose en la teoría del núcleo para extender el espacio original a un espacio de mayor dimensión. Estos métodos pueden manejar datos de muy alta dimensionalidad.

Recientemente, se han propuesto algunos métodos que no se basan en RX. Un detector de anomalías basado en una selección aleatoria ha sido descrito en [20]. Mediante la selección aleatoria de píxeles de fondo y empleando un número suficiente de selecciones aleatorias, es posible reducir la contaminación de las estadísticas del fondo. El método de discriminación de anomalías basado en aprendizaje de métricas se describe en [21]. Este método utiliza una métrica robusta para incrementar la separación entre los píxeles de la anomalía y otros píxeles de fondo utilizando información discriminativa. Algunos métodos basados en subespacios son presentados en [23] y [62]. Dichos métodos pretenden capturar variaciones espectrales locales. Además, se ha desarrollado un método de detección de anomalías multiventana en [43].

Aparte de los métodos mencionados anteriormente, recientemente algunos métodos basados en representación han ganado mucha atención. Estos métodos asumen que las firmas hiperespectrales pueden ser representadas utilizando un diccionario. Con diferentes restricciones en los coeficientes de representación pueden conseguirse diferentes detectores. El detector basado en representación dispersa (sparse-representation-based detector, SRD) se introduce para la detección supervisada de objetivos hiperespectrales [14], [15] y asume que cada muestra puede ser representada por unos pocos átomos del diccionario. Un átomo es un elemento dentro del diccionario que es independiente al resto y que nos sirve para describir mediante una combinación de varios de ellos cualquier elemento en la imagen. Un detector basado en representación colaborativa ha sido propuesto en [38], donde se considera que cada pixel en el fondo puede ser representado por sus vecinos espaciales, mientras que esto no es posible con las anomalías. La representación es la combinación lineal de los píxeles vecinos, y la colaboración entre estos píxeles es reforzada por la minimización mediante la norma ℓ_2 de la representación del vector de pesos. Sin embargo, ninguno de estos métodos toma la correlación de todos los píxeles de la HSI en consideración, por lo que la información global no es tenida en cuenta en ninguno de ellos.

Recientemente, un nuevo método de detección de anomalías basado en la representación dispersa y de bajo rango (low-rank and sparse representation, LRASR) ha sido propuesto [70]. De forma opuesta al resto de métodos mencionados, la base del mismo es la separación de la parte de la anomalía y de la parte del fondo, y la parte de la información del fondo está contenida en la representación de bajo rango de los píxeles de la HSI. La representación de bajo rango (low-rank representation, LRR) [41], [42] puede ser utilizada para encontrar la representación de menor rango de todos los píxeles de forma conjunta. De esta manera, la parte correspondiente a la anomalía puede ser obtenida mediante un residual de la imagen original y la parte del fondo recuperada, utilizando la representación de menor rango.

Así, la relación de todos los píxeles de la HSI puede ser caracterizada desde un punto de vista global. Para una mejor representación, la estructura local de los píxeles de cada coeficiente es de gran importancia. En dicho método LRASR se ha diseñado un criterio de dispersión para caracterizar la estructura local y el conjunto de datos de la imagen, que da una representación precisa de los datos observados [70].

El diccionario del fondo tiene un gran impacto en la capacidad de representación. Para la detección de anomalías, el diccionario debería estar formado por los

píxeles de fondo y cubrir todas las diferentes posibles clases de fondo.

Por ello, una nueva estrategia de construcción de diccionarios es propuesta en este método, de tal manera que pueda realizarse una discriminación más estable y de mayor precisión. Las principales contribuciones del método LRASR se pueden resumir en [70]:

1. Se utiliza una estrategia LRR para propósitos de reconocimiento de anomalías en HSI. La información del fondo es caracterizada por los coeficientes de menor rango, y la información de la anomalía está contenida en el residual.
2. Para describir mejor la estructura de la representación de cada pixel, se introduce un regularizador para inducir a la dispersión en el método propuesto, resultando en una representación mucho más precisa.
3. La construcción del diccionario tiene en cuenta dos factores: el primero es que el diccionario está compuesto de los píxeles del fondo, y el segundo es que contiene todas las clases posibles de fondo.

Tras el método LRASR también han surgido otros métodos, por ejemplo, el método LRRaLD [51]. Este método también hace uso de una matriz de bajo rango para el fondo y separa las anomalías en una matriz dispersa pero, a diferencia del LRASR, utiliza un diccionario aprendido. En la Fig. 2 podemos observar una comparativa entre algunos de los métodos mencionados. Observamos cómo el método LRASR es muy competitivo en la precisión que alcanza, pero se ve limitado por unos tiempos de cálculo superiores a métodos similares. Estos valores dan una idea de la precisión del método mediante el uso de la curva ROC (*Receiver Operating Charasteristic*) [35] y el AUC (*Area Under Curve*) correspondiente a la curva ROC. En la Fig. 2 también aparecen otros métodos de detección de anomalías, como el método GRX [63], el método CRD [38] y otros métodos basados en LRMD (*low-rank matrix decomposition*) al igual que el LRASR como el RPCA [13], LRaSMD [66], además del LRRaLD [51].

B. Descripción del método LRASR

1) LRR para la detección de anomalías

Consideremos una imagen hiperespectral X, donde N es el número de pixeles en el espacio de la imagen y B el número de bandas de cada uno de los píxeles. Los N píxeles forman una imagen HSI de la forma $\mathrm{X} = \{x_i\}_{i=1}^{N} \in \mathbb{R}^{B\times N}$. En las imágenes HSI, un pixel anómalo debe ser diferente a los píxeles de fondo mediante un perfil espectral sustancialmente distinto. Además, existe una fuerte correlación entre los píxeles del fondo, de tal manera que, por ejemplo, es posible representar los píxeles del fondo mediante otros píxeles similares del fondo. Esto significa que la matriz X puede ser descompuesta en dos partes, la parte del fondo y la parte anómala, del siguiente modo:

$$\mathrm{X} = \mathrm{DS} + \mathrm{E} \tag{1}$$

donde DS representa la información del fondo, $\mathrm{D} = [d_1, d_2, \ldots, d_M]$ es el diccionario del fondo formado por los píxeles de fondo (siendo M el número de átomos en el diccionario), $\mathrm{S} = [s_1, s_2, \ldots, s_N]$ refleja los coeficientes de representación, y $\mathrm{E} = [e_1, e_2, \ldots, e_N]$ denota la parte restante correspondiente a las anomalías. Esto significa que la información original puede ser reconstruida mediante el diccionario del fondo [70].

Existen muchas soluciones factibles al problema (1). Para dirigir el problema, necesitamos algún criterio con el objetivo de caracterizar las matrices S y E. Por un lado, sabemos que sólo una pequeña fracción de los píxeles pertenecerán a anomalías, por lo que la matriz E será dispersa. Por otro lado, el espectro de cada pixel corresponde a un tipo de material (lo que se conoce como "pixel puro") o bien a una mezcla de varios materiales (lo que se conoce como "pixel mezclado"). Como es posible representar en un subespacio el espectro de cada material, todos los espectros de la HSI pueden ser dibujados a partir de múltiples subespacios. Por este motivo, la matriz S debería dar la representación de menor rango del conjunto de todos los espectros. En resumen, para la matriz $\mathrm{X} = [x_1, x_2, \ldots, x_N]$ con cada x_i representando el pixel número *i*, es posible inferir las anomalías resolviendo el siguiente problema LRR [41]:

$$\min_{\mathrm{S,E}} \operatorname{rank}(\mathrm{S}) + \lambda \|\mathrm{E}\|_{2,1}$$

$$\text{tal que} \quad \mathrm{X} = \mathrm{DS} + \mathrm{E} \tag{2}$$

donde $\operatorname{rank}(\cdot)$ representa la función rango, el parámetro $\lambda > 0$ es utilizado para balancear los efectos de las dos partes, y $\|\cdot\|_{2,1}$ es la norma $\ell_{2,1}$ definida como la suma de la norma ℓ_2 de las columnas de una matriz.

$$\|\mathrm{E}\|_{2,1} = \sum_{i=1}^{N} \sqrt{\sum_{j=1}^{d} ([\mathrm{E}]_{j,i})^2} \tag{3}$$

Different Data Sets		LRRaLD	GRX	CRD	RPCA	LRaSMD	LRASR
Hyperion	AUC	**0.9998**	0.9978	0.9992	0.9902	0.9981	0.9993
	Time(s)	41.25	5.90	48.73	21.91	42.51	336.5
HYDICE	AUC	**0.9988**	0.9872	0.9961	0.9793	0.8918	0.9874
	Time(s)	18.14	0.87	33.23	6.53	18.81	140.5

Fig. 2. Comparativa de precisión (AUC) y tiempo de ejecución de diferentes métodos de detección de anomalías sobre dos ejemplos diferentes [51].

donde $[\mathrm{E}]_{j,i}$ es la entrada de E. La norma $\ell_{2,1}$ fomenta a las columnas de E a ser cero, lo cual asume que sólo hay algunos valores corruptos (o en nuestro caso, anomalías) y que son específicos de la muestra. Para un vector columna correspondiente al pixel número i, una magnitud mayor implica que el pixel es más anómalo. Como consecuencia, la matriz E mide las anomalías de manera natural. De forma diferente al método propuesto en [66] que utiliza un análisis robusto de componentes principales (robust principal component analysis, RPCA) [8] para separar la información original a una parte de bajo rango y una parte de error disperso, este método utiliza LRR para separar la información. Como apunta [41], el RPCA asume que la información reside en un único subespacio de bajo rango. Sin embargo, debido a la existencia de píxeles mezclados, los píxeles de una HSI son dibujados a partir de múltiples subespacios. Es por esto que puede no ser apropiado utilizar RPCA en este contexto. Por el contrario, si se elige un diccionario apropiado, el LRR (que puede ser visto como una generalización del RPCA) puede recuperar los múltiples subespacios subyacentes.

Sin embargo, resolver (2) resulta en un problema NP-complejo. Afortunadamente, los métodos de rellenado de matrices [8] sugieren que el siguiente problema de optimización convexa es un buen sustituto del problema (2):

$$\min_{\mathrm{S,E}} \|\mathrm{S}\|_* + \lambda\|\mathrm{E}\|_{2,1}$$
$$\text{tal que} \quad \mathrm{X} = \mathrm{DS} + \mathrm{E} \tag{4}$$

donde $\|\cdot\|_*$ denota la norma nuclear de la matriz (suma de los valores singulares de una matriz) [24]. Una vez terminado el proceso de representación, las anomalías para cada espectro $T(\mathrm{x_i})$ pueden ser determinadas mediante la respuesta de la matriz residual E de la siguiente manera:

$$T(\mathrm{x_i}) = \|[E^*]_{:,i}\|_2 = \sqrt{\textstyle\sum_j ([E^*]_{j,i})^2} \tag{5}$$

donde $\|[E^*]_{:,i}\|_2$ denota la norma ℓ_2 de la i-ésima columna de E^*. Si este valor es superior a un umbral, $\mathrm{x_i}$ es reconocido como un pixel anómalo.

2) Regularización dispersa para LRR

Como puede comprobarse en [41], el criterio de bajo rango es superior cuando se captura la estructura global de los datos X observados. Sin embargo, cada espectro tiene su propia estructura local. Cuanto más precisa sea la descripción de la estructura local, mayor será la precisión de los datos observados. La representación dispersa de la señal ha resultado en una herramienta poderosa en varias áreas [4], [69]. Este éxito es debido principalmente a que la mayoría de las señales naturales pueden ser representadas de forma dispersa mediante unos pocos coeficientes que contengan la información más relevante respecto a cierto diccionario o conjunto de bases [69]. En detección de anomalías en HSI los algoritmos dispersos han sido ampliamente utilizados. El detector básico basado en dispersión utiliza un modelo disperso similar al propuesto en [69] para representar de forma dispersa una imagen de test mediante unas pocas muestras de entrenamiento, incluyendo muestras de anomalía y muestras de fondo, y después emplea los residuales de reconstrucción para realizar la detección. En la fase de detección de la anomalía, como la mayoría de las muestras corresponden a píxeles del fondo, dichas anomalías tienen una representación dispersa en términos del diccionario del fondo. De esta forma, la naturaleza dispersa de la matriz nos permite describir la estructura local. El modelo con regularización dispersa puede expresarse de la siguiente forma:

$$\min_{\mathrm{S,E}} \|\mathrm{S}\|_* + \beta\|\mathrm{S}\|_1 + \lambda\|\mathrm{E}\|_{2,1}$$
$$\text{tal que} \quad \mathrm{X} = \mathrm{DS} + \mathrm{E} \tag{6}$$

donde $\|\cdot\|_1$ es la norma ℓ_1 de una matriz, es decir, la suma del valor absoluto de todos los elementos en la matriz; y $\beta > 0$ es un parámetro para compensar el bajo rango y la dispersión. El modelo (6) incorpora la estructura de forma global mediante la propiedad del bajo rango, e incorpora la estructura de forma local mediante la propiedad de la dispersión. Así, la matriz E proporciona una mejor descripción de las anomalías.

El problema LRASR (6) puede ser resuelto mediante el popular método de alternancia de dirección [39], [41]. Sin embargo, es necesario introducir dos variables auxiliares cuando se resuelve (6), y además en cada iteración es necesario llevar a cabo inversiones de matrices muy costosas. Debido a esto, recientemente se ha desarrollado un nuevo método denominado método linealizado de alternancia de dirección con penalización adaptativa (*linearized alternating direction method with adaptive penalty*, LADMAP) [40] que es utilizado para resolver (6).

Para poder separar la función objetivo introducimos una variable auxiliar J, la cual cumple $\mathrm{S} = \mathrm{J}$; entonces podemos reemplazar el segundo término $\|\mathrm{S}\|_1$ en la función objetivo con $\|\mathrm{J}\|_1$. De esta manera, el problema original (6) puede ser transformado en el siguiente problema:

$$\min_{\mathrm{S,E}} \|\mathrm{S}\|_* + \beta\|\mathrm{J}\|_1 + \lambda\|\mathrm{E}\|_{2,1}$$
$$\text{tal que} \quad \mathrm{X} = \mathrm{DS} + \mathrm{E}, \mathrm{S} = \mathrm{J} \tag{7}$$

La función Lagrangiana aumentada del problema (7) es

$$L(\mathrm{S,J,E,Y_1,Y_2},\mu) = \|\mathrm{S}\|_* + \beta\|\mathrm{J}\|_1 + \lambda\|\mathrm{E}\|_{2,1} + \langle \mathrm{Y_1}, \mathrm{X} - \mathrm{DS} - \mathrm{E}\rangle + \langle \mathrm{Y_2}, \mathrm{S} - \mathrm{J}\rangle + \frac{\mu}{2}(\|\mathrm{X} - \mathrm{DS} - \mathrm{E}\|_F^2 + \|\mathrm{S} - \mathrm{J}\|_F^2) = \|\mathrm{S}\|_* + \beta\|\mathrm{J}\|_1 + \lambda\|\mathrm{E}\|_{2,1} + f(\mathrm{S,J,E,Y_1,Y_2},\mu) - \frac{\mu}{2}(\|\mathrm{Y_1}\|_F^2 + \|\mathrm{Y_1}\|_F^2) \tag{8}$$

Donde $(\mathrm{Y_1}, \mathrm{Y_2})$ son los multiplicadores de Lagrange, $\mu > 0$ es el parámetro de penalización y

$$f(\mathrm{S,J,E,Y_1,Y_2},\mu) = \frac{\mu}{2}(\|\mathrm{X} - \mathrm{DS} - \mathrm{E} + \mathrm{Y_1}/\mu\|_F^2 + \|\mathrm{S} - \mathrm{J} + \mathrm{Y_2}/\mu\|_F^2) \tag{9}$$

LADMAP es un problema múltiple de optimización de variables, el cual puede ser resuelto mediante la actualización de una variable de forma alterna minimizando L con las otras variables fijadas. Suponiendo que estamos en la k-ésima iteración, el problema puede ser dividido en los siguientes subproblemas:

1) Se fijan E y J, y se actualiza S. La función objetivo puede escribirse de este modo:

$$S_{k+1} = \arg\min_S \|S\|_* + \langle \nabla_{Sf}(S_k, J_k, E_k, Y_{1,k}, Y_{2,k}, \mu_k), S - S_k \rangle + \frac{\eta_1 \mu_k}{2}(\|S - S_k\|_F^2) = \arg\min_S \|S\|_* + \langle \nabla_{Sf}(S_k, J_k, E_k, Y_{1,k}, Y_{2,k}, \mu_k), S - S_k \rangle + \frac{\eta_1 \mu_k}{2} \times \|S - S_k + [-D^T(X - DS_k - E_k + Y_{1,k}/\mu_k) - (S_k - J_k + Y_{2,k}/\mu_k)]/\eta_1\|_F^2 \quad (10)$$

Donde el término cuadrático *f* es reemplazado por su aproximación de primer orden de las iteraciones previas, y después se le añade un término proximal [40], ∇_{Sf} es el diferencial parcial de *f* respecto a S y $\eta_1 = \|D\|_2^2$.

2) Se fijan E y S, y se actualiza J. La función objetivo puede escribirse de este modo:

$$J_{k+1} = \arg\min_J \beta\|J\|_1 + \frac{\mu_k}{2}\|S_{k+1} - J_k + Y_{2,k}/\mu_k\|_F^2 \quad (11)$$

3) Se fijan J y S, y se actualiza E. La función objetivo puede escribirse de este modo:

$$E_{k+1} = \arg\min_E \lambda\|E\|_{2,1} + \frac{\mu_k}{2}\|X - DS_{k+1} - E + Y_{1,k}/\mu_k\|_F^2 \quad (11)$$

La solución se expone en el Algoritmo 1.

Algoritmo 1 Algoritmo LADMAP para LRASR

Entradas: Matriz de datos X, parámetros $\beta > 0$, $\lambda > 0$
Inicialización: $S_0 = J_0 = E_0 = Y_{1,0} = Y_{2,0}$, $\mu_0 = 0.01$, $\mu_{max} = 10^{10}$, $\rho_0 = 1.1$, $\varepsilon_1 = 10^{-6}$, $\varepsilon_2 = 10^{-2}$, $\eta_1 = \|D\|_2^2$, $k = 0$.

1: while $\|X - DS_0 - E_0\|_F/\|X\|_F \geq \varepsilon_1$ or $\mu_k \max(\sqrt{\eta_1}\|S_k - S_{k-1}\|_F, \|J_k - J_{k-1}\|_F, \|E_k - E_{k-1}\|_F) / \|X\|_F \geq \varepsilon_2$ **do**

2: Actualizar S_{k+1}:

$$S_{k+1} = \theta_{(\eta_1\mu_k)^{-1}}(S_k + [D^T(X - DS_k - E_k + Y_{1,k}/\mu_k) - (S_k - J_k + Y_{2,k}/\mu_k)]/\eta_1)$$

3: Actualizar J_{k+1}:

$$J_{k+1} = \mathcal{S}_{\beta\mu_k^{-1}}(S_{k+1} + Y_{2,k}/\mu_k)$$

4: Actualizar E_{k+1}:

$$E_{k+1} = \Omega_{\lambda\mu_k^{-1}}(X - DS_{k+1} + Y_{1,k}/\mu_k)$$

5: Actualizar los multiplicadores de Lagrange:

$$Y_{1,k+1} = Y_{1,k} + \mu_k(X - DS_{k+1} - E_{k+1})$$

$$Y_{2,k+1} = Y_{2,k} + \mu_k(S_{k+1} - J_{k+1})$$

6: Actualizar μ de la siguiente forma:

$$\mu_{k+1} = \min(\mu_{max}, \rho\mu_k) \text{ donde}$$

$$\rho = \begin{cases} \rho_0, si\ \mu_k \max(\sqrt{\eta_1}\|S_{k+1} - S_k\|_F, \|J_{k+1} - J_k\|_F, \\ \quad \|E_{k+1} - E_k\|_F)/\|X\|_F \geq \varepsilon_2 \\ 1\ , \text{ para el resto de casos} \end{cases}$$

7: Actualizar $k : k \leftarrow k + 1$.
8: end while
Salida: Una solución óptima (S_k, J_k, E_k).

En el Algoritmo 1, θ es el operador umbral de valor singular [7] que requerirá el uso de una operación de descomposición en valores singulares de alta complejidad, $\mathcal{S}$ es el operador de contracción [39] y Ω el operador de minimización $\ell_{2,1}$ [41][70].

3) Construcción del diccionario D

En el algoritmo mencionado anteriormente, el diccionario D tiene un rol muy importante en vista a la detección de anomalías. En la representación dispersa para búsqueda de objetivos, el diccionario se compone del diccionario del fondo y el diccionario de objetivos, los cuales son conocidos a priori. Esto, sin embargo, no ocurre en el caso de la detección de anomalías, donde el diccionario es desconocido a priori y debería representar la información del fondo tanto como sea posible.

Una manera de construir este diccionario es utilizar los datos X originales de forma directa. Aunque los datos X originales contienen anomalías, el número de píxeles anómalos es muy reducido, por lo que pueden ser ignorados en el proceso de representación. Sin embargo, el coste computacional es enorme, ya que el algoritmo requiere la descomposición en valores singulares de una matriz de tamaño $M \times M$, donde M es el número de átomos. Otra manera de generar el diccionario, es elegir algunos de los píxeles de la HSI de forma aleatoria para disminuir el número de átomos, pero la desventaja de este método es que el diccionario no contendría todas las clases de fondo. Afortunadamente, en HSI la mayor parte de la imagen está compuesta por unas pocas clases de materiales muy frecuentes, de tal manera que los píxeles tienen una gran probabilidad de pertenecer a esos materiales comunes. Al construir el diccionario de esta manera, la probabilidad de escoger píxeles de estos materiales frecuentes será muy elevada, por lo que el diccionario estará compuesto por muestras de estos materiales frecuentes, y el resto de materiales poco frecuentes será ignorado. Como resultado, los píxeles que correspondan a materiales con un número menor de muestras representativas serán detectados como anomalías.

En [70] se adopta una estrategia novedosa para la construcción del diccionario. En primer lugar, se utiliza *K*-means para dividir los píxeles en K grupos (denominados clústeres), $X = \{X^1, X^2, X^3, \dots, X^K\}$. La distancia euclídea es utilizada en el algoritmo *K*-means. K debería ser mayor que el verdadero número de tipos de

materiales de suelo para tener la certeza de que los K clústeres cubren todos los tipos de materiales de suelo. Como el átomo debería ser un pixel de fondo, se adopta una estrategia predictiva para elegir los píxeles de fondo de cada clúster. De forma similar al algoritmo RX, se calcula el cuadrado de la distancia de Mahalanobis entre el pixel de test y la media local del fondo. El detector RX asume que cuanto mayor sea el valor del resultado de la detección, será mucho más probable que el pixel resulte anómalo. Sin embargo, los píxeles de ruido o los píxeles correspondientes a materiales raros pueden ser detectados como valores atípicos al ser menos frecuentes en la HSI.

Por otro lado, los píxeles con bajos valores pre-predictivos serán píxeles del fondo con certeza. Así, los P píxeles que arrojen la menor distancia de Mahalanobis serán escogidos para generar átomos en el diccionario. Si el número total de píxeles en el clúster es menor que P, este clúster puede ser ignorado, ya que tenemos un set K mayor que el número de tipos de materiales de suelo.

El algoritmo de detección de anomalías para HSI basado en LRASR utilizando el diccionario clusterizado se resume en el Algoritmo 2 [70].

Algoritmo 2 Algoritmo de detección de anomalías HSI basado en LRASR

Entradas: Matriz de datos X, parámetros $\beta > 0$, $\lambda > 0$, K, P

1: Dividir la matriz de datos X en K partes utilizando el algoritmo K-means, tal que:
$\mathrm{X} = \bigcup_{i=1,\dots,K} \mathrm{X}^i, \mathrm{X}^i \cap \mathrm{X}^j = \emptyset, i \neq j, i,j = 1, \dots K,$
Siendo $N_{i=1,\dots,K}$ el número de píxeles en X^i y $D = \emptyset$.

2: for $i = 1{:}K$

2.1: if $N_i < P$ Saltar directamente al paso 2.2;

2.2: calcular el vector medio μ y el $\sum$ de la matriz de covarianza de los datos $\{\mathrm{x}_i | \mathrm{x}_j \in \mathrm{X}^i, j = 1 \dots N_i\}$.

2.3: calcular el resultado de la pre-detección:

$$PD(\mathrm{x}_j) = (\mathrm{x}_j - \mu)^T \sum^{-1} (\mathrm{x}_j - \mu) j = 1, 2, \dots, N_i$$

2.4: escoger P píxeles $\mathrm{D}^i = [\mathrm{x}_1, \mathrm{x}_2, \dots, \mathrm{x}_P]$ en $\{\mathrm{x}_j | \mathrm{x}_j \in \mathrm{X}^i, j = 1 \dots N_i\}$, tal que:
$PD(\mathrm{x}_i) < PD(\mathrm{x}_j), \mathrm{x}_i \in D^i,$
$\mathrm{x}_j \in \{\mathrm{x}_j | \mathrm{x}_j \in \mathrm{X}^i, j = 1 \dots N_i\} \backslash D^i$

2.5: $\mathrm{D} = \mathrm{D} \cup \mathrm{D}_\mathrm{i}$

end

3: Resolver el siguiente problema utilizando el Algoritmo 1,

$$\min_{\mathrm{S,E}} \|\mathrm{S}\|_* + \beta \|\mathrm{S}\|_1 + \lambda \|\mathrm{E}\|_{2,1}$$
$$\text{tal que} \quad \mathrm{X} = \mathrm{DS} + \mathrm{E}$$

Y obtener la solución óptima (S^*, E^*).

4: calcular $T(\mathrm{x_i}) = \|[E^*]_{:,i}\|_2 = \sqrt{\sum_j ([E^*]_{j,i})^2}$, $i = 1,2, \dots, N$

Salida: Mapa de detección de anomalías.

El método LRASR tiene un gran coste computacional, en concreto podemos destacar como muy costosas las operaciones umbral de valor singular, θ, el operador de contracción, $\ell_{2,1}$, o la propia generación del diccionario, por lo que su implementación requiere del uso de arquitecturas HPC (*High-Performance Computing*).

III. COMPUTACIÓN DE ALTO RENDIMIENTO

Durante los últimos años ha crecido el interés por el desarrollo de software que explote los recursos de los computadores actuales. Estos están caracterizados por la heterogeneidad, que se manifiesta tanto en la capacidad computacional de cada elemento de proceso, como en la interconexión de estos elementos. Los sistemas heterogéneos más extendidos son los clústeres de multiprocesadores de memoria compartida, donde cada nodo posee una arquitectura multicore con distinto número de núcleos, además puede disponer de diversos elementos aceleradores, como unidades vectoriales, FPGAs, GPUs, y una red de interconexión heterogénea compuesta por enlaces de distinto ancho de banda y latencia [16] [25] [56][57].

La explotación de las arquitecturas de alto rendimiento requiere la combinación de distintos modelos de programación paralelos, tales como: modelos de paso de mensajes (MPI); de memoria compartida (OpenMP, PThreads, TBB) [6][12][33][54] o los denominados modelos PGAS (Partitioned Global Address Space), desarrollados para facilitar la producción de software para clústeres de multiprocesadores. Algunos ejemplos de interfaces de programación paralela que se ajustan al modelo PGAS son: Unified Parallel C (UPC), X10, Chapel [9][22][60]. Por otra parte, para la explotación de las distintas arquitecturas aceleradoras, se han desarrollado APIs (Application Programming Interfaces) que facilitan su uso en aplicaciones de propósito general. Así por ejemplo, se han desarrollado diversos interfaces para la programación de GPUs: Compute Unified Device Architecture (CUDA) de NVIDIA [52], ATI Stream SDK de AMD [1] y OpenCL, que se presenta como el estándar para la programación de GPUs [Khro09].

En el contexto de: análisis de imágenes hiperespectrales y otros conjuntos de datos de gran resolución, sofisticados métodos de filtrado de imágenes, segmentación, etc. se han implementado códigos en paralelo capaces de explotar plataformas de supercomputación. De este modo, se ha podido reducir el tiempo del procesado o tratar problemas que, por sus dimensiones, anteriormente eran inabordables. La mayoría de estas implementaciones requieren ser ejecutadas en supercomputadores, que forman parte de centros de supercomputación y que ofrecen sus servicios a la comunidad científica. Sin embargo, desde un punto de vista práctico, y dejando de lado los altos costes económicos, es difícil poner en marcha un sistema de computación paralela cada vez que se requiera trabajar con imágenes hiperespectrales. En este contexto, resulta de interés destacar que los computadores actuales se configuran, se gestionan y se usan fácilmente por parte de cualquier usuario, y contienen una tremenda potencia de computación gracias a varios factores [57]. Los computadores actuales tienen incorporada la capacidad de multiprocesamiento simétrico, gracias a la inclusión de varios núcleos de procesamiento en un mismo microchip (CMP, Chip

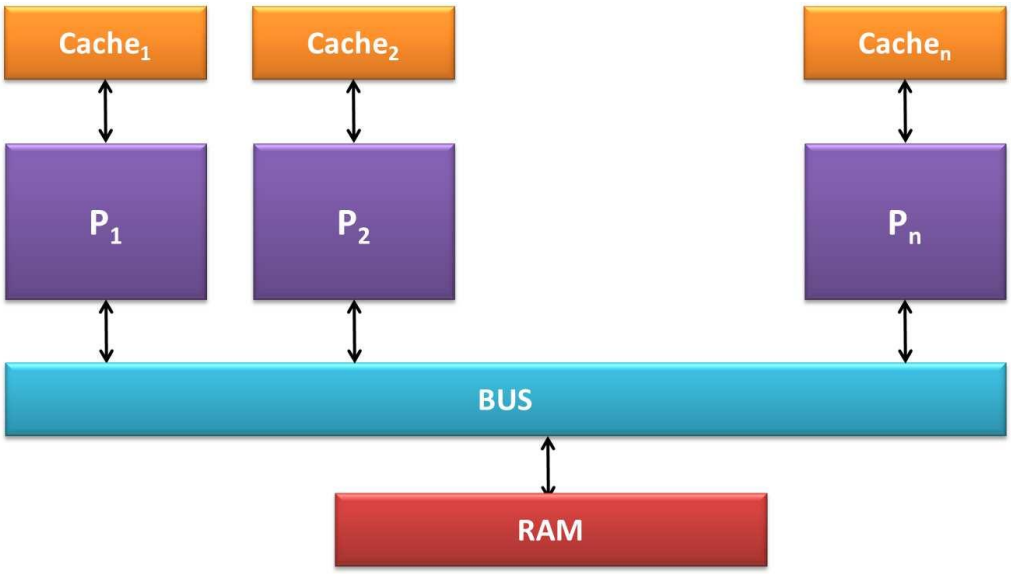


Fig. 3. Arquitectura de memoria compartida

Multi-Core, [25]). Además, suelen incluir una o varias plataformas GPUs que pueden ser utilizadas para el cómputo general.

Las siguientes subsecciones están dedicadas a mostrar las principales características de los sistemas multicore y de las GPUs. Además, se explica la interfaz CUDA para utilizar la potencia de cálculo de las GPUs en aplicaciones de propósito general.

A. Introducción a la computación multicore

Los multiprocesadores de memoria compartida constan de varios procesadores conectados al mismo bus, a través del cual comparten una memoria común. En esta arquitectura, cuando es necesario resolver un problema, éste se asigna sobre los procesadores disponibles, con el objetivo de reducir al mínimo el tiempo de ejecución total. Con este modelo, las variables compartidas de un programa están disponibles para cualquier procesador, en cualquier momento. Las comunicaciones entre procesadores se realizan mediante estas variables compartidas, coordinando los accesos por medio de procesos de sincronización que permiten al sistema resolver las dependencias de datos en el programa.

Los sistemas de memoria compartida pueden ser tanto SIMD como MIMD. Los procesadores vectoriales de una sola CPU pueden considerarse como un ejemplo de SIMD, mientras que los modelos multi-CPU son ejemplos de MIMD. En ocasiones, se utilizan las abreviaturas SM-SIMD y SM-MIMD para las dos subclases [19].

La Fig. 3 muestra la organización tradicional de una arquitectura basada en memoria compartida. En esta arquitectura, se establece una segunda clasificación según la forma de acceder al espacio de memoria común. Los sistemas de Acceso de Memoria Uniforme (UMA) proporcionan la misma latencia para acceder a cualquier palabra de la memoria para cualquier procesador. Por otra parte, en los sistemas de Acceso a Memoria No Uniforme (NUMA), los accesos a ciertas palabras de memoria son mucho más rápidos que los accesos a otros, dependiendo del procesador que los requiera. Los multiprocesadores NUMA son más difíciles de explotar, pero su escalabilidad es mejor que para los sistemas UMA.

Merece la pena destacar que las instrucciones de tipo SIMD son capaces de abordar múltiples operaciones de un mismo tipo de datos por unidad de tiempo (p.ej. las instrucciones MMX, SSE y SSE2 de Intel y 3DKnow! de AMD). Debido a que estos procesadores tienen varios núcleos (por ejemplo: 4 u 8), este tipo de arquitecturas poseen la capacidad de ejecutar simultáneamente varios threads en cada uno de los núcleos. Este hecho junto con el desarrollo de la tecnología y la organización de memoria, ha dado lugar a una jerarquía de memoria más profunda y eficiente, haciendo posible que los threads que están ejecutándose en cada núcleo compartiendo espacio de direcciones de memoria puedan acceder a los datos de forma eficiente. Este modelo de programación basado en hilos (threads) puede explotarse mediante el uso de librerías de bajo nivel, como son PThreads [6] y OpenMP [61].

B. Introducción a la computación en GPU

Otro tipo de plataforma paralela muy interesante en este contexto, por su bajo coste y mantenimiento, así como por su enorme capacidad de procesamiento, es la basada en unidades de procesamiento gráfico (Graphics Processor Units o GPUs).

El constante desarrollo de las arquitecturas hardware de las GPUs ha permitido que estas arquitecturas aumenten de forma notable su capacidad de procesamiento en cada nueva generación. Actualmente, las GPUs ofrecen la posibilidad de ser utilizadas de una forma distinta a su objetivo principal de procesamiento gráfico, surgiendo nuevas líneas de uso en otros objetivos de propósito general como el cálculo computacional.

La alta capacidad de cálculo presente en las GPUs hace posible acelerar las operaciones computacionales, donde la GPU (denominada *device* en este ámbito) puede actuar

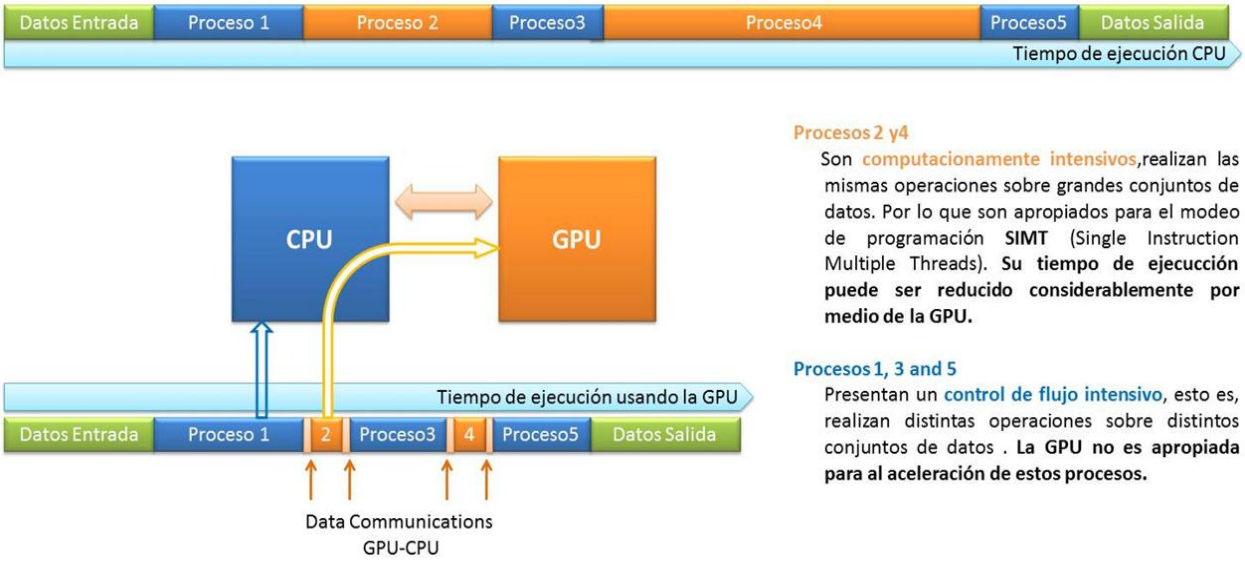


Fig. 4. Modelo computacional de la CPU (arriba) y modelo co-computacional de CPU-GPU (abajo)

como un coprocesador de la CPU. Así surge el concepto del *GPGPU Computing*, que ha motivado el uso de estos dispositivos en diversos problemas de la ciencia y la ingeniería, como la resolución de grandes sistemas de ecuaciones, simulaciones de sistemas moleculares, cálculo financiero, tratamiento de señales, modelos meteorológicos o química cuántica, entre otros muchos problemas [3],[55].

Como observamos en la Fig. 4, el modelo co-computacional hace uso del gran poder de cálculo de la GPU para acelerar procesos computacionalmente intensivos, manteniendo los procesos que requieren control de flujo intensivo (operaciones muy distintas sobre diversos conjuntos de datos) sobre la CPU.

Así mismo, han aparecido diversas herramientas que permiten explotar de forma óptima sus recursos, como el lenguaje CUDA, y diversas librerías (CUBLAS, CUSOLVER, CUSPARSE, entre otras).

C. Arquitectura de la GPU

Es importante entender la arquitectura de la GPU y sus diferencias fundamentales con la arquitectura de la CPU para poder apreciar sus ventajas computacionales. Debido a que cada arquitectura surge para una finalidad específica, su filosofía de diseño es sensiblemente diferente, y también serán distintas sus prestaciones.

Una GPU está diseñada con el objetivo principal de realizar la reconstrucción y recomposición de imágenes, aplicando de forma simultánea la misma operación sobre grandes volúmenes de datos. Esto es una aplicación directa del paradigma SIMT (*Simple Intruction Multiple Threads*) extensión del paradigma SIMD (*Simple Instruction Multiple Data*). Este tipo de sistemas cuenta con miles de cores para poder realizar estas ejecuciones paralelas masivas. Las instrucciones ejecutadas quedan determinadas por aquellas instrucciones elementales que son requeridas para el procesamiento gráfico. Debido a su uso en operaciones computacionales distintas al procesamiento gráfico, se han añadido unidades para la manipulación de valores en punto flotante de doble precisión. Así mismo, la lógica de control también resulta mucho más sencilla, permitiendo el cambio de contexto de un thread en un sólo ciclo de reloj.

La CPU, sin embargo, es un elemento de propósito general, por lo que permite un conjunto de instrucciones mucho mayor al de las GPUs. Esto obliga a que su lógica de control sea más compleja, con el requisito de un mayor coste para el cambio de contexto de un thread. Su número de cores, aun tratándose de una CPU multicore, es muy

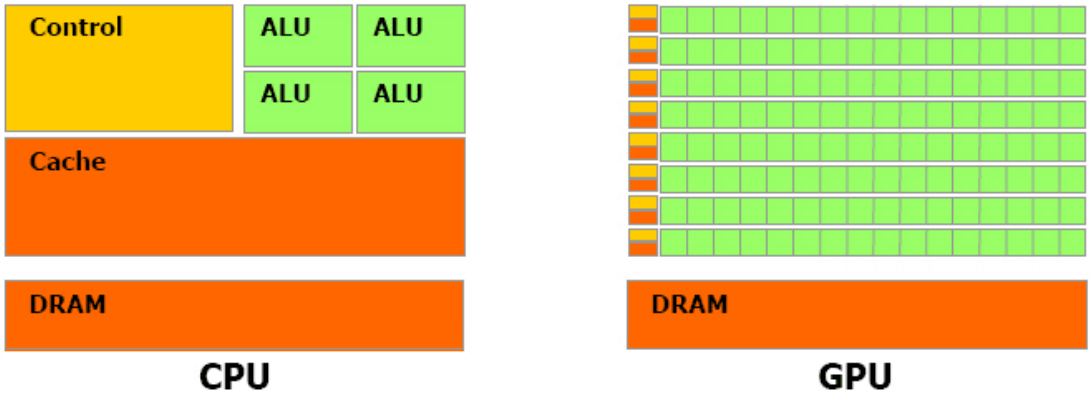


Fig. 5. Diferencias entre CPU y GPU.

inferior al de una GPU, ya que no se trata de un sistema orientado a procesos masivamente paralelizables.

En la Fig. 5 podemos observar cuáles son las principales diferencias en la memoria entre ambas arquitecturas. El procesamiento SIMT de la GPU no requiere de gran capacidad de almacenamiento en las memorias caché, aunque sí va a requerir de su uso para enmascarar la alta latencia de acceso a la memoria del device. Esta memoria, denominada GDDR (*Graphics Double Data Rate*), tiene un gran ancho de banda para la transferencia de datos, pero su latencia de acceso y su capacidad de almacenamiento es inferior que el de la memoria DDR SDRAM del PC (denominado *host* en este ámbito).

La CPU dispone de una memoria caché de mayor tamaño, ya que su estrategia computacional busca explotar la localidad temporal y espacial de los datos que utiliza cada uno de sus cores de forma independiente.

Pese a que cualquier algoritmo que se pueda implementar en una CPU también se puede codificar en una GPU, esas implementaciones no serán igual de eficientes en las dos arquitecturas. En concreto, los algoritmos con un alto grado de paralelismo, sin necesidad de estructuras de datos complejas y con una alta intensidad aritmética son los que mayores beneficios obtienen de su implementación en la GPU.

La comunicación hacia la GPU se realiza mediante el bus PCI-Express, conectado al puente norte del sistema. A dicho puente se conecta también la CPU y se regula la comunicación con el resto de dispositivos presentes en el host. No es posible acceder desde la CPU a la memoria global del device (GDDR), ni tampoco es posible para la GPU acceder a la memoria de la CPU, para ello habrá que hacer un volcado entre las memorias mediante el bus PCI-Express.

El modelo computacional de la GPU está definido mediante una agrupación de multiprocesadores (SM, *Stream Multiprocessor*), constituidos por procesadores escalares (SP, *Scalar Processor*). Cada SP ejecuta un thread, formando la unidad básica de procesamiento, y es controlada por la unidad de instrucción (*Thread Processor*). El Thread Processor es común para todos aquellos SP de un mismo SM, al igual que la caché de instrucciones. Referente a las memorias, cada SP dispone de varios registros de 32 bits de uso estrictamente local, y cada SM dispone de una memoria compartida (*memory on-chip*) para todos los SP.

Existe además una memoria de constantes y otra de texturas (*memory off-chip*), accesibles para todos los SM para lectura. Finalmente está la memoria DRAM GDDR para operaciones de lectura y escritura que es accesible por todos los SM.

D. Interfaces de programación con la GPU: CUDA

CUDA es una herramienta de programación y un lenguaje desarrollados por NVIDIA para poder utilizar la potencia de cálculo que ofrecen las GPUs en aplicaciones de propósito general. El SDK de CUDA está compuesto por compiladores de C/C++ y FORTRAN, diversas librerías adaptadas para GPU, archivos cabecera y demás herramientas que permiten elaborar código, depurarlo y analizar su rendimiento sobre la arquitectura de la GPU.

En la Fig. 6 podemos observar la representación esquemática de los fragmentos de código que se ejecutan en la GPU en el modelo CUDA, denominados *kernels*. Cada uno de estos kernel está compuesto por un conjunto bloques denominado *grid*. Cada uno de los bloques de este grid está compuesto a su vez de multitud de threads, la unidad básica de ejecución en un SP. La unidad básica de ejecución dentro de un SM se denomina *warp*, que es un grupo de 32 threads. Todos los warps de un bloque se ejecutan de forma concurrente en el SM.

Un programa CUDA se ejecutará por lotes, donde el SM reserva en una cola activa grupos de bloques para ejecutarlos en paralelo. Cada core en el device podrá ejecutar varios warps de un mismo bloque de forma simultánea. Sin embargo, el hecho de introducir sentencias condicionales en nuestro código puede provocar que los warps no se ejecuten de forma simultánea. Esta divergencia computacional hace que una parte de los warps se ejecute y otra se quede esperando a que la condición de bifurcación se vea modificada. Para no perder el alto grado de paralelismo que nos aporta CUDA, es fundamental reducir al mínimo las instrucciones condicionales a ejecutar en la GPU.

Durante la programación con la API de CUDA, es habitual tener que realizar un volcado entre los datos del host y el device. El host puede escribir y leer sobre la memoria global del device, así como en la memoria de constantes y de texturas. Es esta separación del espacio de direccionamiento de la memoria principal del host y la memoria global del device lo que obliga a realizar constantes volcados de memoria en ambas direcciones, del host al device y viceversa.

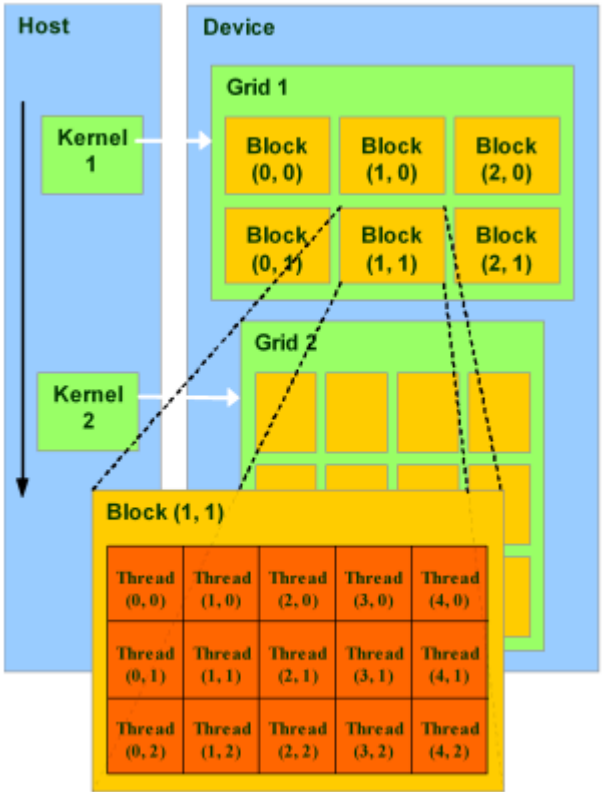
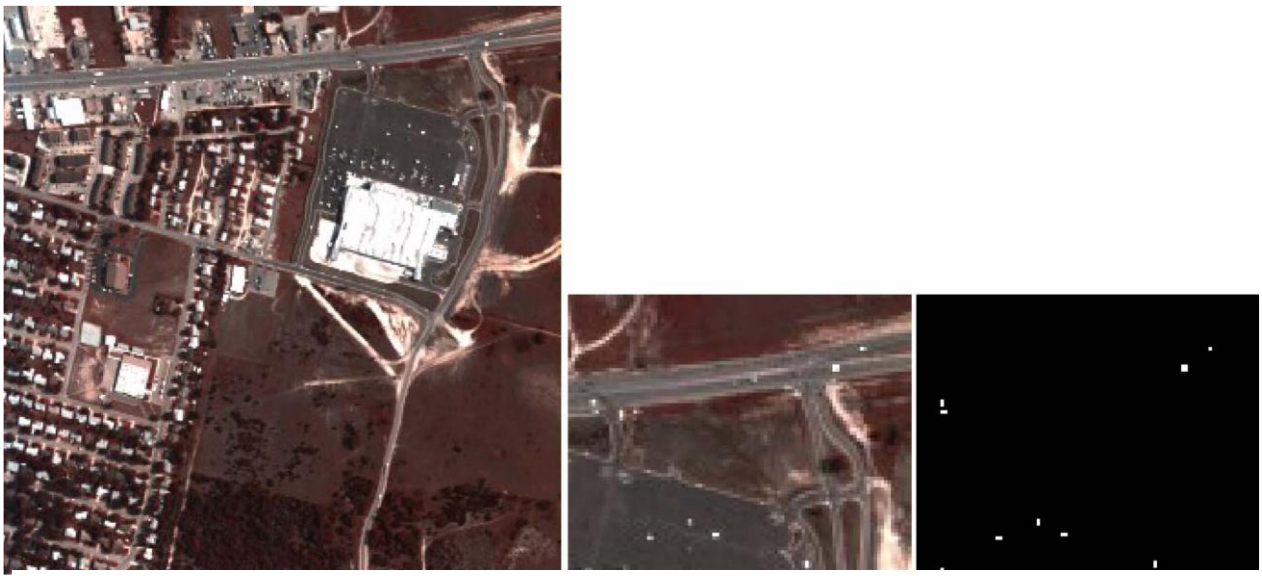


Fig. 6. Esquema de un Kernel en CUDA.



(a) (b) (c)

Fig. 7. Ejemplo de datos reales utilizados. (a) Imagen en falso color de la escena, (b) imagen en falso color del área de detección, (c) mapa de verdad de anomalías [70].

IV. CARACTERIZACIÓN DEL PROBLEMA

En vista a realizar una aceleración del método LRASR, es necesario comprobar su perfil computacional para determinar aquellas secciones del código que tienen un alto coste computacional y una baja diversificación de operaciones y orígenes de datos.

A. Ejemplo de estudio

Para la elaboración del presente trabajo, se ha tomado como imagen test para su caracterización uno de los ejemplos de [70]. El ejemplo se basa en una imagen hiperespectral real de un entorno urbano, correspondiente a uno de los ejemplos de test de [70]. Se trata de un conjunto de datos hiperespectrales HYDICE, obtenido de una plataforma aérea. Cubre un área urbana que comprende parte de vegetación, áreas de construcción y varias carreteras que contienen vehículos. La imagen tiene una resolución espectral de 10nm y una resolución espacial de 1m. Eliminando las bandas de absorción del vapor de agua y aquellas de baja relación SNR (1–4, 76, 87, 101–111, 136–153, y 198–210), permanecen 162 bandas. El conjunto completo de datos tiene un tamaño de 307x307 píxeles, tal como se muestra en la Fig. 7(a).

El mapa de verdad de anomalías muestra que los objetivos anómalos son los tejados y los coches de la sección superior derecha de la escena. La sección considerada consiste en los píxeles que cubren esta región. Una representación en color y el mapa de verdad de anomalías se muestra en la Fig. 7(b) y (c) respectivamente. Los 21 píxeles anómalos objetivo son los vehículos de diferentes tamaños de la escena urbana. [70]

B. Caracterización computacional

Se han realizado experimentos en la máquina Bullx para modelar el coste computacional del problema en vista a obtener una primera aproximación.

La máquina Bullx R424-E3 posee las siguientes características técnicas: 2 Intel Xeon E5 2650 (16 cores) y 64 GB de RAM, 128 GB SSD con una GPU Tesla M2070 (1.792 cores).

Se ha partido de la implementación MATLAB de LRASR desarrollada por los autores de [70].

Hemos comprobado que el método converge en 105 iteraciones para el ejemplo dado, y que el tiempo total de ejecución sobre la máquina descrita corresponde a 121,11 seg de media. Para poder comprobar dónde reside la carga computacional se ha hecho uso de la herramienta profiler de MATLAB. La ejecución del ejemplo de los autores tiene el perfil de carga indicado en la Tabla I.

Comprobamos que dentro del ejemplo, el método LRASR es el que ocupa la mayor parte del tiempo de cálculo. El desglose del método LADMAP_LRASR se presenta por rutinas y operaciones en la Tabla II.

TABLA I

PERFIL DE TIEMPOS OBTENIDO CON LA HERRAMIENTA PROFILER DE MATLAB. SE INDICA LA FUNCIÓN LLAMADA, LA RUTINA DESDE LA QUE ES INVOCADA Y EL TIEMPO CONSUMIDO.

Función llamada	Descripción	Tiempo empleado (seg.)
LADMAP_ LRASR	Algoritmo LRASR	113,95
DicCon	Creación del diccionario	5,52
RxDetector	Detección Reed-Xiaoli	0,95
Resto		0,69
Total		**121,11**

TABLA II

PERFIL DE TIEMPOS DE LA RUTINA LADMAP_LRASR OBTENIDO CON LA HERRAMIENTA PROFILER DE MATLAB. SE INDICA LA LÍNEA LLAMADA, EL NÚMERO DE VECES QUE ES EJECUTADA Y EL TIEMPO CONSUMIDO.

Línea	Nº Ej.	Tiempo empleado (seg.)	% rutina
Operación SVD	105	47,41	41,61%
Subrutina solve_l1l2 (normalizaciones)	105	27,69	24,30%
Cálculo de S temporal (varias multiplicaciones)	105	10,38	9,11%
Cálculo de condición de salida (operaciones varias)	105	7,74	6,79%
Resto de líneas	-	20,72	18,19%
Total	-	**113,95**	100,00%

Podemos observar como una parte muy notable del tiempo de cálculo corresponde a una línea en concreto, donde se realiza una operación SVD (Singular Value Decomposition). Las otras partes más costosas y computacionalmente intensivas son la creación del diccionario y la rutina *solve_l1l2*.

De cada una de estas tres operaciones podemos destacar los siguientes aspectos:

Operación SVD. Esta operación realiza la descomposición en valores singulares de una matriz de valores reales o complejos. Se trata de un tipo de factorización que en el ejemplo a optimizar se realiza sobre una matriz de valores de doble precisión de dimensiones $M \times N$. Con M el número de átomos en el diccionario y n la cantidad de píxeles en la imagen. En la imagen de test, esta matriz acaba resultando en una matriz densa de bajo rango con dimensiones 300×8000. La operación SVD es la más costosa de todo el método, y su aceleración no es trivial, ya que existe dependencia en las operaciones internas de este método y, por tanto, su grado de paralelización es limitado.

Rutina *solve_l1l2*. Esta rutina actualiza la matriz E para cada iteración, tal como se puede apreciar en el detalle del código MATLAB 1. La rutina recibe una matriz temporal W, que divide en columnas y las envía a la subrutina *solve_l2*. Esta subrutina opera para cada columna, que corresponde a un pixel (el número de columnas n es igual al número de píxeles de la imagen), y devuelve para cada uno:

- Una columna de ceros, si no supera cierto umbral marcado por su norma vectorial.
- Una columna de valores determinada por el operador de minimización $\ell_{2,1}$ [41], si supera cierto umbral marcado por su norma vectorial.

Esta rutina realiza una operación que, a priori, parece altamente paralelizable, ya que el resultado de cada columna es independiente del resto de columnas, pudiéndose hacer un batch de operaciones norma vectorial por pixel, y tras ello un batch de decisiones por pixel.

Detalle del código MATLAB 1 Sección del código MATLAB de la rutina solve_l1l2 y de la subrutina solve_l2

Entradas: Matriz W ($B \times N$, con dimensiones el número de bandas, B, y el número de píxeles en la imagen, n), constante lambda.

```
function [E] = solve_l1l2(W,lambda)
n = size(W,2);
E = W;
for i=1:N
    E(:,i) = solve_l2(W(:,i),lambda);
end

function [x] = solve_l2(w,lambda)
% min lambda |x|_2 + |x-w|_2^2
nw = norm(w);
if nw>lambda
    x = (nw-lambda)*w/nw;
else
    x = zeros(length(w),1);
end
```

Salida: Valor de E ($B \times N$, con dimensiones el número de bandas, B, y el número de píxeles en la imagen, N) actualizado para cada iteración.

Rutina DicCon. Esta rutina realiza la creación inicial del diccionario según el método explicado en la Subsección II B (apartado 3). Este algoritmo, que se puede observar en el detalle del código MATLAB 2, recibe como entrada los datos en la matriz $X(B \times N)$, con la notación establecida anteriormente. Divide esa imagen en un número de clústeres K utilizando el algoritmo *k-means*, y para cada clúster realiza una detección para calcular los P píxeles de menor distancia de Mahalanobis, es decir, más cercanos al valor medio y por tanto con mayor probabilidad de ser fondo. Si un clúster tiene menos de P píxeles será ignorado. Esto nos devuelve como máximo los K grupos de P píxeles más representativos del fondo, generando así el diccionario. Esta rutina también puede ser altamente paralelizable al operar por regiones, pudiendo hacer un batch de cálculo de distancias para cada clúster K de forma independiente.

Detalle del código MATLAB 2 Sección del código MATLAB de la rutina DicCon para la creación del diccionario.

Entradas: Matriz X ($B \times N$, con dimensiones el número de bandas, B, y el número de píxeles en la imagen, N), número de clusters, K, y píxeles escogidos por cluster, P.

```
function Dictionary = DicCon(X, K, P)
 [K1 K2] =
kmeans(X',K,'Start','cluster');
Dictionary = [];
```

```
for i=1:K
    st1=find(K1==i);
    if length(st1)<P
        continue;
    end
    temp = X(:,st1);
    kr = RxDetector(X(:,st1));
    [d1 d2] = sort(kr,'ascend');
    Dictionary= [Dictionary
,temp(:,d2(1:P))];
end
```

Salida: Diccionario del fondo de la imagen.

Aparte de las tres operaciones comentadas anteriormente, en el método LRASR se pueden observar numerosas operaciones matriz-matriz y matriz-vector, que podrían ser optimizadas mediante el uso de estrategias paralelas. Sin embargo, éstas no suponen un porcentaje relevante del tiempo total de cálculo comparado con las tres operaciones mencionadas.

Tras este análisis del ejemplo, se concluye que la mayor carga computacional reside en la operación SVD que se realiza para cada iteración del método. Por lo tanto, el foco de nuestros esfuerzos se ha centrado en acelerar esa operación, a pesar de que, a priori, no conocemos su grado de paralelización.

C. Operación SVD en HPC

La operación SVD merece especial consideración por su complejidad y su utilidad en diversos aspectos de las matemáticas aplicadas a la ciencia y la ingeniería. El SVD es una operación crítica para resolver problemas de mínimos cuadrados, determinar la pseudoinversa de una matriz o calcular aproximaciones de matrices de bajo rango, con aplicación directa a procesamiento de la señal, reconocimiento de patrones o estadística [27][28].

La operación SVD busca obtener la descomposición de la matriz A :

$$A = U * \Sigma * V^H$$

donde Σ es una matriz de elementos diagonales que corresponden a los valores singulares, U es la matriz de vectores singulares izquierda, y V^H es la matriz de vectores singulares derecha traspuesta.

El algoritmo estándar para el SVD primero reduce una matriz densa a su forma bidiagonal, para después extraer los valores singulares de la forma condensada utilizando el algoritmo QR [18][68] o "divide y vencerás" [29]. Finalmente se realizan las transformaciones ortogonales necesarias si se requiere obtener el conjunto de los vectores singulares.

Mientras la fase de reducción (también llamada *back transformation* en inglés) es rica en operaciones de nivel 3 de BLAS (operaciones matriz-matriz), la reducción a la forma bidiagonal se ve generalmente como el cuello de botella de este algoritmo para las arquitecturas HPC debido a la predominancia de operaciones de nivel 2 BLAS (operaciones matriz-vector) [65]. Este aspecto de la paralelización del SVD sobre arquitecturas HPC continúa siendo actualmente objeto de estudio.

Para entender la base algorítmica de la operación SVD en computación, es necesario conocer el concepto de algoritmos en bloque. La computación de matrices se divide en dos fases sucesivas:

1. Factorización en panel, donde las operaciones de nivel 2 BLAS se acumulan para su posterior uso en un bloque de columnas denominado "panel".
2. Actualización de la submatriz de arrastre (del inglés *trailing submatrix*), donde las transformaciones acumuladas del panel son aplicadas mediante operaciones nivel 3 de BLAS en la parte no reducida de la matriz.

Estas dos fases representan el núcleo de la metodología algorítmica para resolver sistemas lineales de ecuaciones y problemas de autovalores y descomposición en valores singulares [65].

El algoritmo SVD estándar consta de tres fases bien diferenciadas, tal como se puede observar en la Fig. 8:

a. El primer paso es reducir la matriz densa a una forma bidiagonal mediante transformaciones ortogonales.

$$A = \begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix} \xrightarrow{H_L} \begin{bmatrix} * & * & * & * \\ & * & * & * \\ & * & * & * \\ & * & * & * \end{bmatrix} \xrightarrow{H_R} \begin{bmatrix} * & * & & \\ & * & * & * \\ & * & * & * \\ & * & * & * \end{bmatrix}$$

$$\xrightarrow{H_L} \begin{bmatrix} * & * & & \\ & * & * & * \\ & & * & * \\ & & * & * \end{bmatrix} \xrightarrow{H_R} \begin{bmatrix} * & * & & \\ & * & * & \\ & & * & * \\ & & * & * \end{bmatrix} \xrightarrow{H_R} \begin{bmatrix} * & * & & \\ & * & * & \\ & & * & * \\ & & & * \end{bmatrix} \equiv B$$

Los factores ortogonales se acumulan como:
$A = U_A B V_A^H$, donde $U_A = (\prod H_L)^H$, $V_A = \prod H_R$

b. Después, se aplica un solver iterativo, como pueden ser el algoritmo QR o el algoritmo "divide y vencerás" para calcular los valores singulares y los vectores singulares asociados de esa matriz bidiagonal.

$$B = U_B \Sigma V_B^H$$

c. Finalmente, en la fase de reducción o "back transformation", se multiplican los vectores singulares que se acaban de obtener en el paso anterior con las transformaciones ortogonales acumuladas.

$$A = (U_A U_B)\Sigma(V_B^H V_A^H) = U \Sigma V^H$$

Para entender los inconvenientes del método SVD es importante conocer los dos tipos de transformaciones que existen. En primer lugar, están las transformaciones unilaterales, que se utilizan típicamente para resolver sistemas lineales de ecuaciones y problemas de mínimos cuadrados: la matriz es reducida y factorizada durante la fase de factorización en panel y esta última no requiere acceder a la matriz de datos que se encuentra fuera del panel en ese instante. Las transformaciones resultantes se aplican después a la submatriz de arrastre en el lado izquierdo.

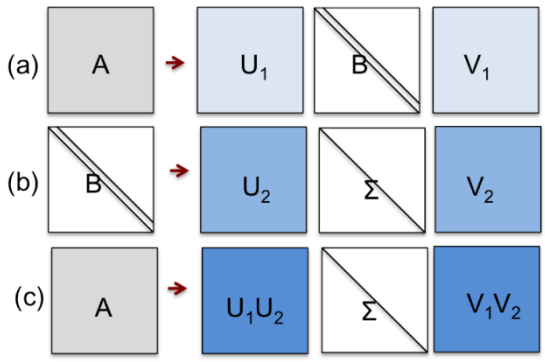


Fig. 8. Etapas computacionales de la operación SVD: (a) reducción bidiagonal, (b) solver bidiagonal del SVD y (c) fase de reducción (back transformation) [65].

Por otro lado, las transformaciones bilaterales son necesarias para los solvers de SVD y el cálculo de autovalores. Su factorización en panel es mucho más costosa que las transformaciones unilaterales, ya que requieren leer toda la submatriz de arrastre en cada actualización de columna para el panel. Una vez que el panel es reducido, las transformaciones acumuladas se aplican a ambos lados, derecho e izquierdo, de la submatriz sin reducir, lo que provoca que sea muy difícil encontrar técnicas para incrementar la concurrencia. Esto no ocurre en las transformaciones unilaterales [65].

Para ilustrar el cuello de botella de estas transformaciones bilaterales, existen ejemplos como el de la Fig. 9(a), que describe el perfil computacional del algoritmo DGESVD de la librería MAGMA (con una única GPU). Aunque sólo un 50% de las operaciones resultan en nivel 2 BLAS, consumen un 90% del tiempo total. Por otro lado puede verse en la Fig. 9(b) cuando se calculan también los vectores singulares cómo las operaciones nivel 2 BLAS, a pesar de ser sólo el 6% de los FLOPS totales, consumen el 30% del tiempo total de ejecución [65].

Por último, cabe destacar la complejidad de esta operación, en términos de número de operaciones en coma flotante (*flops*). Suponiendo una matriz cuadrada de dimensión n, la complejidad de la reducción bidiagonal utilizando reflectores de Householder tiene un coste de $\frac{8}{3}n^3$flops. Esta operación es común para las rutinas DGESVD, que utiliza el solver QR, y DGESDD, que utiliza el solver "divide y vencerás", implementadas en la librería LAPACK. Para el cálculo de los vectores singulares, si es necesario, el algoritmo requiere $O(n^2)$ flops. La estimación del coste total de las rutinas DGESVD y DGESDD es de $22n^3$ [17].

V. HERRAMIENTAS Y MÉTODOS EMPLEADOS

Como se ha visto en la sección IV, las arquitecturas de alto rendimiento ponen a disposición de la comunidad científica una serie de recursos computacionales que posibilitan la aceleración de las implementaciones de sus modelos. No obstante, es necesario un esfuerzo importante en la selección de las arquitecturas e interfaces de programación, de acuerdo con las características de estos y de los modelos matemáticos que se pretenden implementar.

Para la elaboración de este trabajo se han utilizado diversas herramientas, como las librerías CuSolver, MAGMA y el entorno de desarrollo MATLAB. A continuación se detallan las herramientas utilizadas para la implementación.

A. Librería CuSolver

Para realizar las funciones de paralelismo y optimización del problema descrito, se utilizará la librería CuSolver. Esta es una librería de alto nivel basada en las librerías cuBLAS y cuSparse, y está compuesta a su vez por tres librerías que pueden ser utilizadas de manera independiente: CuSolverDN, CuSolverSP y CuSolverRF.

El objetivo de CuSolver es proporcionar características útiles similares a LAPACK, tales como factorización común y triangulación de matrices densas o cálculo de autovalores. CuSolver proporciona una nueva librería de refactorización útil para resolver secuencias de matrices que comparten un patrón de dispersión.

La primera parte de CuSolver es CuSolverDN, que trabaja con factorización de matrices densas, resolviendo rutinas como LU, QR, SVD y LDLT, además de proporcionar otras operaciones útiles tales como permutaciones de matrices y vectores.

La segunda parte es CuSolverSP, que proporciona todo un conjunto de nuevas rutinas basadas en la factorización QR de matrices dispersas. No todas las matrices tienen un buen patrón de dispersión para poder utilizar paralelismo en su factorización, así que la librería CuSolverSP también proporciona una manera de manipular esas matrices prácticamente secuenciales desde la CPU. Para matrices con paralelismo abundante, el modo GPU supondrá un gran incremento en el rendimiento. La librería está diseñada para ser llamada desde C y C++.

La parte final es CuSolverRF, un paquete de refactorización dispersa que puede lograr muy buenos rendimientos cuando se resuelve una secuencia de matrices donde los coeficientes son variables, pero los patrones de dispersión permanecen iguales.

Esta librería requiere hardware que tenga capacidades de computación con CUDA.

B. Librería MAGMA

Como alternativa a la librería CuSolver, se ha planteado utilizar también la librería MAGMA. El proyecto MAGMA (Matrix Algebra on GPU and Multicore Architectures) [32] busca crear una librería para álgebra lineal densa similar a LAPACK, pero orientada a arquitecturas híbridas y heterogéneas, comenzando por los sistemas actuales "Multicore+GPU". Magma es un proyecto de código abierto desarrollado por el Innovative Computing Laboratory (ICL), de la Universidad de Tennessee USA).

La investigación MAGMA se basa en la idea de que, para cumplir los complejos retos de los nuevos entornos híbridos, las soluciones óptimas software deberán hibridarse también, utilizando las distintas estrategias y fortalezas de cada algoritmo en un único framework. Así, el objetivo final del proyecto MAGMA es diseñar algoritmos y frameworks de álgebra lineal para entornos

multicore donde sea posible explotar las fortalezas que cada uno de los componentes híbridos ofrece.

La interfaz, funcionalidades, y almacenamiento de datos de MAGMA están diseñados de forma similar a los de la librería LAPACK, de tal manera que los investigadores puedan trasladar fácilmente sus componentes software desde LAPACK a MAGMA en vista a obtener las ventajas de las nuevas arquitecturas híbridas.

Esta librería incluye:

- Factorizaciones LU, QR y Cholesky.
- Reducción de Hessenberg.
- Solvers lineales basados en las descomposiciones LU, QR y Cholesky.
- Solvers para el problema de los autovalores y de los valores singulares.
- Solvers basados en factorizaciones LU, QR y Cholesky.
- Solvers de refinamiento e iterativos de precision mezclada Basados en las factorizaciones LU, QR and Cholesky.

Todas las rutinas MAGMA se encuentran disponibles en los siguientes tipos de datos:

- s (float) – real en simple precisión,
- d (double) – real en doble precisión,
- c (magmaFloatComplex) – complejo en simple precisión,
- z (magmaDoubleComplex) – complejo en double-precisión

Para que la librería MAGMA pueda ser instalada y tenga una funcionalidad completa, es necesario que previamente se instalen las siguientes librerías: CUDA, BLAS, LAPACK y MKL (Math Kernel Library).

MAGMA utiliza una metodología híbrida donde los algoritmos de interés son divididos en tareas de distinta granularidad y su ejecución es planeada sobre los componentes hardware de la arquitectura. El scheduling puede ser estático o dinámico. En cualquiera de los dos casos, las pequeñas tareas no paralelizables, normalmente en el camino crítico, son ejecutadas en la CPU, y las tareas más grandes y paralelizables, a menudo del nivel 3 de las BLAS, son ejecutadas en la GPU. El esquema de la metodología híbrida puede observarse en la Fig. 10.

El nombre de las rutinas MAGMA identifica el número de GPUs que se van a utilizar en ella, así como el punto de partida de los datos que intervienen (o que estén en la CPU, o en la GPU). Centrándonos en este segundo aspecto, puede ocurrir que inicialmente los datos estén en la CPU. Entonces la CPU toma las entradas y devuelve los resultados también en la memoria de la CPU, de forma que el uso de la GPU es transparente para el usuario. Si los datos estuviesen inicialmente en la GPU, MAGMA toma las entradas y produce el resultado también en la memoria de la GPU. En ambos casos se utiliza un algoritmo híbrido CPU/GPU. De este modo, nos encontramos con que cada rutina podría tener cuatro posibles implementaciones. Se indica el tipo de implementación en el nombre de la rutina con cadena vacía, "_m", "_gpu" o "_mgpu", tal y como se puede observar en la Tabla III.

TABLA III

TIPOS DE IMPLEMENTACIONES DE LAS RUTINAS MAGMA ATENDIENDO TANTO A LA PLATAFORMA EN LA QUE ESTÁN LOS DATOS ANTES DE LLAMAR A LA RUTINA, COMO AL NÚMERO DE GPUS UTILIZADAS.

Tipos	Rutina de ejemplo	Clasificación
none	magma_dgetrf	Rutina híbrida CPU/GPU donde la matriz es inicializada en la memoria de la CPU.
_m	magma_dgetrf_m	Rutina híbrida CPU/múltiples-GPUs donde la matriz es inicializada en la memoria de la CPU.
_gpu	magma_dgetrf_gpu	Rutina CPU/GPU donde la matriz es inicializada en la memoria de la GPU.
_mgpu	magma_dgetrf_mgpu	Rutina híbrida CPU/multiple-GPU donde la matriz se distribuye entre múltiples memorias de las GPUs que se están utilizando.

Como particularidades de esta librería, merece la pena destacar que tanto ella como LAPACK, utilizan almacenamiento por columnas. Además, para matrices simétricas, hermíticas y triangulares, sólo se accede al triángulo inferior o superior, que será especificado en la rutina, mientras que el otro será ignorado.

C. Librería MKL

MKL (Math Kernel Library) es una librería que contiene funciones matemáticas optimizadas para aplicaciones científicas e ingenieriles optimizadas para procesadores Intel [34]. Incluye: BLAS, LAPACK, PARDISO, biblioteca matemática vectorial y biblioteca estadística vectorial. Multitud de librerías y entornos de desarrollo utilizan la librería MKL, tales como MATLAB o MAGMA.

D. Librería cuBLAS

La librería cuBLAS es una implementación de BLAS (Basic Linear Algebra Subprograms) sobre el entorno CUDA de NVIDIA, que permite las mismas funcionalidades que la librería BLAS estándar aceleradas mediante GPU. Permite al usuario acceder a los recursos computacionales de las GPUs de NVIDIA. Esta serie de rutinas permiten realizar operaciones sencillas tales como multiplicaciones matriz-matriz o matriz-vector, así como transformaciones sencillas sobre matrices y vectores en la GPU.

```c
#include <stdio.h>
#include <stdlib.h>
#include <mex.h>
double sumar (double oper1,double oper2){
    return (oper1+oper2);
}
void mexFunction(int nlhs, mxArray *plhs[],int nrhs, const mxArray *prhs[]){
    double val_1=0,val_2=0,val_res=0;
    val_1=mxGetScalar(prhs[0]);
    val_2=mxGetScalar(prhs[1]);
    val_res=sumar(val_1,val_2);
    plhs[0]=mxCreateDoubleScalar(val_res);
}
```

Fig. 11. Ejemplo de archivo MEX-file en lenguaje C, *"sumar.c"*.

La forma de utilizar la API de desarrollo requiere de reservar un espacio de memoria en la GPU, rellenarlo con los datos a trabajar, llamar a la secuencia de funciones cuBLAS deseadas y finalmente devolver los resultados de la GPU al host. Los datos siempre deben estar en el host, utilizando únicamente la GPU para realizar las operaciones.

Esta forma de trabajar es muy similar a la de la librería CuSolver, que extiende estas funcionalidades sencillas a operaciones más complejas. En nuestro desarrollo, utilizaremos esta librería para complementar con funciones sencillas al resto de librerías utilizadas.

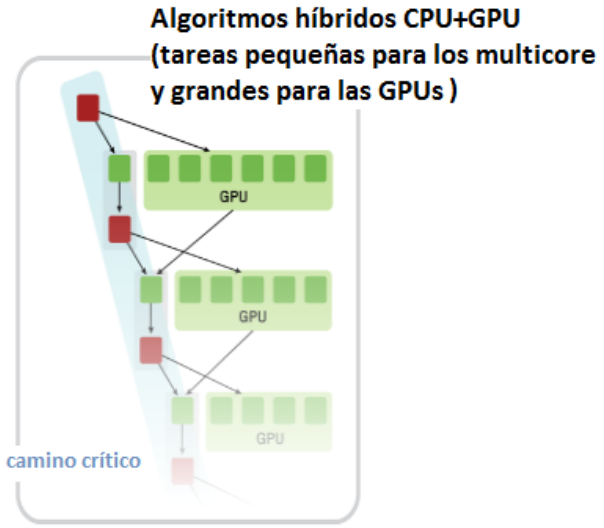


Fig. 10. Metodología híbrida de MAGMA, basada en la descomposición de tareas de distinto tamaño que serán ejecutadas en las CPUs o GPUs [32].

E. MATLAB

MATLAB (abreviatura de MATrix LABoratory, "laboratorio de matrices") es una herramienta de software matemático que ofrece un entorno de desarrollo integrado (IDE) con un lenguaje de programación propio (lenguaje M). Está disponible para las plataformas Unix, Windows, Mac OS X y GNU/Linux [47].

MATLAB puede llamar a funciones y a subrutinas escritas en C o Fortran. Se crea una función envoltorio que permite que sean pasados y devueltos tipos de datos de MATLAB. Los archivos objeto creados compilando esas funciones se denominan "MEX-files", aunque la extensión de nombre de archivo depende del sistema operativo y del procesador. Estos archivos se pueden cargar de forma dinámica en tiempo de ejecución.

Dado que el código original del problema a resolver está desarrollado en MATLAB, éste ha sido nuestro entorno de referencia para la optimización llevada a cabo. La parte fundamental que se ha desarrollado en este proyecto ha sido el uso de MEX-files para transformar secciones del código MATLAB a rutinas en C que puedan ser ejecutadas, total o parcialmente, en la GPU o en varios cores, con el fin de obtener mejoras en los tiempos de ejecución. Los MEX-files son rutinas enlazadas dinámicamente, y su extensión en el entorno MATLAB es de la forma ".mexa32" o ".mexa64" para arquitecturas de 32 ó 64 bits respectivamente.

Estos MEX-files son de interés cuando un cierto programa MATLAB quiere llamar a rutinas C, C++ o Fortran, sin necesidad de reescribir dichos códigos en un archivo ".m". También son útiles cuando se desean optimizar determinadas funciones dentro del código MATLAB que, debido a su alto coste computacional, resultan ser aceleradas cuando son ejecutadas sobre otras arquitecturas de mayor rendimiento. Los archivos MEX-file orientados a un código elaborado sobre lenguaje C disponen de las siguientes secciones, tal y como se puede visualizar en la Fig. 11:

1. Sección para incluir los archivos de cabecera, en concreto para incluir el archivo de declaración MEX-file, denominado "mex.h". Con la llamada *#include <mex.h>* nos aseguramos de que todas las estructuras necesarias para la manipulación de datos de entrada y salida entre el programa y el entorno MATLAB son declaradas.
2. La función de enlace con MATLAB, también denominada *gateway routine*, que realiza las funciones de canal bidireccional para la transferencia de datos entre la rutina implementada y el entorno MATLAB. Es la rutina principal, aunque no se denomina *main*, ya que establece el punto de entrada y salida de la aplicación en C.
3. Las rutinas computacionales. Estas rutinas están elaboradas en el lenguaje de programación definido (C/C++ o FORTRAN) y permiten hacer uso de otras librerías y herramientas computacionales,

tales como CUDA, CUBLAS, MAGMA, etc. Dichas librerías proporcionan funciones para el entorno MATLAB sin la necesidad de que dichas funciones estén implementadas en el lenguaje MATLAB.

La Gateway routine tiene una forma de implementación muy concreta sobre las reglas gramaticales de C, que se indican a continuación:

*void mexFunction(int nlhs, mxArray *plhs[], int nrhs,const mxArray *prhs[]) {…}*

donde:

- *nlhs (number of left hand side arguments)*, hace referencia al número de argumentos de salida del fichero MEX-file.
- *plhs (pointer to left hand side arguments)*, es el vector que almacena los punteros de los argumentos de salida, que serán estructuras del tipo mxArray.
- *nrhs (number of right hand side arguments)*, hace referencia al número de argumentos de entrada al fichero MEX-file.
- *prhs (pointer to right hand side arguments)*, es el vector que almacena los punteros de los argumentos de salida, que serán estructuras del tipo *mxArray*.
- *mxArray*, es la estructura que representa el tipo de dato básico del entorno MATLAB, que almacena información sobre el tipo de dato que contiene, sus dimensiones y los punteros a las direcciones de memoria donde se almacenan los datos de dicho array.

Es muy importante mencionar que las matrices que utiliza MATLAB se almacenan por columnas (formato *column-mayor*), a diferencia de C que realiza un almacenamiento por filas. Esto implica que los elementos adyacentes de un array de MATLAB corresponderán a la misma columna. Las matrices que se entregan al MEX-file van en formato array de una única dimensión, donde se encuentran almacenadas, una a continuación de otra, todas las columnas de la matriz. Por esta razón también es importante enviar información sobre las dimensiones de las matrices que se pasan al MEX-file. Conviene recordar también que la indexación de los arrays en C es de base 0, mientras que en MATLAB es de base 1, por lo que debe ser tenido en cuenta cuando se hace referencia a la misma posición de un array dentro y fuera del entorno MATLAB.

VI. IMPLEMENTACIÓN

Esta sección está dedicada a proponer distintas estrategias para acelerar la parte computacionalmente más costosa del algoritmo LADMAP. Para ello, nos hemos centrado en la operación SVD, que resulta ser la más costosa computacionalmente para el problema estudiado. Se han seleccionado dos librerías con el fin de acelerar dicha rutina: la librería CuSolver y la librería MAGMA. En las siguientes subsecciones se explica cómo se han introducido dichas librerías en el código MATLAB original.

A. Aproximación mediante la librería CuSolver

La primera aproximación al problema ha consistido en realizar la descomposición SVD y las operaciones que permiten actualizar el parámetro S en la GPU. Se han estudiado las operaciones posteriores al SVD para comprobar qué secciones adicionales de código son susceptibles de acelerar mediante GPU Computing y podrían verse beneficiadas al ya tener los datos en el device. A continuación se muestra la sección del código original a optimizar, en el detalle del código MATLAB 3:

Detalle del código MATLAB 3 Sección del código MATLAB a optimizar en GPU

Entradas: Matriz *temp*, constantes *mu* e *ita1*.

```
[U,sigma,V] = svd(temp,'econ');
sigma = diag(sigma);
svp = length(find(sigma>1/(mu*ita1)));
if svp>=1
        sigma = sigma(1:svp)-
1/(mu*ita1);
    else
        svp = 1;
        sigma = 0;
    end
S =
V(:,1:svp)*diag(sigma)'*U(:,1:svp)';
```

Salida: Valor de *S* actualizado para la iteración.

Se observa que dicho fragmento muestra la descomposición SVD y una serie de operaciones como multiplicaciones matriz-matriz y también un bucle para establecer un umbral para los valores singulares obtenidos, sustituyendo los pequeños por ceros y reduciendo los otros en un threshold (la operación θ descrita en el algoritmo del método, que es el operador umbral de valor singular [7]). Se ha realizado la implementación sustituyendo la sección descrita por una llamada a una rutina en C, utilizando un fichero MEX-file para realizar el traspaso de datos entre MATLAB y la rutina.

La parte más costosa computacionalmente de nuestro problema es la operación SVD de una matriz densa de valores de doble precisión, para la cual hemos hecho uso de la función *CuSolverDnDgesvd*. Esta función, incluida en la librería CuSolverDN, calcula la descomposición en valores singulares de una matriz A de dimensiones $M \times N$, correspondiendo los vectores singulares derecho y/o izquierdo. La operación se escribe de forma general según la siguiente expresión:

$$A(M \times N) = U(M \times M) * \Sigma(M \times N) * V'(N \times N)$$

donde Σ es la matriz $m \times n$ compuesta por ceros excepto por sus $min(M, N)$ elementos diagonales, U es una matriz

unitaria $M \times M$, y V es una matriz unitaria $N \times N$ (nótese que utilizamos la matriz traspuesta en dicho cálculo). Los elementos diagonales de Σ son los valores singulares de A. Estos valores son reales y no negativos, y se devuelven en orden descendente. Las primeras $min(M, N)$ columnas de U y V son los vectores singulares izquierdo y derecho de A.

En ocasiones, ocurre que la matriz A no se trata de una matriz cuadrada, de tal modo que $M > N$ o bien $M < N$. En estos casos, es posible utilizar una versión truncada del SVD que requiere un menor uso de memoria y, además, tiene una capacidad de computación similar a utilizar el SVD de MATLAB con el parámetro de optimización 'econ'.

Si $M < N$ tenemos el SVD con V truncada:

$$A(M \times N) = U(M \times M) * \Sigma(M \times M) * V'(N \times M)$$

Si $M > N$ tenemos el SVD con U truncada:

$$A(M \times N) = U(M \times N) * \Sigma(N \times N) * V'(N \times N)$$

La matriz Σ de nuestro problema, a la que vamos a aplicar la operación SVD, es de la forma $M < N$, ya que en nuestro ejemplo concreto es de dimensiones 300×8000, por lo que podremos truncar V.

La función *CuSolverDnDgesvd*, sin embargo, está limitada a $m \geq n$, por lo que tendremos que utilizar una pequeña estrategia matemática para poder emplearla en nuestro problema. Sea la descomposición de una matriz rectangular de la forma:

$$A(M \times N) = U(M \times M) * \Sigma(M \times N) * V'(N \times N)$$

entonces la descomposición de su matriz traspuesta será de la forma:

$$A'(N \times M) = V(N \times N) * \Sigma'(N \times M) * U'(M \times M).$$

Bastará pues con utilizar la función con la matriz traspuesta, para luego trasponer los vectores y valores singulares que nos devuelve la operación.

Por las consideraciones expuestas sobre la librería CuSolver, es necesario trasladar una matriz $M > N$, así que el paso previo a lanzar la rutina consiste en trasponer dicha matriz, intercambiando los M y N.

Detalle del código MATLAB 4 Llamada a la rutina svdCuSolver en el MEX-file desde el código MATLAB

Entradas: Matriz temp traspuesta, dimensiones N y M de dicha matriz, constantes *mu* e *ita1*, número de iteraciones *iter*.

```
[S]=svdCuSolver(temp', n, m, mu, ita1, iter);
```

Salida: Valor de *S* actualizado para la iteración.

En el detalle del código MATLAB 4 podemos ver la parte que sustituye al código original, que consiste en una llamada a un MEX-file desde el entorno MATLAB. Dicho MEX-file se ha elaborado en C para acelerar esa sección del código, como puede verse en la Fig. 12.

El código C elaborado se ha realizado utilizando el esquema clásico de un MEX-file, incluyendo los enlaces a las librerías CuSolver y CUBLAS, así como al archivo de declaración mexfile "mex.h". Los parámetros nlhs, plhs, nrhs y prhs de la mexFunction se han adecuado para referenciar las entradas y salidas descritas. Para las operaciones a realizar es necesario conocer las dimensiones de la matriz "temp", por lo que también se han pasado como parámetro M y N. El número de iteraciones, determinado por el parámetro "iter", es necesario para conocer si nos encontramos en la primera iteración, y sólo realizar la inicialización de la librería en esta iteración.

La matriz referenciada de entrada temp se pasa a la llamada *CuSolverDnDgesvd* que nos devuelve un vector de autovalores y las matrices U y V que son los vectores singulares izquierdo y derecho de la matriz temp. Para poder utilizar esta operación en la GPU, es necesario llevarnos la matriz temp a la memoria del device. Para poder mover la información del host al device y viceversa se hace uso de la función CUDA *cudaMemcpy*. Tras realizar la operación, aplicamos un bucle sencillo para hacer el thresholding utilizando los valores *mu* e *ita1*, realizando la operación θ mencionada anteriormente, y se genera la matriz Σ.

Para el resto de operaciones posteriores al SVD se ha realizado una implementación mediante algunas funciones de la librería CUBLAS. En concreto, la operación *cublasDgeam* nos permite hacer una trasposición sencilla sobre la matriz de vectores singulares izquierdos U, ya que será necesario para obtener la matriz de salida S.

En el código original, la matriz Σ se ve reducida a una matriz diagonal donde sólo los autovalores más grandes forman parte de esa diagonal, limitando sus dimensiones al parámetro *svp* (número de autovalores de la matriz temp que superan cierto umbral). La matriz Σ quedará limitada pues a las dimensiones $\Sigma(svp \times svp)$, y la operación para calcular S será de la forma:

$$S(M \times N) = V(M \times svp) * \Sigma'(svp \times svp) * U'(svp \times N)$$

Esto deja claro que sólo necesitaremos los valores de las svp primeras columnas en V y las svp primeras filas en U'. Para realizar esta simplificación se ha utilizado la función *cublasSetMatrix* de la librería CUBLAS, que permite obtener una matriz parcial a partir de otra mayor.

Para el cálculo de la matriz S es necesario realizar dos multiplicaciones matriz-matriz, una operación que tradicionalmente ha arrojado muy buenos resultados al ser acelerada en la GPU utilizando el paradigma del GPU Computing. Por ello, se han realizado ambas multiplicaciones con dos llamadas a la función *cublasDgemm* de CUBLAS, que realiza dicha multiplicación en la GPU. Finalmente, el resultado de las dos multiplicaciones sucesivas es devuelto a la memoria del host utilizando de nuevo la función cudaMemcpy.

Con la nueva matriz S calculada en el MEX-file, el programa original puede continuar y el método continúa iterando con el valor actualizado para esa iteración. Para

cada iteración se vuelve a llamar al MEX-file, que vuelve a actualizar la matriz S para la iteración. En conclusión, esa sección del código podría verse acelerada al realizarse sobre la GPU.

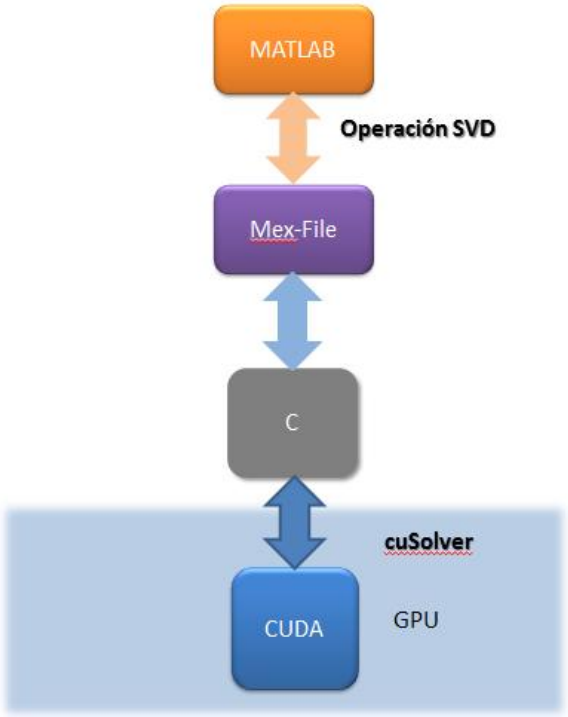


Fig. 12. Jerarquía de entornos para la aproximación mediante CuSolver.

B. Aproximación heterogénea CPU-GPU mediante la librería MAGMA

Adicionalmente, se ha realizado una integración similar para la sección de código MATLAB 3 mediante la librería MAGMA.

El principal uso de esta librería en nuestro problema será similar al de la librería CuSolver, ya que la utilizaremos para optimizar la operación SVD utilizando las funciones que la librería pone a nuestra disposición.

Existen dos funciones que nos permiten realizar esta operación. La función *magma_dgesvd* se comporta de forma similar a la función de CuSolver *CuSolverDnDgesvd*, que calcula la descomposición en valores singulares de una matriz A de valores en doble precisión de dimensiones $m \times n$, calculando también opcionalmente los vectores singulares derecho y/o izquierdo. La operación se escribe de forma general según la siguiente expresión:

$$A(M \times N) = U(M \times M) * \Sigma(M \times N) * V'(N \times N)$$

donde Σ es la matriz $M \times N$ compuesta por ceros excepto por sus $min(M, N)$ elementos diagonales, U es una matriz ortogonal $M \times M$, y V es una matriz ortogonal $N \times N$ (nótese que nos referimos en el cálculo a la matriz traspuesta). Los elementos diagonales de Σ son los valores singulares de A.

Por otro lado, la función *magma_dgesdd* actúa de forma similar, pero realiza la descomposición utilizando una estrategia "divide y vencerás". Esta estrategia incluye algunos supuestos moderados sobre la aritmética de punto flotante. Funcionará correctamente en máquinas con un dígito de guarda en las operaciones de suma/resta, o en aquellas máquinas binarias que no teniendo dígito de guarda realicen la resta como las Cray X-MP, Cray Y-MP, Cray C-90, o Cray-2. Podrá fallar en aquellas máquinas hexadecimales o decimales sin dígitos de guarda, pero los autores de la librería no tienen constancia de ninguna.

La implementación ha hecho uso de otra función integrada en el código MATLAB mediante un MEX-file al igual que con la aproximación mediante CuSolver. En la Fig. 13 se observa el esquema de esta integración a alto nivel. El código MATLAB original ha sido modificado para llamar a una función en C mediante un MEX-file. Dicha integración en C hace uso de la librería MAGMA, que a su vez utiliza otras librerías como las librerías MKL.

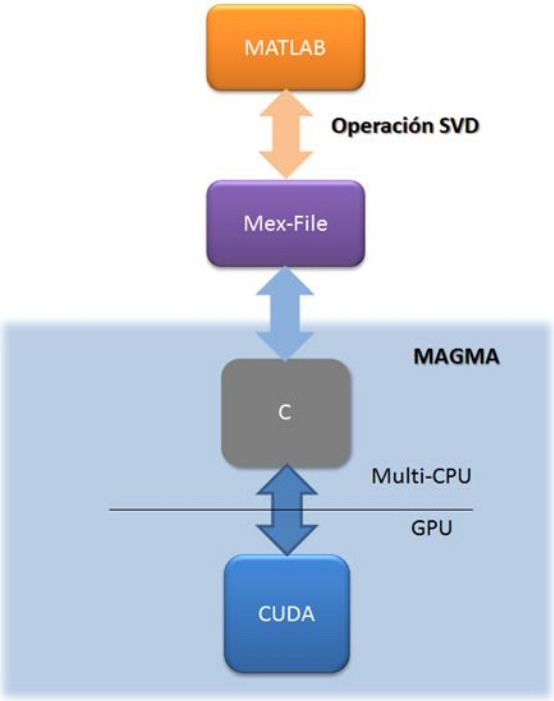


Fig. 13. Jerarquía de entornos para la aproximación mediante MAGMA.

Un factor importante a tener en cuenta cuando se realizan integraciones de este tipo, es que MATLAB también utiliza las librerías MKL de forma interna, y puede generar grandes problemas de incompatibilidades que, a priori, pueden ser muy difíciles de detectar. Para ello, tenemos varias opciones:

- Utilizar la misma versión que utiliza MATLAB de las librerías MKL, e instalando MAGMA sobre esa versión concreta de MKL. Esto permite que haya interoperabilidad entre MATLAB y la librería MAGMA utilizando MEX-files.
- Forzar a que MATLAB utilice la misma versión de MKL sobre la que se ha instalado la librería MAGMA cuando se ejecute dicho MEX-file. Para ello, basta con cambiar una variable del sistema que apunta al origen de la librería MKL. Para ello, basta con ejecutar las siguientes instrucciones y asegurar la ruta correcta para exportar la variable MKLROOT:

```
export MKLROOT=
/ruta/linux/mkl/lib/intel64
```

De igual forma, podemos exportar las rutas de las librerías BLAS y LAPACK con las variables BLAS_VERSION y LAPACK_VERSION, para asegurar también su interoperabilidad con MATLAB.

En la implementación realizada se ha escogido la segunda opción, es decir, se ha forzado a MATLAB a utilizar la última versión de MKL disponible sobre la que se ha instalado MAGMA.

El uso de la librería MAGMA conlleva aprovechar las arquitecturas disponibles de nuestro multicomputador. Actualmente, MAGMA utiliza una o varias GPUs (en función de la rutina) y un número de procesadores que podemos definir mediante la variable de entorno parámetro MKL_NUM_PROCESSOR. Por ejemplo, para establecer 16 procesadores:

```
export MKL_NUM_PROCESSOR= 16
```

También es posible definir en el código el número de GPUs a utilizar, pero las rutinas que hemos empleado, tanto *magma_dgesdd* como *magma_dgesvd* para la operación SVD no permiten actualmente utilizar más de una GPU adicional a los procesadores.

En la implementación realizada se han elaborado dos versiones de este código, una para la rutina *magma_dgesdd* y otra para la rutina *magma_dgesvd*. En el capítulo VII se detallan los experimentos que han hecho que nuestro interés se centre en la rutina *magma_dgesdd*. En la llamada a estas dos rutinas es posible especificar el número de vectores propios a calcular mediante el parámetro *jobz*, que permite los siguientes valores de entrada:

- MagmaAllVec: Las M columnas de U y las N filas de V^T se devuelven en los arrays U y VT.
- MagmaSomeVec: las primeras $\min(M,N)$ columnas de U y las primeras $\min(M,N)$ filas de V^T son devueltas en los arrays U y VT.
- MagmaOverwriteVec: Si $M \geq N$, las primeras n columnas de U son sobreescritas en el array de entrada de la matriz A y todas las filas V^T son devueltas en el array VT; en otro caso, todas las filas de U son devueltas en el array U y las primeras M filas de V^T son sobreescritas en el array A.
- MagmaNoVec: no se calcula ningún vector singular.

Para ambas funciones se ha utilizado el valor "MagmaSomeVec", ya que no todos los vectores singulares son necesarios para el método LRASR como se mencionó anteriormente en la aproximación mediante CuSolver (Capítulo VI sección A), la matriz Σ quedará limitada a las dimensiones $\Sigma(svp \times svp)$, y la operación para calcular S será de la forma:

$$S(M \times N) = V(M \times svp) * \Sigma'(svp \times svp) * U'(svp \times N)$$

Para la implementación realizada, ambas funciones devuelven todos los parámetros a la memoria del host, al sólo tener una parte de los mismos que es transparente para nosotros en la GPU. Una vez llegados a este punto, se ha realizado una variante del código donde se utiliza o no la librería CUBLAS para realizar las operaciones posteriores al SVD. La primera variante es de la forma:

1. Llamada a la función MEX-file y transferencia de la matriz temp al entorno C.
2. Llamada a la rutina *magma_dgesdd* / *magma_dgesvd* para el cálculo de U, Σ y V^T.
3. Cálculo del thresholding de los valores singulares de la matriz Σ.
4. Llamada a la rutina *cublasSetMatrix* para obtener las matrices parciales de U y V^T.
5. Cálculo de S mediante las multiplicaciones realizadas por la rutina *cublasDgemm.*
6. Se devuelve S al entorno MATLAB.
7. El algoritmo LRASR continúa iterando con el parámetro S actualizado.

La segunda variante prescinde de las operaciones con CUBLAS, presentando esta forma:

1. Llamada a la función MEX-file y transferencia de la matriz temp al entorno C.
2. Llamada a la rutina *magma_dgesdd* / *magma_dgesvd* para el cálculo de U, Σ y V^T.
3. Cálculo del thresholding de los valores singulares de la matriz Σ.
4. Se devuelven U, Σ y V^T al entorno MATLAB.
5. El algoritmo LRASR continúa iterando a partir de este punto, actualizando el parámetro S en el entorno MATLAB.

El uso de la segunda variante se justifica para la librería MAGMA en que los valores de U, Σ y V^T se encuentran en la memoria del host, y es necesario volverlos a llevar al device para operar con CUDA, lo cual se traduce en retardos adicionales que pueden reducir la eficiencia de operar en la GPU. Las funciones *magma_dgesdd* y *magma_dgesvd* están implementadas para tomar los valores del host, y devolverlos al host, por lo que no es posible actualmente bajar al nivel del device y mantener ahí los datos para poder reutilizarlos en sucesivas operaciones en el device. Es por eso que la primera variante será más costosa en tiempo que la segunda.

VII. RESULTADOS

Para la evaluación, se ha utilizado un servidor Bullx R424-E3 Intel Xeon E5 2650 (con 16 cores) con una GPU TeslaM2075. El driver de CUDA que se ha utilizado es CUDA 7.5 [52]. Las características de la GPU se muestran en la Tabla IV. Los experimentos han sido

TABLA VI

TABLAS DE TIEMPOS DE EJECUCIÓN DEL EJEMPLO PARA DIFERENTES IMPLEMENTACIONES DE SVD

Tiempo total de ejecución del ejemplo (segundos)									
	Operación SVD	Número de procesos MKL							
		1mkl	2mkl	4mkl	6mkl	8mkl	12mkl	16mkl	Solo GPU
Test 1	magma_svd	188,64	140,75	128,25	121,71	122,60	120,66	126,25	
	magma_sdd	186,94	133,21	124,07	118,99	118,65	118,85	121,15	
	MATLAB	120,15							
	CuSolver								>1 hora
Test 2	magma_svd	238,25	238,61	240,63	239,14	241,36	236,20	239,77	
	magma_sdd	385,12	276,63	240,21	227,19	228,86	232,92	237,62	
	MATLAB	240,30							
	CuSolver								>1 hora
Test 3	magma_svd	2609,71	1612,81	1348,68	1366,44	1348,36	1347,90	1325,49	
	magma_sdd	2615,06	1616,98	1374,67	1314,68	1320,06	1331,12	1352,73	
	MATLAB	1421,44							
	CuSolver								>1 hora

compilados con NVIDIA CUDA y C y el compilador gcc con la opción de optimización -O2.

La versión de MATLAB empleada es la R2012a (7.14.0.739) 64-bit (glnxa64). Aunque para las medidas se ha forzado a MATLAB a utilizar la última versión de MKL disponible, lo cual reduce significativamente los tiempos de cálculo.

La versión que se ha empleado de gcc para compilar el código C es "4.6.3-1ubuntu5". La versión de MAGMA utilizada es la 2.0.

TABLA IV

CARACTERÍSTICAS DE LA GPU (TESLA M2070) UTILIZADA EN LA EVALUACIÓN

Característica	Valor
Peak performance (double precision) (GFlops)	515
Peak performance (simple precision) (GFlops)	1030
Device memory (GB)	5.2
Clock rate (GHz)	1.2
Memory bandwidth (GBytes/sec)	150
Multiprocessors	14
CUDA cores	448
Compute Capability	2
Year architecture	2010
DRAM TYPE	GDDR5

Se han realizado varias medidas para cuantificar la optimización del método y de la operación SVD en concreto. Para las medidas se han realizado ejecuciones del ejemplo descrito en la sección IV, además de algunas variaciones sobre dicho ejemplo para incrementar su tamaño. Los distintos tamaños de imagen hiperespectral utilizados se muestran en la tabla V. La aceleración percibida por el método en las diferentes implementaciones de la operación SVD se representa en la Tabla VI.

TABLA V

TAMAÑOS DE IMAGEN HIPERESPECTRAL UTILIZADOS EN LAS MEDIDAS

	Tamaño	Valor	Nº iteraciones necesarias
Test 1	Pequeño (ejemplo original)	80x100 píxeles	105
Test 2	Mediano	120x130 píxeles	109
Test 3	Grande (imagen completa)	307x307 píxeles	111

Para la implementación en MAGMA, se han elaborado dos variantes de código, donde en la primera variante se realizan operaciones adicionales al SVD, mientras que en la segunda se acelera exclusivamente el SVD. Las medidas presentadas en las tablas corresponden a la segunda variante del código, ya que en ningún caso se consiguió acelerar el problema con la primera variante. Este resultado era previsible, ya que obligar a llevar el dato de nuevo al device para poder realizar esas operaciones posteriores penalizaba cualquier aceleración conseguida.

En la Tabla VI se puede comprobar cómo el ejemplo es acelerado para todos los tamaños respecto a la implementación original en MATLAB para ciertas combinaciones de método SVD y procesos MKL. De esta tabla podemos observar lo siguiente:

- La operación SVD realizada mediante CuSolver en una arquitectura pura de GPU no es funcional en la actualidad, para ninguno de los tamaños.

TABLA VII

TABLAS DE TIEMPOS DE EJECUCIÓN DE LA OPERACIÓN SVD PARA DIFERENTES IMPLEMENTACIONES

	Tiempo medio de ejecución de la operación SVD (segundos)								
	Operación SVD	Número de procesos MKL							
		1mkl	2mkl	4mkl	6mkl	8mkl	12mkl	16mkl	Solo GPU
Test 1	magma_svd	0,7764	0,5005	0,3660	0,3108	0,2938	0,2600	0,2625	
	magma_sdd	0,8036	0,4583	0,3410	0,2881	0,2725	0,2395	0,2440	
	MATLAB	0,2103							
	CuSolver								>1 min
Test 2	magma_svd	1,6372	0,8146	0,5344	0,4505	0,3910	0,3710	0,3374	
	magma_sdd	1,6017	0,7786	0,5008	0,4283	0,3811	0,3465	0,3213	
	MATLAB	0,3551							
	CuSolver								>1 min
Test 3	magma_svd	12,1571	4,3226	2,7444	2,1737	1,8822	1,5942	1,4787	
	magma_sdd	12,0982	4,2527	2,6220	2,1526	1,8932	1,5436	1,4395	
	MATLAB	1,9448							
	CuSolver								>1 min

- En las implementaciones con MAGMA, es posible obtener mejores tiempos que la arquitectura multicore basada en la librería MKL de MATLAB.
- Las implementaciones con MAGMA son muy sensibles al número de procesos MKL definidos, pudiendo incluso empeorar mucho los tiempos respecto a MATLAB si la elección de dicho número de procesos no es óptima.
- De forma general, los mejores tiempos se obtienen con el algoritmo SDD de MAGMA, que utiliza la estrategia "divide y vencerás". Sin embargo, para un número pequeño de procesos MKL, esta estrategia arroja peores tiempos que el algoritmo SVD clásico de MAGMA.
- La aceleración que sufre el ejemplo aumenta con la magnitud del mismo. Esto hace interesante este tipo de aceleración cuando se trabaja con grandes volúmenes de información.

Debido a que este trabajo se enfoca principalmente a la aceleración de la operación SVD dentro de un problema concreto, se han realizado medidas para cuantificar la aceleración que experimentan los resultados con arquitecturas heterogéneas. Para ello, se ha elaborado la Tabla VII, que muestra el tiempo medio de cálculo de la operación SVD, ya que el tiempo de las iteraciones no es exactamente el mismo, siendo las primeras algo más costosas.

Es importante destacar que la aceleración que sufre el problema no se debe exclusivamente a que la operación SVD sea más rápida, sino también a que realiza algunos pasos que MATLAB debe realizar posteriormente. Estos pasos son la trasposición de la matriz de vectores singulares derecha, V, que MAGMA devuelve directamente como resultado, y la extracción de los valores singulares en un único vector, paso que también realiza MAGMA directamente pero que MATLAB debe realizar de forma posterior. Estos retardos adicionales que sufre MATLAB pero no las implementaciones de MAGMA se cuantifican en el orden de 0,0136s para el ejemplo pequeño, 0,0364s para el ejemplo mediano y 0,2238s para el ejemplo grande.

Existe una ligera incoherencia entre las Tablas VI y VII, ya que, a priori, parecería que al tardar menos el SVD con un mayor número de procesos MKL, todo el problema debería verse acelerado, lo cual no ocurre. Esto es debido a que al forzar a MATLAB a realizar operaciones sencillas con un número mayor de procesos MKL, se generan ineficiencias.

Respecto a la implementación con CuSolver, las medidas realizadas sobre la operación de descomposición SVD en la GPU han arrojado tiempos muy altos de cálculo, tal y como puede comprobarse en la Tabla VIII. Se observa que la primera iteración es la más costosa, sin embargo, los valores de tiempos para la operación SVD no se reducen significativamente. La función *CuSolverDnDgesvd* de la librería CuSolver en su estado actual no permite mejorar esos tiempos. Sin embargo, aparecen opciones interesantes que se implementarán a futuro, como la posibilidad de calcular sólo una parte de los vectores U y V, reduciendo el esfuerzo computacional de cálculo que no sería necesario para nuestro problema.

En la Tabla VIII, los tiempos previos a la operación SVD corresponden al volcado de los datos y a la asignación de memoria en la GPU, que observamos que son mucho mayores en la primera iteración.

En la misma Tabla VIII, los tiempos posteriores a la operación SVD corresponden a las operaciones de multiplicación de matrices mediante la librería cuBLAS, y a las transformaciones necesarias para truncar los valores de los vectores singulares U y V y aplicar el threshold a los valores singulares obtenidos (operación umbral de valores singulares, θ). Se observan mayores tiempos en la primera iteración, debidos a la

inicialización de algunas variables adicionales y a las asignaciones de memoria en la GPU.

TABLA VIII

VALORES REALES DE TIEMPOS DE LA RUTINA SVD CUSOLVER. AL FINAL DE LA TABLA APARECEN LOS VALORES MEDIOS DE LAS DIEZ ITERACIONES POSTERIORES A LA PRIMERA.

Valor	Iteración	Tiempo empleado (segundos)
Tiempo previo a SVD	1	1,099
Operación SVD	1	61,232
Tiempo posterior a SVD	1	0,965
Tiempo previo a SVD	2	0,011
Operación SVD	2	59,224
Tiempo posterior a SVD	2	0,047
Tiempo previo a SVD	3	0,09
Operación SVD	3	60,176
Tiempo posterior a SVD	3	0,036
Tiempo previo a SVD	**Media de iteraciones 2-11**	**0,010**
Operación SVD	**Media de iteraciones 2-11**	**60,544**
Tiempo posterior a SVD	**Media de iteraciones 2-11**	**0,041**

Debido a la imposibilidad de reducir significativamente los tiempos de ejecución de la descomposición SVD con la librería CuSolver, que deberían estar en el orden de 1 ó 2 segundos para poder compararse con el resto de implementaciones, se decidió abandonar esta línea de trabajo hasta la optimización de esta librería por parte de los autores.

Por lo tanto, los posteriores esfuerzos deberían de centrarse en la librería MAGMA y las arquitecturas heterogéneas, que han demostrado estar mucho más optimizadas.

VIII. CONCLUSIONES

En este trabajo se ha realizado un desarrollo sobre el método de detección de anomalías sobre imágenes hiperespectrales LRASR. Dicho desarrollo se ha enfocado a la parte más costosa computacionalmente del problema, la operación SVD. Se ha realizado una evaluación de las librerías paralelas existentes para realizar la operación SVD en paralelo centrándonos en los tamaños de interés para las aplicaciones de las imágenes hiperespectrales. A pesar de que paralelizar la operación SVD tradicionalmente no ha arrojado buenos resultados, se ha conseguido acelerar dicha operación a partir de ciertos tamaños del problema, sin empeorar el resultado significativamente para ejemplos pequeños.

La operación SVD es muy importante para diversos problemas de la ciencia y la ingeniería, y su análisis y aceleración sobre arquitecturas heterogéneas puede abrir la posibilidad de trabajar sobre problemas anteriormente inabordables. El desarrollo actual permite plantear unos tamaños de datos a analizar mucho mayores para nuestro método, consiguiente ahorros significativos en los tiempos empleados gracias a las arquitecturas HPC.

Se ha observado que para tamaños pequeños de problema, la implementación que realiza MATLAB sobre la librería MKL es bastante eficiente, sin embargo para tamaños mayores de imágenes los resultados son mucho más prometedores sobre arquitecturas heterogéneas.

Las librerías actuales no permiten una implementación eficiente de la operación SVD exclusivamente sobre GPU, ya que las operaciones de nivel 2 BLAS son muy costosas sobre esta arquitectura. Sin embargo, la librería MAGMA utiliza una GPU para acelerar parte de las operaciones de nivel 3 BLAS, además de varios procesadores que le permiten realizar las operaciones nivel 2 BLAS, que resulta en una implementación óptima en algunos casos.

También concluimos que el número de procesos MKL óptimo para un problema no tiene por qué ser el mismo para todas sus secciones. Una implementación más rápida debería tener en cuenta el coste computacional de cada operación, utilizando un número variable de procesadores y GPU para alcanzar el mejor tiempo de cada subproblema.

IX. TRABAJO FUTURO

Existen diversas líneas de trabajo futuro sobre la aceleración de este método. En primer lugar, las librerías utilizadas aún están en desarrollo y se encuentran en distintas fases de optimización.

Por un lado, la librería CuSolver que actualmente no arroja los rendimientos esperados sigue en desarrollo. Y, por otro, la librería MAGMA está implementando métodos novedosos que permiten resolver la operación SVD utilizando múltiples GPU. En concreto, los autores están integrando el QDWH-SVD Solver [65], que permitirá utilizar múltiples GPUs en la operación SVD, lo cual se espera que mejore el rendimiento para dicha operación.

Además, el problema actualmente consta de diversos elementos susceptibles en alto grado a la paralelización, como la creación del diccionario o el operador de minimización $\ell_{2,1}$, tal como se comenta en la sección de caracterización del problema. El hecho de acelerar otras partes del problema puede llevarnos a conseguir aceleraciones mayores, tanto para el tamaño propuesto en el ejemplo escogido por los autores como para problemas con conjuntos de datos mucho mayores.

AGRADECIMIENTOS

Este trabajo ha sido posible gracias al gran respaldo por parte de mis directoras de proyecto, Ester y Gloria. Valoro enormemente el esfuerzo que han dedicado a enseñarme todo lo posible y a mantener una perspectiva crítica y científica, a pesar de la distancia y de las complicaciones. También quiero agradecer a Antonio Plaza y al resto de autores del método LRASR de la Nanjing University of Science and Technology [70] por proporcionarnos el código original imprescindible para poder elaborar este trabajo. Agradecer finalmente a toda mi familia por el apoyo y la confianza que han depositado en mí, que no dudaron ni por un momento en que este trabajo fuera posible, alentándome a realizar un trabajo meticuloso y a no rendirme en el camino.

REFERENCIAS

[1] AMD. ATI Stream technology. 2016. URL: http://www.amd.com/en-us/innovations/software-technologies/firepro-graphics/stream

[2] A. Bannerjee, P. Burlina, and C. Diehl, "A support vector method for anomaly detection in hyperspectral imagery," *IEEE Trans. Geosci. Remote Sens.*, vol. 44, no. 8, pp. 2282–2291, Aug. 2006.

[3] J.I. Agulleiro, F. Vazquez, E.M. Garzón, J.J. Fernández, "Hybrid Computing: CPU+GPU co-processing and its application to tomographic reconstruction", *Ultramicroscopy*, Vol. 115, pp 109-114, April 2012.

[4] A. M. Bruckstein, D. L. Donoho, and M. Elad, "From sparse solutions of systems of equations to sparse modeling of signals and images," *SIAM Rev.*, vol. 51, no. 1, pp. 34–81, Feb. 2009.

[5] M. Borengasser, W. S. Hungate, and R. Watkins, *Hyperspectral Remote Sensing—Principles and Applications*. Boca Raton, FL, USA: CRC Press, 2008.

[6] D.R. Butenhof. "Programming with POSIX Threads. Professional Computing Series". Addison-Wesley, 1997.

[7] J.F. Cai, E. J. Candès, and Z. Shen, "A singular value thresholding algorithm for matrix completion," *SIAM J. Optim.*, vol. 20, no. 4, pp. 1956–1982, 2010.

[8] E. J. Candès, X. Li, Y. Ma, and J. Wright, "Robust principal component analysis?" *J. ACM*, vol. 58, no. 3, pp. 1–39, May 2011.

[9] B. Chamberlain, D. Callahan, and H. Zima. "Parallel programmability and the chapel language". *International Journal of High Performance Computing Applications*, 21(3):291, August 2007.

[10] C.-I. Chang, H. Ren, and S.-S. Chiang, "Real-time processing algorithms for target detection and classification in hyperspectral imagery", *IEEE Transactions on Geoscience and Remote Sensing*, vol. 39, no. 4, pp. 760–768, 2001.

[11] C.-I. Chang, "Hyperspectral Imaging: Techniques for Spectral Detection and Classification". Norwell, MA: Kluwer, 2003.

[12] B. Chapman, G. Jost, and R. van der Pas. "Using OpenMP. Portable Shared Memory Parallel Programming". The Mit Press, 2007.

[13] Chen, S.; Yang, S.; Kalpakis, K.; Chang, C.I. "Low-rank decomposition-based anomaly detection". *Proc. SPIE* 2013, 8743, 1–7.

[14] Y. Chen, N. M. Nasrabadi, and T. D. Tran, "Simultaneous joint sparsity model for target detection in hyperspectral imagery," *IEEE Geosci. Remote Sens. Lett.*, vol. 8, no. 4, pp. 676–680, Jul. 2011.

[15] Y. Chen, N. M. Nasrabadi, and T. D. Tran, "Hyperspectral image classification via kernel sparse representation," in *Proc. IEEE Int. Conf. Image Process.*, Brussels, Belgium, Sep. 2011, pp. 1233–1236.

[16] R. J. Chevance. Server "Architectures: Multiprocessors, Clusters, Parallel Systems, Web Servers, Storage Solutions". Elsevier, 2005.

[17] Per Christian Hansen, "Rank-Deficient and Discrete Ill-Posed Problems: Numerical Aspects of Linear Inversion", *Mathematical Modeling and Computation*, Society for Industrial and Applied Mathematics, Philadelphia, 1998.

[18] J. Demmel and W. Kahan, "Computing Small Singular Values of Bidiagonal Matrices With Guaranteed High Relative Accuracy", *SIAM Journal on Scientific and Statistical Computing*, 5 (1990), pp. 873–912.

[19] A.J. van Der Steen. "Overview of Recent Supercomputers". Technical report, EuroBen Foundation, SURFsara, The Netherlands, 2013.

[20] B. Du and L. Zhang, "Random-selection-based anomaly detector for hyperspectral imagery," *IEEE Trans. Geosci. Remote Sens.*, vol. 49, no. 5, pp. 1578–1589, May 2011.

[21] B. Du and L. Zhang, "A discriminative metric learning based anomaly detection method," *IEEE Trans. Geosci. Remote Sens.*, vol. 52, no. 11, pp. 6844–6857, Nov. 2014.

[22] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. "UPC: Distributed Shared Memory Programming". Edt. Willey, 2005.

[23] M. D. Farrell and R. M. Mersereau, "On the impact of PCA dimension reduction for hyperspectral detection of difficult targets," *IEEE Geosci. Remote Sens. Lett.*, vol. 2, no. 2, pp. 192–195, Apr. 2005.

[24] M. Fazel, "Matrix rank minimization with applications," Ph.D. dissertation, Dept. Elect. Eng., Stanford Univ., Stanford, CA, USA, 2002.

[25] S. Gochman, A. Mendelson, A. Naveh, and E. Rotem. "Introduction to intel core duo processor architecture". *Intel Technology Journal*, 10(2):89-97, 2006.

[26] H. Goldberg, H. Kwon, and N. M. Nasrabadi, "Kernel eigenspace separation transform for subspace anomaly detection in hyperspectral imagery," *IEEE Geosci. Remote Sens. Lett.*, vol. 4, no. 4, pp. 581–585, Oct. 2007.

[27] G.H. Golub and C. Reinsch, "Singular Value Decomposition and Least Squares Solutions", *Numerische Mathematik*, 14 (1970), pp. 403–420.

[28] G.H. Golub and Charles F. Van Loan, "Matrix Computations", *John Hopkins Studies in the Mathematical Sciences*, Johns Hopkins University Press, Baltimore, Maryland, 3 ed., 1996.

[29] M. Gu and S.C. Eisenstat, "A Divide-and-Conquer Algorithm for the Bidiagonal SVD", *SIAM Journal on Matrix Analysis and Applications*, 16 (1995), pp. 79–92.

[30] Q. Guo *et al.*, "Weighted-RXD and Linear Filter-Based RXD: Improving background statistics estimation for anomaly detection in hyperspectral imagery," *IEEE J. Sel. Topics Appl. Earth Obs. Remote Sens.*, vol. 7, no. 6, pp. 2351–2366, Jun. 2014.

[31] J. Hennessy and D. Patterson, "Computer Architecture: A Quantitative Approach". Morgan Kaufmann publications, 2006.

[32] Innovative Computing Laboratory, Universidad de Tennessee USA, MAGMA library (Matrix Algebra on GPU and Multicore Architectures), 2016. URL: http://icl.cs.utk.edu/magma.

[33] Intel(R) Threading Building Blocks. Reference Manual. Document Number 315415-004US, 2016. URL: https://software.intel.com/en-us/intel-tbb

[34] Intel math kernel library (developer Guide), 2016. URL: https://software.intel.com/en-us/mkl-for-linux-userguide

[35] J. Kerekes, "Receiver operating characteristic curve confidence intervals and regions". *IEEE Geosci. Remote Sens.* Lett. 2008, 5, 251–255.

[36] H. Kwon and N. M. Nasrabadi, "Kernel RX-algorithm: A nonlinear anomaly detector for hyperspectral imagery," *IEEE Trans. Geosci. Remote Sens.*, vol. 43, no. 2, pp. 388–397, Feb. 2005.

[37] C. Lee, S. Gasster, A. Plaza, C.-I. Chang, and B. Huang, "Recent developments in high performance computing for remote sensing: A review", *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 4, no. 3, pp. 508–527, 2011.

[38] W. Li and Q. Du, "Collaborative representation for hyperspectral anomaly detection," *IEEE Trans. Geosci. Remote Sens.*, vol. 53, no. 3, pp. 1463–1474, Mar. 2015.

[39] Z. Lin, M. Chen, and Y. Ma, "The augmented Lagrange multiplier method for exact recovery of corrupted low-rank matrices,– Univ. Illinois Urbana–Champaign, Champaign, IL. USA, UIUC Tech. Rep. UILU-ENG-09-2215, 2009.

[40] Z. Lin, R. Liu, and Z. Su, "Linearized alternating direction method with adaptive penalty for low rank representation," *Adv. Neural Inf. Process. Syst.*, pp. 612–620, 2011.

[41] G. Liu, Z. Lin, and Y. Yu, "Robust subspace segmentation by low-rank representation," in *Proc. Int. Conf. Mach. Learn.*, 2010, pp. 663–670.

[42] G. Liu *et al.*, "Robust recovery of subspace structures by low-rank representation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 1, pp. 171–184, Jan. 2013.

[43] W. Liu and C. I. Chang, "Multiple-window anomaly detection for hyperspectral imagery," *IEEE J. Sel. Topics Appl. Earth Obs. Remote Sens.*, vol. 6, no. 2, pp. 644–658, Apr. 2013.

[44] D. Manolakis and G. Shaw, "Detection algorithms for hyperspectral imaging applications," *IEEE Signal Process. Mag.*, vol. 19, no. 1, pp. 29–43, Jan. 2002.

[45] D. Manolakis, D. Marden, and G. A. Shaw, "Hyperspectral image processing for automatic target detection application"s, *MIT Lincoln Laboratory Journal*, vol. 14, pp. 79–116, 2003.

[46] G. Martin Hernández, "Diseño e implementación eficiente de nuevos métodos de preprocesado espacial para desmezclado de imágenes hiperespectrales de la superficie terrestre", Tesis doctoral, Universidad de Extremadura, 2013.

[47] Mathworks. MATLAB. 2016. URL: http://uk.mathworks.com/products/matlab/

[48] S. Matteoli, M. Diani, and G. Corsini, "A tutorial overview of anomaly detection in hyperspectral images", *IEEE Aerospace and Electronic Systems Magazine*, vol. 25, no. 7, pp. 5–28, Jul. 2010

[49] S. Matteoli, M. Diani, and G. Corsini, "Improved estimation of local background covariance matrix for anomaly detection in hyperspectral images," *Opt. Eng.*, vol. 49, no. 4, pp. 1–16, 2010.

[50] N. M. Nasrabadi, "Regularization for spectral matched filter and RX anomaly detector," in *Proc. SPIE*, vol. 6966, 2008, pp. 1–12.

[51] Y. Niu and B. Wang, "Hyperspectral Anomaly Detection Based on Low-Rank Representation and Learned Dictionary", *Remote Sens.* 2016, 8, 289.

[52] NVIDIA Corporation: "CUDA C PROGRAMMING GUIDE PG-02829-001 v7.5 (2015)"

[53] Nvidia Fermi architecture whitepaper. URL: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architcture_Whitepaper.pdf

[54] OpenMPI. 2016. URL: https://www.open-mpi.org/

[55] G. Ortega, E. M. Garzón, F. Vázquez and I. García, "The Biconjugate gradient method on GPUs", *J Supercomput.* 64:49–58, 2013.

[56] S. Patel and W. W. Hwu. "Accelerator architectures". *IEEE Micro*, 28(4):4–12, 2008.

[57] D. Patterson, J. Hennessy, and E. M. Kaufmann. "Computer Organization and Design". 2014.

[58] A. Plaza and C.-I. Chang, "High performance computing in remote sensing". Boca Raton: CRC Press, 2007.

[59] A. Plaza, J. Benediktsson, J. Boardman, J. Brazile, L. Bruzzone, G. Camps-Valls, J. Chanussot, M. Fauvel, P. Gamba, J. Gualtieri, M. Marconcini, J. C. Tilton, and G. Trianni, "Recent advances in techniques for hyperspectral image processing", *Remote Sensing of Environment*, vol. 113, pp. 110–122, 2009.

[60] V. Saraswat. Report on the Programming Language X10 Version 2.0.

[61] M.J. Quinn. "Parallel Programming in C with MPI and OpenMP". McGraw-Hill Education Group, 2003.

[62] K. I. Ranney and M. Soumekh, "Hyperspectral anomaly detection within the signal subspace," *IEEE Geosci. Remote Sens. Lett.*, vol. 3, no. 3, pp. 312–316, Jul. 2006.

[63] I. S. Reed and X. Yu, "Adaptive multiple-band CFAR detection of an optical pattern with unknown spectral distribution," *IEEE Trans. Acoust. Speech Signal Process.*, vol. 38, no. 10, pp. 1760–1770, Oct. 1990.

[64] W. Sakla, A. Chan, J. Ji, and A. Sakla, "An SVDD-based algorithm for target detection in hyperspectral imagery," *IEEE Geosci. Remote Sens. Lett.*, vol. 8, no. 2, pp. 384–388, Mar. 2011.

[65] D. Sukkari, H. Ltaief, D.Keyes, "A High Performance QDWH-SVD Solver Using Hardware Accelerators," *ACM Transactions on Mathematical Software (TOMS)*, vol. 43, issue 1, nº6, Aug. 2016.

[66] W. Sun, C. Liu, and J. Li, "Low-rank and sparse matrix decomposition based anomaly detection for

hyperspectral imagery," *J. Appl. Remote Sens.*, vol. 8, no. 1, May 2014, Art. ID 083641.

[67] Y. Tarabalka, T. V. Haavardsholm, I. Kasen, and T. Skauli, "Real-time anomaly detection in hyperspectral images using multivariate normal mixturemodels and gpu processing*", J. Real-Time Image Process*, vol. 4, pp. 1–14, 2009.

[68] K Vince Fernando and Beresford N Parlett, "Accurate Singular Values and Differential QD Algorithms", *Numerische Mathematik*, 67 (1994), pp. 191–229.

[69] J. Wright *et al.*, "Sparse representation for computer vision and pattern recognition," *Proc. IEEE*, vol. 98, no. 6, pp. 1031–1044, Jun. 2010.

[70] Y. Xu, Z. Wu, J. Li, A. Plaza and Z. Wei, "Anomaly detection in hyperspectral images based on low-rank and sparse representation," *IEEE Trans. Geosci. Remote Sens.*, vol. 54, no. 4, pp. 1990–2000, Apr. 2016.

UNIVERSIDAD DE ALMERÍA

Resumen/Abstract

La detección de anomalías es un campo de gran importancia para el procesado de imágenes hiperespectrales. Este trabajo presenta varias aproximaciones para acelerar el método LRASR de detección de anomalías en imágenes hiperespectrales (HSI), con foco en aquellas secciones del método que consumen la mayor parte del tiempo de procesamiento (principalmente la operación de descomposición en valores singulares, SVD). La aceleración de la operación SVD se ha llevado a cabo utilizando diversas librerías que permiten explotar tanto plataformas GPU como heterogéneas (CPU-GPU). Se han llevado a cabo diversos experimentos sobre imágenes test para observar el impacto del uso de dichas librerías sobre el SVD. Dichos experimentos han demostrado que las librerías heterogéneas para computar el SVD presentan factores de aceleración interesantes cuando estamos trabajando con los mayores tamaños de las imágenes test.

Anomaly detection is very important in hyperspectral images (HSI) processing. This work shows several strategies to accelerate the LRASR anomaly detection method for hyperspectral images, with focus on the most time-expensive sections of the algorithm (mainly the singular value decomposition operation, SVD). To achieve this acceleration, many libraries have been used to exploit the power of GPU and CPU-GPU heterogeneous architectures. Different experiments have been studied to observe the impact of the libraries over the test images and the SVD operation in their processing. These experiments have shown that heterogeneous libraries for computing the SVD present notable accelerations when working with the largest test images.