UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Metodología de alto nivel para implementar el algoritmo *Vertex Component Analysis* en una GPU

Autor: **Ricardo Topham González**
Tutor(es): **Gustavo Marrero Callicó**
Sebastián López Suárez
Fecha: **26 -julio - 2011**

IUMA
t +34 928 451 086 | iuma@iuma.ulpgc.es | Campus Universitario de Tafira
f +34 928 451 083 | www.iuma.ulpgc.es | 35017 Las Palmas de Gran Canaria

Versión: 2.0 – 14 jun 2011

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Metodología de alto nivel para implementar el algoritmo *Vertex Component Analysis* en una GPU

HOJA DE FIRMAS

Alumno/a: Ricardo Topham González Fdo.:

Tutor/a: Gustavo Marrero Callicó Fdo.:

Tutor/a: Sebastián López Suárez Fdo.:

Fecha: 26 -julio - 2011

IUMA
t +34 928 451 086 | iuma@iuma.ulpgc.es | Campus Universitario de Tafira
f +34 928 451 083 | www.iuma.ulpgc.es | 35017 Las Palmas de Gran Canaria

Versión: 2.0 – 14 jun 2011

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Metodología de alto nivel para implementar el algoritmo *Vertex Component Analysis* en una GPU

HOJA DE EVALUACIÓN

Calificación: .....

Presidente: Aurelio Vega Martínez Fdo.:

Secretario: Gustavo Marrero Callicó Fdo.:

Vocal: Benito González Pérez Fdo.:

Fecha: 26 -julio - 2011

IUMA
t +34 928 451 086 | iuma@iuma.ulpgc.es | Campus Universitario de Tafira
f +34 928 451 083 | www.iuma.ulpgc.es | 35017 Las Palmas de Gran Canaria

Versión: 2.0 – 14 jun 2011

Índice de contenido

Resumen

En estas primeras líneas de la memoria se presenta un resumen del Trabajo de Fin de Máster realizado.

El objetivo principal de este Trabajo de Fin de Máster es el desarrollo de una metodología de alto nivel para implementar el algoritmo *Vertex Component Analysis* en una GPU. La metodología desarrollada permitirá programar y ejecutar el algoritmo en una GPU aprovechando el potencial que ofrece su arquitectura para la computación en paralelo. Para ello se empleará una GPU NVIDIA GeForce GTX 480 y el *Parallel Computing Toolbox* de MATLAB.

La metodología hará uso de varias funciones disponibles en el *Parallel Computing Toolbox* de MATLAB creadas expresamente para la programación en una GPU, y también de la interacción que ofrece con NVIDIA CUDA. Además, dicha metodología podrá ser extrapolada a otros algoritmos de análisis hiperespectral.

A pesar de que hace unos años no era posible programar en GPUs desde MATLAB, gracias a las nuevas funciones disponibles en el *Parallel Computing Toolbox* sí se puede, ofreciendo una serie de funcionalidades bastante interesantes. De esta forma se realizará un enfoque novedoso que tratará de sacar el máximo partido posible a estos dispositivos desde el entorno MATLAB.

Por tanto, con este Trabajo Fin de Máster se pretende desarrollar una metodología de alto nivel para poder programar el algoritmo *Vertex Component Analysis* en una GPU, y aprovechar las bondades que ofrece su arquitectura paralela para las aplicaciones científicas costosas desde el punto de vista computacional.

Abstract

In these first lines of the memory a summary of the Master Thesis carried out is presented.

The main goal of the Master Thesis is to develop a high level methodology for the implementation of the *Vertex Component Analysis* algorithm in a GPU. The methodology will allow to program and execute the algorithm oin a GPU taking advantage of its architecture and its potential for parallel computing. For this purpose a NVIDIA GeForce GTX 480 and MATLAB's *Parallel Computing Toolbox* will be used.

The methodology will use several functions available in MATLAB's *Parallel Computing Toolbox* built for programming a GPU, and the interaction that it offers with NVIDIA CUDA. In addition, this methodology can be extrapolated to other hyperspectral analysis algorithms.

Although a few years ago it was not possible to program GPUs making use of MATLAB, thanks to new features available in the *Parallel Computing Toolbox* it is now possible, offering a series of very interesting functionalities. Thus this will be a novel approach that will try to take advantage of these devices using MATLAB.

Therefore, this Master Thesis aims to develop a high level methodology to program the *Vertex Component Analysis* algorithm on a GPU, and take advantage of the benefits offered by its parallel architecture for scientific applications that are costly from the computational point of view.

Capítulo 1. Introducción

En este capítulo se describen los objetivos básicos a cumplir en este Trabajo Fin de Máster. Se empieza por los antecedentes, se pasa luego a los objetivos y finalmente se define la estructura seguida en la memoria.

1.1. Motivaciones

El presente trabajo se ha desarrollado dentro de las líneas de investigación actuales de la División de Sistemas Integrados, DSI, del Instituto Universitario de Microelectrónica Aplicada, IUMA, de la Universidad de Las Palmas de Gran Canaria. En el mismo se tratará de desarrollar una metodología de alto nivel para la implementación del algoritmo *Vertex Component Analysis*, también conocido como VCA, en una GPU.

Las potenciales aplicaciones potenciales del algoritmo objeto de estudio son múltiples, destacando aplicaciones de detección de minerales, aplicaciones militares tales como detección de material armamentístico camuflado, anomalías, identificación de agentes contaminantes en aguas y atmósfera, etc.

En el presente TFM se propone una metodología para implementar el VCA en una GPU, y, de esta forma, tratar de hacer frente a las limitaciones de otro tipo de soluciones. Conviene destacar que, con una sola GPU, pueden llegar a obtenerse mejoras notables a la hora de procesar cálculos de tipo científico, como es el caso de los algoritmos de tratamiento de imágenes hiperespectrales, a un coste razonable, partiendo desde unos 120€ en las GPU con las configuraciones más básicas, y además ocupando un espacio mínimo. No obstante, no todas las tarjetas GPU disponibles en el mercado se ajustan a nuestros requerimientos. Finalmente, indicar que en este trabajo se ha intentado ir un poco más allá, y para trabajar de forma totalmente innovadora se ha utilizado la arquitectura CUDA incorporada en las tarjetas gráficas de NVIDIA de la serie GTX, Quadro y Tesla, siendo la GeForce GTX 480 la que ha sido objeto de estudio en este TFM.

Para la realización del mismo se ha empleado un ordenador personal con un procesador Intel Core i5 650 de 3,20GHz, una GPU NVIDIA GeForce GTX 480 y el sistema operativo Windows 7.

1.2. Objetivos

El artículo [1] propone una implementación software del algoritmo *Vertex Component Analysis* que será utilizada como referencia para el cumplimiento del objetivo principal de este Trabajo Fin de Máster, que es desarrollar una metodología de alto nivel para implementar dicho algoritmo. Con múltiples núcleos y con un gran ancho de banda de memoria, hoy por hoy las GPUs ofrecen prestaciones muy elevadas para procesamiento gráfico y científico [2 – 6].

Para desarrollar dicha metodología de alto nivel se empleará el entorno de programación MATLAB, CUDA, y una GPU Geforce GTX 480. Se hará uso del *Parallel Computing Toolbox* [7]. Éste está disponible desde el año 2004, y desde entonces se han ido incorporando numerosas funciones y funcionalidades que hacen que éste sea muy interesante para intentar sacar el máximo partido posible a los nuevos procesadores multinúcleo y multihilo. Desde la versión de MATLAB r2009b, se da soporte nativo a las GPUs, pudiendo hacer uso de CUDA en el propio entorno de programación [8 – 9].

En la página web de uno de los autores del artículo [1], se halla el algoritmo del *Vertex Component Analysis* en código MATLAB [10], que será la base de este Trabajo de Fin de Máster y sobre el que se trabajará para conseguir el objetivo principal propuesto. Para lograr este objetivo principal, se pretende desarrollar los siguientes objetivos específicos:

1. Identificar los segmentos de código del algoritmo *Vertex Component Analysis* susceptibles de implementarse en la GPU.
2. Familiarizarse con el *Parallel Computing Toolbox* y las posibilidades que ofrece de programación en la GPU.
3. Extraer las correspondientes conclusiones durante el desarrollo del trabajo, e identificar posibles mejoras y líneas de trabajo futuro, relacionados con el TFM llevado a cabo.

1.3. Estructura de la memoria

En este apartado se pasará a definir un estudio detallado de las tareas a realizar durante el desarrollo del TFM. La memoria estará formada por 6 capítulos, con el siguiente contenido:

- Capítulo 1. "*Introducción*": En este capítulo se ha expuesto el contexto sobre el que se desarrollará el trabajo, las motivaciones, los objetivos y se presenta la estructura de la memoria.

- Capítulo 2. "*Resumen del estado del arte*": Se realiza un resumen del estado del arte para situar en contexto el trabajo de fin de máster desarrollado.

- Capítulo 3. "*Imágenes hiperespectrales*": Se realiza una descripción sobre el concepto de imágenes hiperespectrales, se explican los pasos en la adquisición de una imagen hiperespectral; la captura de la imagen con el sensor y las distintas conversiones que se llevan a cabo en la misma, así como otros detalles de suma relevancia para el tratamiento de estas imágenes. Finalmente se realizará una descripción del algoritmo objeto de este TFM, el *Vertex Component Analysis*.

- Capítulo 4. "*Tarjetas gráficas programables GPUs*": Se realiza una breve descripción sobre las tarjetas gráficas programables GPUs y el entorno de programación empleado para programar en ellas.

- Capítulo 5. "*Metodología*": Se desarrolla la metodología de alto nivel para implementar el algoritmo en la GPU de NVIDIA.

- Capítulo 6. "*Resultados*": Se describen los resultados obtenidos con la metodología de alto nivel desarrollada.

- Capítulo 7. "*Conclusiones y líneas futuras*": Se revisan los objetivos iniciales y se exponen las conclusiones alcanzadas tras la realización del Trabajo Fin de Máster, así como algunas de las dificultades más relevantes. También se proponen las posibles líneas futuras de trabajo.

Capítulo 2. Resumen del estado del arte

En este capítulo se realiza un resumen del estado del arte para situar en contexto el Trabajo Fin de Máster desarrollado.

Los algoritmos de procesamiento de imágenes hiperespectrales se encuentran en continuo desarrollo y evolución. El tratamiento de las mismas implica una cadena de procesos compleja, entre los que destacan el preprocesado, la extracción de *endmembers*, la segmentación o la clasificación. Uno de los algoritmos existentes para la tarea de extracción de *endmembers* es el *Vertex Component Analysis* [1]. Este último ofrece unos resultados bastante buenos en lo referente a bondad de la extracción de *endmembers* y complejidad de la solución, aunque no es el único algoritmo o método para tal fin.

Desde finales de los años 80 los autores empezaron a explorar las posibilidades para tratar de reducir la problemática asociada al procesamiento de las imágenes hiperespectrales. Uno de los trabajos más interesantes de aquellos años proponía una técnica que conseguía reducir la dimensionalidad de los datos y detectar la presencia de la firma espectral de interés [11]. Por aquel entonces el análisis de imágenes hiperespectrales estaba en sus inicios, pero al mismo tiempo se empezaba a vislumbrar un mundo de posibilidades para las aplicaciones científicas.

La evolución en el campo hiperespectral es imparable; tanto que una década más tarde varios métodos y algoritmos para realizar la extracción de *endmembers* han hecho su aparición. En 2004, Plaza [12] realiza la comparación entre varios de ellos: *Manual Endmember Extraction Tool* (MEST) [13], *Pixel Purity Index* (PPI) [14], *N–FINDR* [15], *Iterative Error Analysis* (IEA) [16], *Optical Real–Time Adaptative Spectral Identification System* (ORASIS) [17], *Convex Cone Analysis* (CCA) [18], *Automated Morphological Endmember Extraction* (AMEE) [19] y *Simulated Annealing Algorithm* (SAA) [20]. El *Vertex Analysis Component* (VCA) [1] es uno más, habiendo hecho su aparición en 2005. Cada uno de ellos tiene sus ventajas y sus desventajas.

Uno de los principales problemas que presentan los algoritmos de análisis hiperespectral es que generalmente conllevan un gran coste computacional, lo que se traduce en una gran cantidad de tiempo necesario para proporcionar resultados (en el caso del algoritmo VCA se deben ejecutar una serie de iteraciones). A ello también contribuye el gran tamaño de las imágenes a procesar, que lleva asociada una transferencia de datos entre

el procesador y la memoria [21]. Debido a ello interesa optimizar el dispositivo hardware especializado utilizado como coprocesador. Al requerir una serie de iteraciones, y tener dentro de éstas operaciones no triviales, los algoritmos de análisis hiperespectral generalmente suelen implicar un consumo de CPU elevado [22 – 23]. A la vista de todo lo anterior, resulta recomendable la utilización de coprocesadores con arquitecturas especializadas.

Las técnicas tradicionales en la literatura para abordar este problema se han decantado por soluciones basadas en el uso de clústeres, sistemas multiprocesador e incluso FPGAs, como pone de manifiesto [24]. El problema de todos ellos tiene que ver con el coste, consumo, peso y/o tiempos de implementación. No obstante, en los últimos años han empezado a adoptarse soluciones basadas en el empleo de GPUs [25 – 27], una alternativa bastante interesante para el tratamiento de datos hiperespectrales.

Unos de los entornos más empleados a la hora de implementar todo tipo de algoritmos en las GPUs es CUDA [28 – 33], desarrollado por NVIDIA. Recientemente MATLAB a través de su *Parallel Computing Toolbox* permite interactuar con CUDA [8 – 9], facilitando notablemente la programación de las GPUs [34], aunque con una serie de restricciones y limitaciones.

La propuesta de este Trabajo Fin de Máster pasa por desarrollar una metodología de alto nivel para implementar el algoritmo *Vertex Component Analysis* en una GPU mediante el uso de MATLAB y CUDA.

Capítulo 3. Imágenes hiperespectrales

En este capítulo se realiza una descripción sobre el concepto de imágenes hiperespectrales, se explican los pasos en la adquisición de una imagen hiperespectral; la captura de la imagen con el sensor y las distintas conversiones que se llevan a cabo en la misma, así como otros detalles de suma relevancia para el tratamiento de estas imágenes. Finalmente se realizará una descripción del algoritmo objeto de este TFM, el *Vertex Component Analysis*

3.1. Concepto de imagen hiperespectral

Las imágenes hiperespectrales son parte de una clase de técnicas comúnmente conocidas como la proyección de imágenes espectrales o análisis espectral. Las imágenes hiperespectrales guardan mucha relación con las imágenes multiespectrales; la diferencia entre ellas se debe al tipo de medida que se realiza [35].

En el campo multiespectral se toman varias imágenes en bandas discretas y relativamente estrechas. Un sensor de este tipo puede tener varias bandas que cubren el espectro de lo visible hasta el infrarrojo de onda larga (LWIR, de *Long Wave Infra Red*). Conviene indicar que las imágenes multiespectrales no reproducen el espectro de un objeto. El satélite Landsat y su sensor TM/ETM/ETM+ son un claro ejemplo de instrumento multiespectral.

Las imágenes hiperespectrales, por su parte, disponen de varias bandas espectrales estrechas y casi contiguas entre sí. De esta forma se reproducen los espectros de todos los píxeles en la escena. Con esta definición, un sensor con sólo 20 bandas también se puede considerar como hiperespectral cuando se cubre el rango de 500–700 nm con 20 bandas con un ancho de 10 nm; en cambio, un sensor con 20 bandas discretas que cubre el visible, el infrarrojo cercano, el infrarrojo de onda corta, el infrarrojo medio, y el infrarrojo de onda larga, sería considerado multiespectral.

El auge de la tecnología hiperespectral en aplicaciones de observación de la Tierra ha permitido que se desarrollen instrumentos de muy elevada resolución tanto espacial como espectral. Los sensores hiperespectrales adquieren imágenes digitales en una gran cantidad de bandas espectrales muy cercanas entre sí, obteniendo, para cada porción de la escena o píxel, una firma espectral característica de cada material. El resultado de la toma de datos por parte de un sensor hiperespectral sobre una determinada escena puede ser representado en forma de cubo de datos, con dos dimensiones para representar la ubicación espacial de un píxel, y una tercera dimensión que representa la singularidad espectral de cada píxel en diferentes longitudes de onda [36].

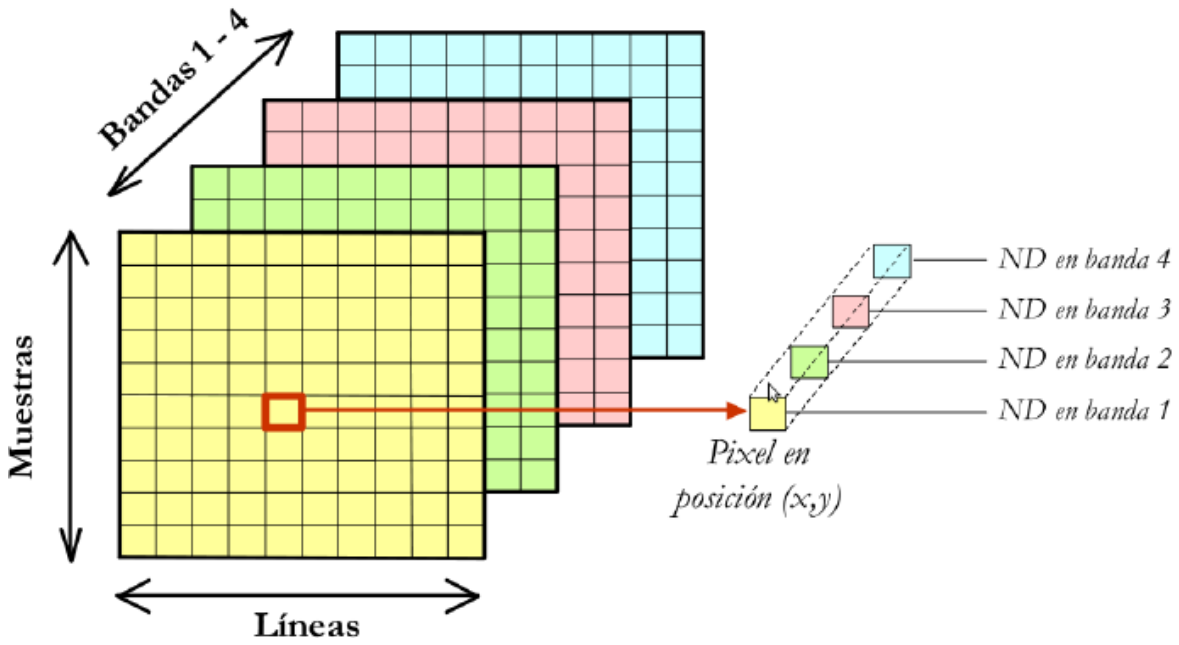

Figura 3.1. Concepto de imagen hiperespectral.

En la figura 3.1. se representa el resultado de la toma de datos por parte de un sensor hiperespectral de una determinada escena. Este resultado puede ser representado en forma de cubo de datos, con dos dimensiones para definir la ubicación espacial de un píxel, y una tercera dimensión para la singularidad espectral de cada píxel en diferentes longitudes de onda. La capacidad de observación de los sensores hiperespectrales permite la obtención de una firma espectral detallada para cada píxel de la imagen, dada por los valores de reflectancia adquiridos por el sensor en diferentes longitudes de onda. Esto último permite una caracterización muy precisa de los elementos de la escena. En la figura 3.2. se muestra el procedimiento de análisis hiperespectral mediante un sencillo diagrama, en el que se ha considerado como ejemplo el sensor AVIRIS (*Airborne Visible Infra–Red Imaging Spectrometer*), desarrollado por el *Jet Propulsión Laboratory* de la NASA. Este sensor cubre el rango de longitudes de onda entre 0,4 y 2,5 nm empleando 224 canales y una resolución espectral de alrededor de 10 nm.

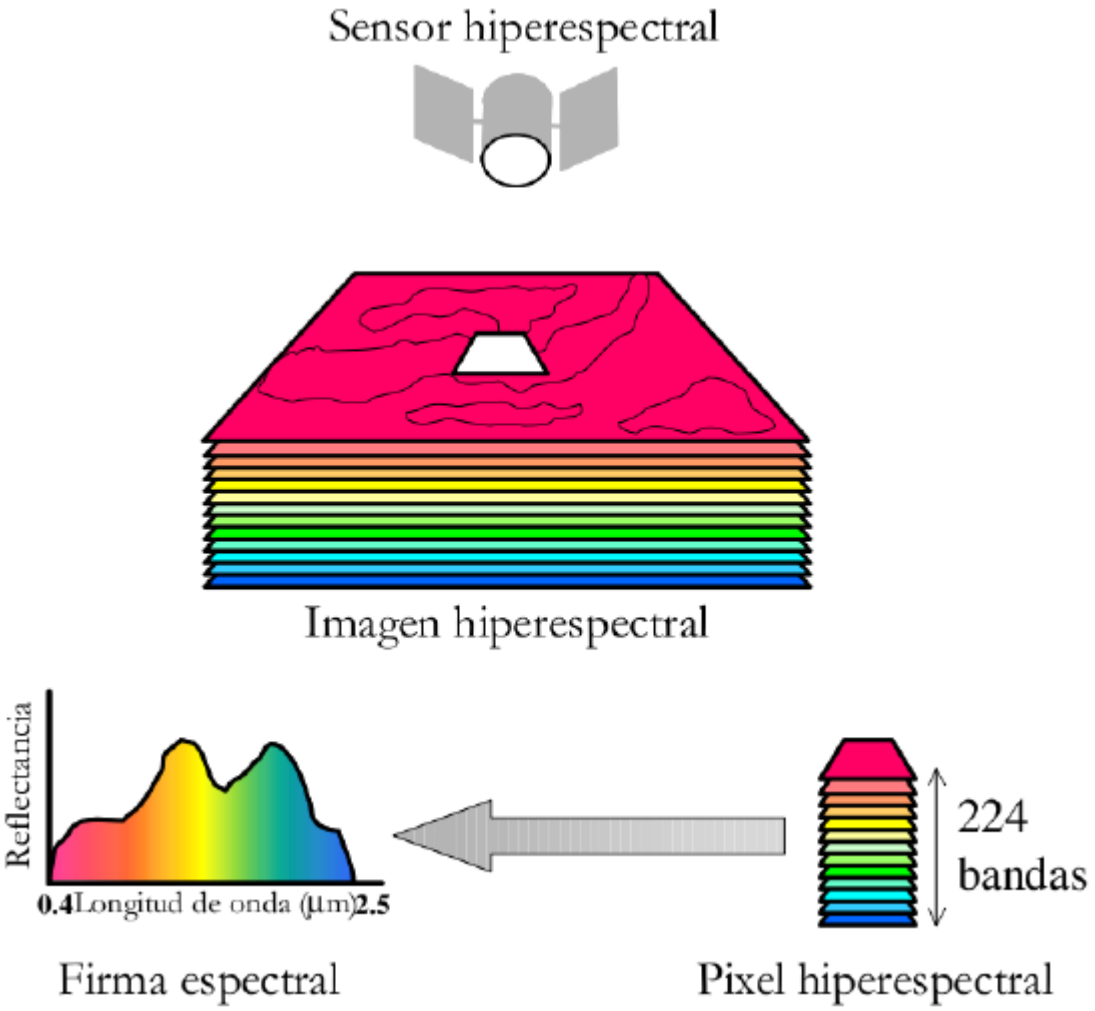

Figura 3.2. Adquisición de imágenes hiperespectrales por el sensor AVIRIS.

Como se observa en la figura 3.2, la capacidad de observación de este sensor permite la obtención de una firma espectral detallada para cada píxel de la imagen. No hay que pasar por alto que en este tipo de imágenes es habitual la existencia de mezclas a nivel de subpíxel, por lo que a grandes rasgos podemos encontrar dos tipos de píxeles en estas imágenes: píxeles puros y píxeles mezcla [37]. Se puede definir un píxel mezcla como aquel en el que cohabitan diferentes materiales. Este tipo de píxeles son los que constituyen la mayor parte de la imagen hiperespectral, en parte, debido a que este fenómeno es independiente de la escala considerada ya que tiene lugar incluso a niveles microscópicos [19]. La figura 3.3. muestra un ejemplo del proceso de adquisición de píxeles puros (a nivel macroscópico) y mezcla en imágenes hiperespectrales.

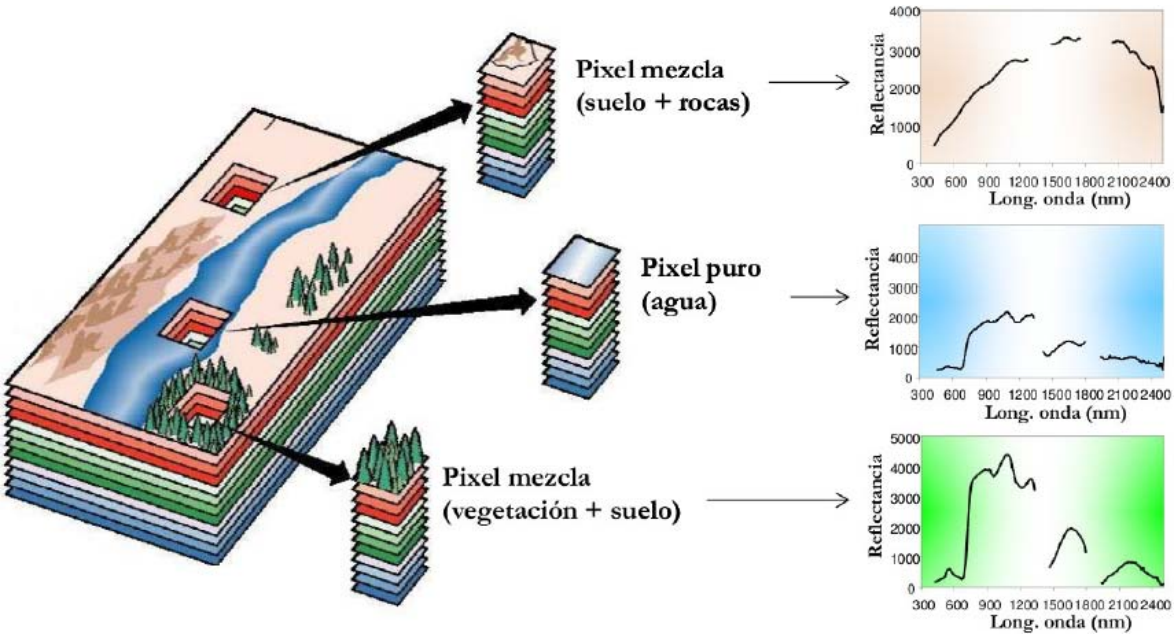

Figura 3.3. Tipos de píxeles en imágenes hiperespectrales.

El desarrollo tecnológico introducido por la incorporación de sensores hiperespectrales en plataformas de observación remota de la tierra de última generación ha sido notable durante los últimos años. En este sentido, existen dos instrumentos en sendos satélites que se encuentran en funcionamiento en la actualidad: Hyperion [38] a bordo del satélite Earth Observing–1 [39] de la NASA, y CHRIS [40] en el satélite PROBA [41] de la ESA, que llevan incorporados sensores de este tipo, permitiendo así la posibilidad de obtener imágenes hiperespectrales de la práctica totalidad del planeta de manera casi continua.

El gran desarrollo en los instrumentos de observación terrestre no ha sido igualada por la evolución en las técnicas de análisis de los datos proporcionados por dichos sensores. El principal problema del análisis hiperespectral es el poco provecho que se obtiene de la gran cantidad de información espacial y espectral presente en las imágenes hiperespectrales. Resolver esta problemática constituye un objetivo de gran interés para la comunidad científica, y de ahí surgen las diferentes técnicas de análisis hiperespectral.

3.2. El sensor hiperespectral AVIRIS

En la actualidad, existe una amplia gama de sensores hiperespectrales de observación terrestre. Dichos sensores pueden clasificarse según su plataforma de transporte en el

momento de la adquisición de datos [42 – 44]. La mayor parte de los sensores hiperespectrales actuales son aerotransportados (siendo el ejemplo más claro de este tipo de instrumentos el sensor AVIRIS [45], considerado en el presente trabajo).

AVIRIS es un sensor hiperespectral aerotransportado con capacidades analíticas en las zonas visibles e infrarrojas del espectro [46 – 48]. Lleva en funcionamiento desde 1987. Fue el primer sistema de adquisición de imágenes capaz de obtener información en una gran cantidad de bandas espectrales estrechas y casi contiguas. AVIRIS es un instrumento único en el mundo de la teledetección, pues permite obtener información espectral en 224 canales espectrales contiguos, cubriendo un rango de longitudes de onda entre 0,4 y 2,5 nm, siendo el ancho entre las bandas muy pequeño, aproximadamente 10 mm.

En 1989, AVIRIS se convirtió en un instrumento aerotransportado. Desde entonces se vienen realizando campañas de vuelo anuales para tomar datos. El sensor ha realizado adquisiciones de datos en Estados Unidos, Canadá y Europa, utilizando para ello dos plataformas:

- Un avión ER–2 perteneciente al *Jet Propulsion Laboratory* de la NASA. El ER–2 puede volar a un máximo de 20 km sobre el nivel del mar, a una velocidad máxima de aproximadamente 730 km/h.
- Un avión denominado Twin Otter, capaz de volar a un máximo de 4 km sobre el nivel del mar, a velocidades de 130 km/h.

Algunas de las características más relevantes en cuanto al diseño interno del sensor AVIRIS son las siguientes:

- El sensor utiliza un explorador de barrido que permite obtener un total de 614 píxeles por cada oscilación.
- La cobertura de la parte visible del espectro es realizada por un espectrómetro EFOS–A, compuesto por un array de 32 detectores lineales.
- La cobertura en el infrarrojo es realizada por los espectrómetros EFOS–B, EFOS–C y EFOS–D, compuestos todos ellos por arrays de 64 detectores lineales.

- La señal medida por cada detector se amplifica y se codifica utilizando 12 bits. Esta señal se almacena en una memoria intermedia donde es sometida a una etapa de pre procesado, siendo registrada a continuación en una cinta de alta densidad de 10,4 GB a velocidad de 20,4 MB/s.
- El sensor dispone de un sistema de calibración a bordo, que utiliza una lámpara halógena de cuarzo que proporciona la radiación de referencia necesaria para comprobar el estado de los diferentes espectrómetros.
- A lo largo de los últimos años, el sensor ha ido mejorando sus prestaciones en cuanto a la relación señal a ruido, como se muestra en la figura 3.4., que describe la evolución de la relación SNR del sensor a lo largo de los últimos años.

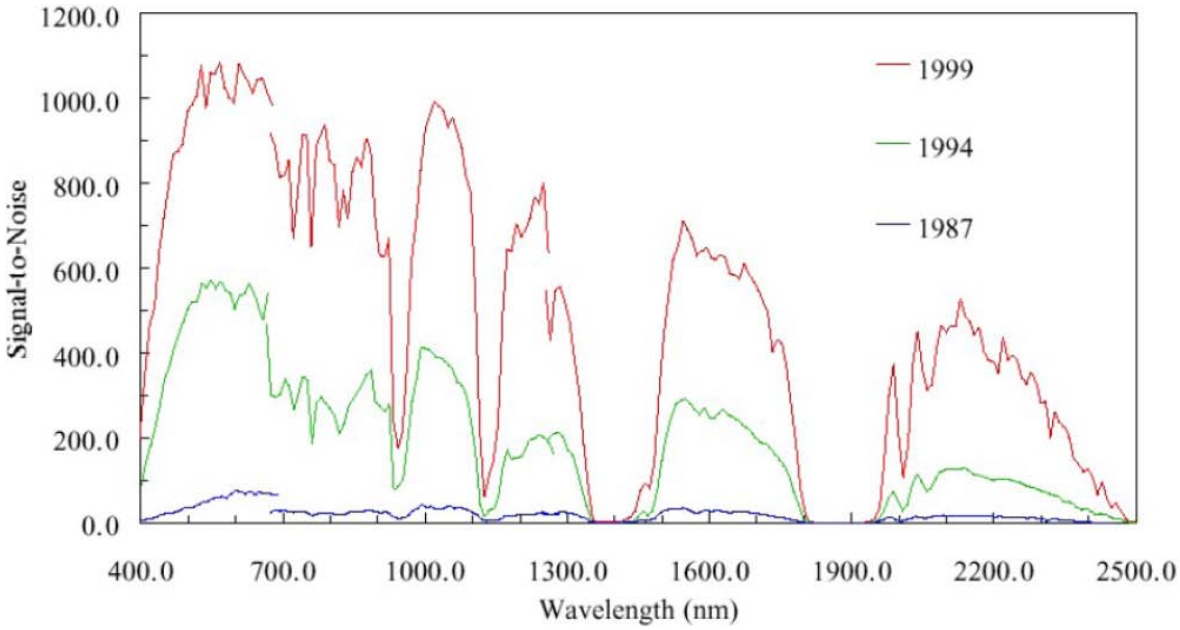

Figura 3.4. Evolución de la relación señal a ruido (SNR) del sensor AVIRIS.

3.3. Técnicas de análisis hiperespectral y necesidad de paralelismo

La mayoría de las técnicas de análisis hiperespectral desarrolladas hasta la fecha presuponen que la medición obtenida por el sensor en un determinado píxel viene dada por la contribución de diferentes materiales que residen a nivel sub–píxel. El fenómeno de la mezcla puede venir ocasionado por una insuficiente resolución espacial del sensor. La realidad es que este fenómeno ocurre de forma natural en el mundo real, incluso a

niveles microscópicos, por lo que el diseño de técnicas capaces de modelarlo de manera adecuada resulta imprescindible. No obstante, las técnicas basadas en este modelo son altamente costosas desde el punto de vista computacional. A continuación, detallamos las características genéricas de las técnicas basadas en este modelo y hacemos énfasis en la necesidad de técnicas paralelas para optimizar su rendimiento computacional.

3.3.1. Técnicas basadas en el modelo lineal de mezcla

Dependiendo de la resolución espacial del sensor, la mayor parte de los píxeles de una imagen hiperespectral corresponden a una mezcla de componentes puros a nivel macroscópico. En la figura 3.5. puede observarse dicho efecto. Estos píxeles mezcla pueden expresarse como una combinación de elementos espectralmente puros, denominados *endmembers*.

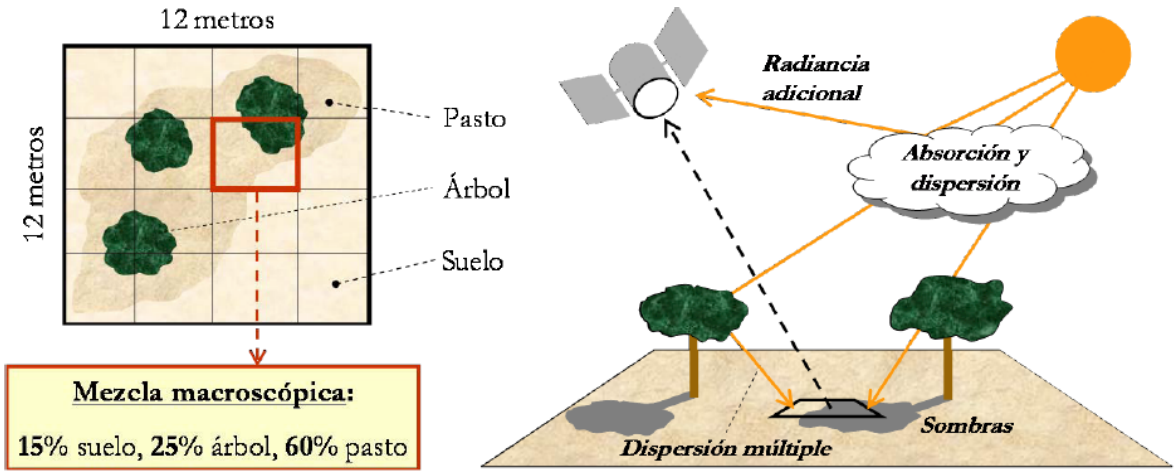

Figura 3.5. El problema de la mezcla en las imágenes hiperespectrales.

Para tratar de tratar de hacer frente al problema anterior, el modelo lineal de mezcla expresa los píxeles mezcla [49] como una combinación lineal de firmas asociadas a componentes espectralmente puros (*endmembers*) en la imagen [12]. Este modelo ofrece resultados satisfactorios cuando los componentes que residen a nivel sub–píxel aparecen espacialmente separados, situación en la que los fenómenos de absorción y reflexión de la radiación electromagnética incidente pueden ser caracterizados siguiendo un patrón

estrictamente lineal. En la actualidad, el modelo lineal de mezcla es el más utilizado en análisis hiperespectral, debido a su sencillez y generalidad.

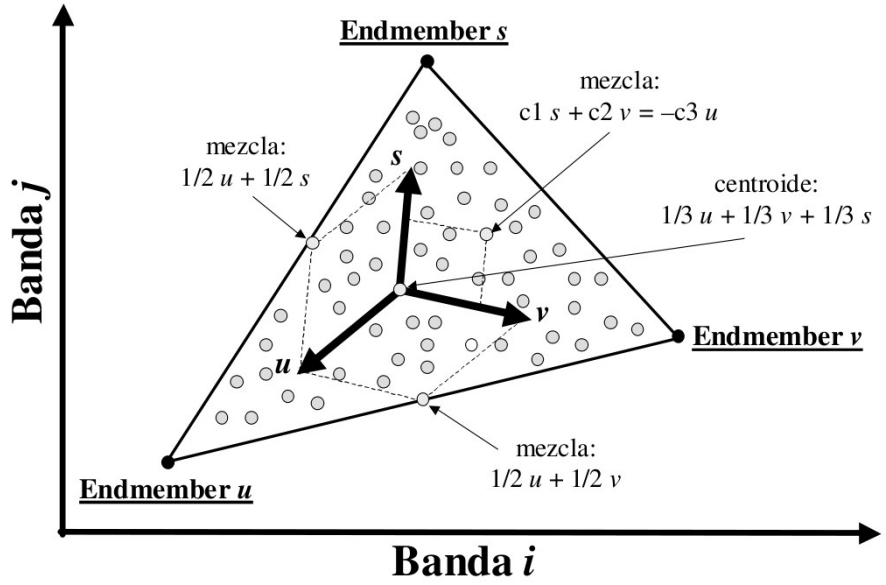

Figura 3.6. Interpretación gráfica del modelo lineal de mezcla.

El modelo lineal de mezcla puede interpretarse de forma gráfica en un espacio bidimensional utilizando un diagrama de dispersión entre dos bandas poco correlacionadas de la imagen, tal y como se muestra en la figura 3.6. En la misma puede apreciarse que todos los puntos de la imagen quedan englobados dentro del triángulo formado por los tres puntos más extremos (elementos espectralmente más puros). Los vectores asociados a dichos puntos constituyen un nuevo sistema de coordenadas con origen en el centroide de la nube de puntos, de forma que cualquier punto de la imagen puede expresarse como combinación lineal de los puntos más extremos, siendo estos puntos los mejores candidatos para ser seleccionados como *endmembers* [13]. El paso clave a la hora de aplicar el modelo lineal de mezcla consiste en identificar de forma correcta los elementos extremos de la nube de puntos N–dimensional. En la literatura reciente se han propuesto numerosas aproximaciones al problema de identificación de *endmembers* en imágenes hiperespectrales. Por ejemplo, el método *Vertex Component Analysis* (VCA) [1] se basa en la generación de vectores aleatorios que dan lugar a una serie de vectores ortonormales al subespacio contemplado y en el que hay que encontrar los extremos de las proyecciones generadas. Tras la ejecución de un número de

iteraciones igual al número de *endmembers* que se quiere encontrar, cada uno de los extremos de las proyecciones equivale a un *endmember*.

3.3.2. Necesidad de paralelismo

La mayoría de técnicas de análisis hiperespectral se basan en la realización de operaciones matriciales que resultan muy costosas desde el punto de vista computacional [50]. No obstante, el gran número de iteraciones que se tienen que realizar para obtener los *endmembers*, las hace altamente susceptibles de ser implementadas en diferentes tipos de arquitecturas paralelas. Así se consigue una reducción significativa de los tiempos de ejecución. Éste es un aspecto clave para poder emplear dichas técnicas en aplicaciones que precisan de una respuesta en tiempo casi real.

Las técnicas de computación paralela han sido ampliamente utilizadas para llevar a cabo tareas de procesamiento de imágenes de gran dimensionalidad, facilitando la obtención de tiempos de respuesta muy reducidos y pudiendo utilizar diferentes tipos de arquitecturas [51 – 53]. En la actualidad, es posible obtener arquitecturas paralelas de bajo coste mediante la utilización de GPUs de última generación que cuentan con múltiples procesadores.

3.3.3. El papel de las GPUs

Una de las tareas más importantes en el tratamiento de datos hiperespectrales, sino la que más, es la extracción de endmembers. En los últimos años se han desarrollado muchos algoritmos para la extracción automática de *endmembers*: PPI, N–FINDR, VCA, ATGP, análisis de componentes vértices, o IEA.

Estas técnicas hiperespectrales introducen un nuevo reto de procesamiento particularmente para conjunto de datos de alta dimensionalidad. Desde un punto de vista computacional cada algoritmo muestra un patrón de acceso a los datos regular y que muestra un paralelismo inherente a muchos niveles: a nivel de vectores de píxeles, a nivel

de información espectral e incluso a nivel de tarea. Como resultado se asocian con sistemas paralelos compuestos por CPUs (por ejemplo clústeres Beowulf). Desafortunadamente estos sistemas son caros y difíciles de adaptar a bordo de escenarios de procesamiento remotos.

Un nuevo mundo surge en el campo de la computación con los procesadores gráficos programables (GPUs). Guiadas por la creciente demanda de la industria de los videojuegos, las GPUs han evolucionado como sistemas programables altamente paralelos. En el caso concreto de los algoritmos de imágenes hiperespectrales, resulta más que interesante beneficiarse de las GPUs, ya que de esta forma se sacaría partido de sus ventajas, sobre todo en lo referente a la potencia de cálculo que ofrecen.

3.4. El algoritmo Vertex Component Analysis

Hace unos años, concretamente en 2005, [1] propuso un algoritmo que ofrece unos resultados bastante buenos en lo referente a bondad de la extracción de *endmembers* y complejidad de la solución.

El algoritmo trata de descomponer espectralmente y de manera lineal los vectores mezcla de la imagen en sus correspondientes componentes puros (*endmembers*). Es un algoritmo no supervisado, que se basa en dos aspectos: por un lado los *endmembers* son los vértices de un símplex, y la transformación afín de un símplex es también otro símplex. El algoritmo funciona tanto con imágenes proyectadas a otro espacio y reduciendo la dimensión espectral de la imagen, como con las imágenes originales que no sufren un preprocesado previo.

De una manera iterativa se van proyectando los valores de la imagen sobre una dirección perpendicular al subespacio determinado por los endmembers ya calculados. El nuevo *endmember* equivale al valor extremo de esta proyección. Se producen tantas iteraciones como *endmembers* se deseen calcular, es decir, “p” iteraciones. Este detalle hace que sea

menos costoso que otras alternativas. En la figura 3.7. se representa de forma gráfica el funcionamiento del algoritmo.

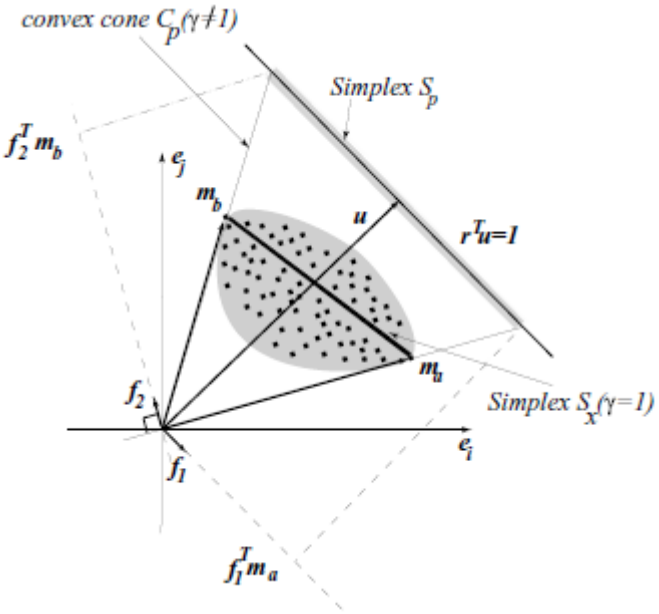

Figura 3.7. Representación del algoritmo *Vertex Component Analysis*.

El algoritmo representado en pseudocódigo queda como sigue:

ENTRADA p, $R \equiv [r_1, r_2, \ldots, r_N]$

1: $SNR_{th} = 15 + 10log_{10}(p)$ *dB*

2: si $SNR > SNR_{th}$ *entonces*

3: $d := p;$

4: $X := U_d^T R$; *{*U_d *obtenido por SVD, descomposición en valores singulares}*

5: $u :=$ *media(X); {u es un vector de 1 x d}*

6: $[Y]_{:,j} := [X]_{:,j} / ([X]_{:,j}^T u)$; *{proyección}*

7: sino

8: $d := p - 1;$

9: $[X]_{:,j} := U_d^T ([R]_{:,j} - \bar{r})$;*{*$U_d$ *obtenido por análisis del componente principal}*

10: $c := argmax_{j=1\ldots N} ||[X]_{:,j}||;$

11: $c := [c|c| \ldots |c]$; *{c es un vector de 1 x N]*

12: $Y := \begin{bmatrix} X \\ c \end{bmatrix};$

13: fin si

14: A := [eu|0| … |0]; {$e_u = [0, \dots, 0, 1]^T$ y A es una matriz auxiliar de p x p}

15: para i := 1 hasta p hacer

16: w := randn(0, Ip); {w vector Gaussiano aleatorio de media cero y covarianza I_p}

17: $f := ((I - AA^{\#})w/(||(I - AA^{\#})w||)$; {f vector ortonormal al subespacio de $[A]_{:,1:i}$}

18: $v := f^T Y$;

19: k := argmaxj=1, … , N |[v]:,j|; {encuentra el extremo de la proyección}

20: $[A]_{:,i} := [Y]_{:,k}$;

21: $[índice]_i := k$; {guarda el índice del píxel}

22: fin para

23: si $SNR > SNR_{th}$

24: $\hat{M} := U_d[X]_{:,\,índice}$;{ $\hat{M}$ es una matriz estimada de mezcla de L x p}

25: sino

26: $\hat{M} := U_d[X]_{:,\,índice}$; { $\hat{M}$ es una matriz estimada de mezcla de L x p}

27: fin si

Las líneas de la 1 a la 13 realizan una proyección de la matriz de entrada (imagen) a un espacio calculado o bien empleando la transformación SVD a un espacio de "p" dimensiones o bien la transformación PCA a un espacio de "p – 1" dimensiones, en función de que la relación señal ruido (SNR) de la misma sea o no superior a un umbral, que se calcula en la primera instrucción del algoritmo. Esta primera fase que se ha descrito constituye la fase de preprocesado de la imagen. Destacar que los pasos 4 y 9 aseguran que ninguno de los productos escalares entre "$[X]_{:,i}$" y "u" sean negativos, aspecto crucial para el correcto funcionamiento del algoritmo VCA.

Del presente trabajo hay que subrayar, en primer lugar, que se ha realizado un estudio detallado del algoritmo VCA para analizar las posibles modificaciones que permitieran implementar de manera más sencilla y eficaz el algoritmo en hardware.

A partir de la línea 14, comienza el algoritmo VCA con la inicialización de la matriz "A" , matriz sobre la que se irán almacenando las proyecciones de los *endmembers* que se vayan encontrando, de la forma en que se muestra, todos sus columnas a cero,

exceptuando la primera que está ocupada por el vector "e_u = [0, 0, ... , 1]'". Esta inicialización de la matriz "A" produce la reducción de una de las dimensiones, obteniéndose todos los puntos proyectados en un espacio de "p – 1" dimensiones. Este hecho es fundamental, ya que se recuerda que el simplex se obtiene proyectando los datos sobre un espacio de estas características.

Una vez inicializada la matriz, se entra en el bucle, línea 15, que calcula en cada iteración un vector "f", ortonormal a la matriz "A", a través de la creación de un vector aleatorio, que es de esta naturaleza para evitar realizar una proyección sobre las columnas de la matriz "A" de un vector nulo. Una vez calculado "f", en la línea 18 se realiza la proyección de la matriz "Y" sobre este vector, y se almacena en "v", de tal forma que en la siguiente línea se calcula el índice del valor mayor absoluto del vector, y se almacena en "k".

En la línea 20, con este índice calculado, se selecciona de la matriz "Y" el vector, que supondrá la proyección del nuevo endmember, y posteriormente se guarda el índice "k" en un vector de "p" componentes.

Las líneas 23 a 27 se encargan de reconstruir la imagen al espacio original de "L" bandas.

Capítulo 4. Unidad de procesamiento gráfico (GPU)

En este capítulo se realiza una breve descripción sobre las tarjetas gráficas programables GPUs y el entorno de programación empleado para programar en ellas

4.1. Las GPU como dispositivos de cálculo

Desde hace unos años las GPUs han ido evolucionando, convirtiéndose en elementos cada vez más complejos y capaces de realizar una cantidad ingente de operaciones por segundo. En las figuras 4.1. y 4.2. se muestra una comparativa de la evolución de la capacidad de cómputo y del ancho de banda de las CPUs y las GPUs, respectivamente. Con múltiples núcleos y con un gran ancho de banda de memoria, hoy día las GPUs ofrecen prestaciones muy elevadas para procesamiento gráfico y científico [2 – 6].

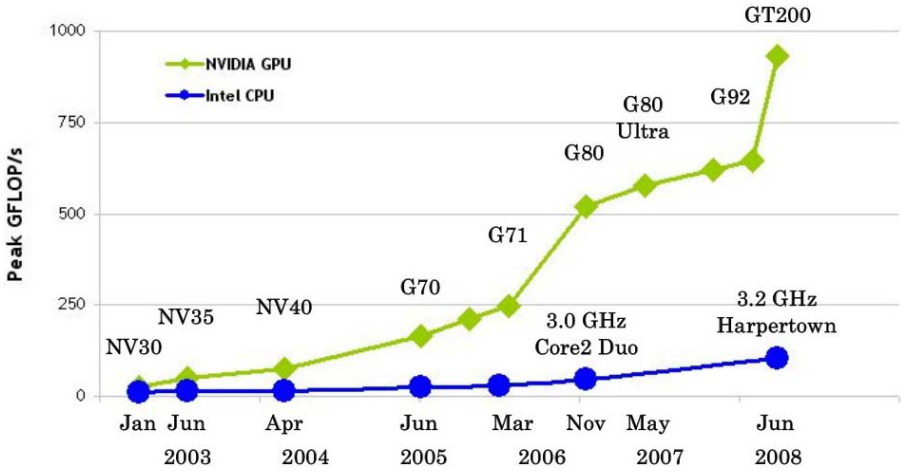

Figura 4.1. Comparación de FLOPS entre CPU y GPU.

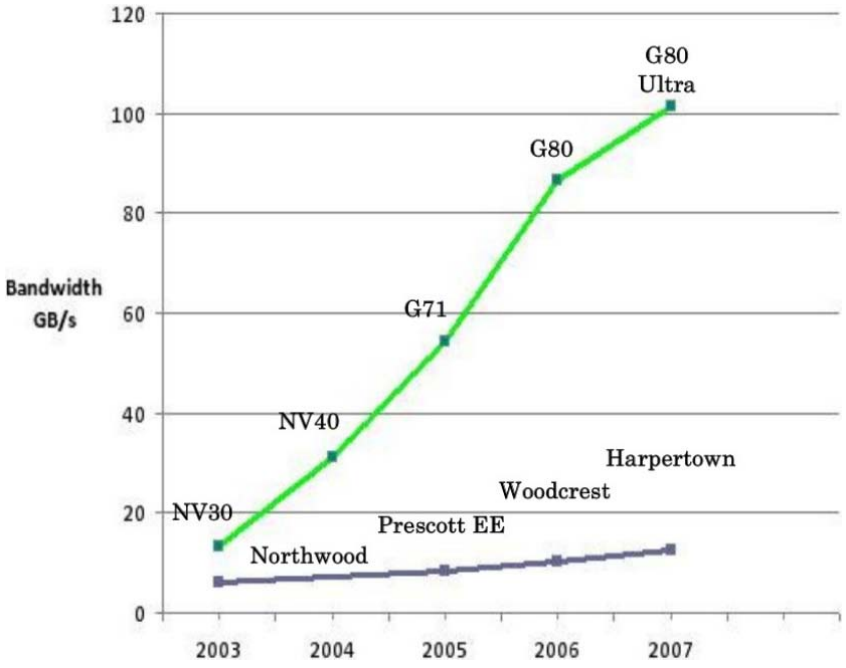

Figura 4.2. Comparación de anchos de banda de CPU y GPU.

A la vista de las dos figuras anteriores, queda patente la espectacular evolución que han sufrido las GPUs en cuanto a potencia de cálculo y ancho de banda. Gracias a ello, se han vuelto dispositivos muy interesantes para ser empleados en aplicaciones científicas.

4.1.1. Evolución del uso de GPUs en aplicaciones científicas

A las aplicaciones científicas les resultan interesantes las arquitecturas GPU por el hecho de que éstas están especializadas para el cómputo intensivo y para el paralelismo. Las GPU tienen más transistores dedicados al procesamiento de datos que a la transferencia de los mismos y al control de flujo. Estas diferencias entre la CPU y la GPU se ven claramente en la figura 4.3.

Figura 4.3. Arquitectura de una CPU y de una GPU. [NVIDIA]

Este paralelismo en el procesamiento de datos se consigue asociando datos a elementos de proceso paralelos. Las aplicaciones que procesan grandes conjuntos de datos en forma de vectores o matrices pueden beneficiarse de un modelo de programación de datos paralelos para acelerar los cálculos. En renderizado 3D los conjuntos de píxeles y vértices se asignan a hilos paralelos. De la misma manera, aplicaciones de procesamiento de imágenes y multimedia como postprocesado de imágenes renderizadas, codificación y decodificación de vídeo, escalado de imágenes, visión estéreo, y patrones de reconocimiento, pueden asociar bloques de la imagen y píxeles a hilos de procesamiento paralelo. De hecho, muchos algoritmos fuera del campo del renderizado como el procesamiento de señales, simulaciones físicas finanzas o biología, se aceleran con el procesamiento de datos en paralelo.

Hasta la fecha, sin embargo, a pesar de acceder a todo el poder de computación contenido en al GPU y usarlo eficientemente para aplicaciones científicas, seguía siendo difícil obtener las siguientes pautas:

- La GPU solamente podía ser programada a través de la API (*Application Programming Interface*) gráfica; esto provocaba que la curva de aprendizaje para un desarrollador principiante fuese muy elevada al tener que trabajar con una API inadecuada que no estaba adaptada a la aplicación científica.
- La DRAM de la GPU podía ser leída de manera general (los programas de GPU pueden obtener elementos de datos de cualquier parte de la DRAM) pero no se podía escribir de manera general (los programas de GPU no pueden escribir la información en cualquier parte de la DRAM), quitándole flexibilidad a la programación ya disponible en la CPU.
- Algunas aplicaciones tenían el problema del "cuello de botella", debido al ancho de banda de la memoria DRAM, utilizando escasamente la potencia computacional de la GPU.

En este sentido, una de las principales motivaciones del presente TFM es demostrar que dichas limitaciones en la actualidad pueden superarse mediante la utilización de la arquitectura CUDA para procesamiento de datos científicos en la GPU. Dicho aspecto será abordado en detalle en el siguiente subapartado del presente capítulo de la memoria.

4.2. CUDA: una nueva arquitectura para la computación en paralelo

CUDA son las siglas en inglés de *Compute Unified Device Architecture* y es una nueva arquitectura hardware y software. Está diseñada para realizar cálculos en la GPU como un elemento de computación de datos de una forma sencilla. CUDA está disponible para las familias GeForce 8XXX/9XXX/2XX/4XX/5XX, Quadro FX/NVS/Plex, ION y Tesla [54]. El mecanismo de multitarea del sistema operativo es responsable de manejar el acceso a la GPU mediante CUDA, y las aplicaciones gráficas funcionan de forma simultánea. A

continuación se describe el pipeline unificado del que disponen las actuales GPUs de NVIDIA y que puede ser explotado de forma eficiente mediante CUDA, así como la arquitectura completa de la GeForce GTX 480.

4.2.1. El entorno de programación MATLAB

El lenguaje elegido para desarrollar este proyecto ha sido MATLAB (*Matrix Laboratory*), estando motivada esta elección por su gran difusión en el ambiente universitario, tanto a nivel docente como de investigación. Posee una caja de herramientas para el procesamiento de imágenes con funciones incorporadas que facilitarán en gran medida el desarrollo del entorno software. Hay que considerar que si MATLAB es utilizado por multitud de universidades y empresas de I+D, es porque se considera una inversión rentable. Tiene unos recursos y un potencial muy por encima de otras aplicaciones rivales del mismo tipo.

Uno de sus puntos débiles es que se trata de un software propietario, que no es de libre distribución. Esto encarece notablemente el proyecto, ya que para poder utilizar la interfaz es necesario tener instalado MATLAB en el ordenador. Cabe señalar que ya es posible realizar ejecutables que no necesiten de MATLAB, pero éste no es el caso.

MATLAB es un lenguaje de alto rendimiento orientado a las comunidades de científicos, ingenieros y matemáticos. Estas comunidades siempre requieren de herramientas adecuadas para el cálculo, programación y visualización de datos y resultados. MATLAB es una de las mejores aplicaciones capaces de proporcionar al mismo tiempo grandes capacidades computacionales y gráficas en un entorno fácil de usar. Es, además, un sistema interactivo cuyo elemento de datos básico es un array sin dimensiones. Este detalle es fundamental a la hora de resolver muchos problemas de cómputo, especialmente en lo que se refiere al tiempo empleado. Este último es mucho menor que el requerido para escribir un programa en un lenguaje escalar no interactivo como C o Fortran. MATLAB se utiliza ampliamente en:

- Matemáticas y computación
- Desarrollo de algoritmos
- Modelado, simulación y prueba de prototipos
- Análisis de datos, exploración y visualización
- Gráficos
- Desarrollo de aplicaciones que requieran de una interfaz gráfica de usuario

Se ve, por tanto, que va a ser posible escribir los algoritmos de una manera más rápida y sencilla, visualizar y analizar los datos obtenidos, para finalmente crear una interfaz que englobe todos los algoritmos y permita analizarlos.

El sistema de MATLAB lo conforman 5 partes principales:

1. **Entorno de desarrollo:** es el conjunto de herramientas y facilidades que ayudan al usuario a utilizar funciones y ficheros de MATLAB. Incluye, por ejemplo, la barra de herramientas y la ventana de comandos de MATLAB.
2. **Librería de funciones matemáticas:** colección de algoritmos computacionales desde las funciones más elementales (sin, sum...) a las funciones más sofisticadas (matriz inversa, autovalores, funciones de Bessel, transformadas de Fourier, lectura de ficheros HDF...).
3. **El lenguaje MATLAB:** lenguaje matriz/array de alto nivel que controla funciones, estructuras de datos, entrada/salida, y características de programación orientada a objetos. Es capaz de crear desde programas muy sencillos hasta programas muy extensos y complicados.
4. **Manipulación de gráficos:** permite la visualización de datos en 2 y 3 dimensiones, procesado de imágenes, animación y presentación de gráficos. Permite personalizar la apariencia de los gráficos así como crear interfaces gráficas.
5. **API (Application Program Interface) de MATLAB:** permite escribir programas en C y Fortran que interactúan con MATLAB.

Aunque se trata de un sistema muy completo para el desarrollo de algoritmos complicados, a ciertos niveles de complejidad resulta bastante más lento que otros lenguajes de programación, como por ejemplo C/C++ o Python.

4.2.2. Parallel Computing Toolbox

Para el desarrollo de este trabajo de fin de máster se ha utilizado la versión r2011a de MATLAB, con el *Parallel Computing Toolbox* instalado. Un problema que posee MATLAB es que algunas funciones básicas propias del lenguaje de programación o sus nombres pueden cambiar de una versión a otra, por lo que con esta versión no se garantiza el pleno funcionamiento del software desarrollado en otras versiones, ya sean anteriores o posteriores. Los cambios que habría que hacer para que funcionara en otras versiones serían mínimos, aunque posiblemente de tarea laboriosa.

El *Parallel Computing Toolbox* está disponible desde noviembre del año 2004 [7], cuando se lanzó su primera versión. Con el paso de los años se han ido incorporando numerosas funciones y funcionalidades que hacen que éste sea muy interesante para intentar sacar el máximo partido posible a los nuevos procesadores multinúcleo y multihilo. Por si fuera poco, desde la versión de MATLAB r2009b, se da soporte nativo a las GPUs, pudiendo hacer uso de CUDA en el propio entorno de programación.

4.2.3. Modelo de programación CUDA y MATLAB

A partir de MATLAB r2009b, ciertas GPUs de NVIDIA se pueden programar directamente desde MATLAB empleando para ello el *Parallel Computing Toolbox*. El único requisito es que la GPU tenga un *compute capability* mayor o igual a 1.3.

Esta funcionalidad evita tener que pasar por el entorno de programación de CUDA, pero a la vez tampoco lo excluye; el *Parallel Computing Toolbox* también permite realizar llamadas a funciones CUDA desde MATLAB y que sean ejecutadas en este último. Así se simplifica notablemente la programación de las GPUs y se hace más accesible la

programación de las mismas. Es decir, se puede programar la GPU sólo empleando MATLAB, o se puede programar una parte en MATLAB y otra en CUDA (realizándose la ejecución en MATLAB y haciendo la correspondiente llamada a la función correspondiente de CUDA cuando sea necesario).

Un usuario con conocimientos de MATLAB simplemente tiene que identificar las instrucciones que le permiten programar sobre la GPU y poco más; puede dedicarse a programar prácticamente como lo haría en condiciones normales, aunque teniendo en cuenta que existen ciertas limitaciones a la hora de programar:

- No se puede acceder a los procesadores de la GPU de forma individual, algo que sí se puede hacer en CUDA. MATLAB trata de maximizar su uso de forma automática, pero no permite que el usuario tome el control para intentar optimizar al máximo la aplicación.
- Existen una serie de instrucciones con restricciones en cuanto a su uso en la GPU, mientras otras ni siquiera se pueden ejecutar en ella.
- La transferencia de datos hacia la GPU se produce sin que el usuario tenga control sobre ella, simplemente se mueven de una memoria a otra memoria. En CUDA, en cambio, se puede controlar cómo se reparten dichos datos en la memoria de la GPU (memoria local y memoria compartida).

4.3. Implementación hardware

En este apartado se presenta la arquitectura hardware de la GPU empleada en el TFM. A lo largo de la presente memoria se han cubierto los aspectos principales de la programación de GPUs de NVIDIA en MATLAB, así que ahora podemos echar un vistazo a los aspectos específicos de la arquitectura GeForce GTX 480, la tarjeta que se está usando para realizar este TFM. La Figura 4.4 muestra la arquitectura hardware de dicha tarjeta.

Figura 4.4. Arquitectura del chip GF100. [NVIDIA]

Este chip GF100 está formado por hasta 15 *Streaming Multiprocessor* (SM), que a su vez contienen un total de 32 *CUDA cores*. Mediante un sencillo cálculo, se llega a la conclusión de que la NVIDIA GeForce GTX 480 posee 480 *CUDA cores*. La arquitectura de los *Streaming Multiprocessor* se muestra en la figura 4.5.

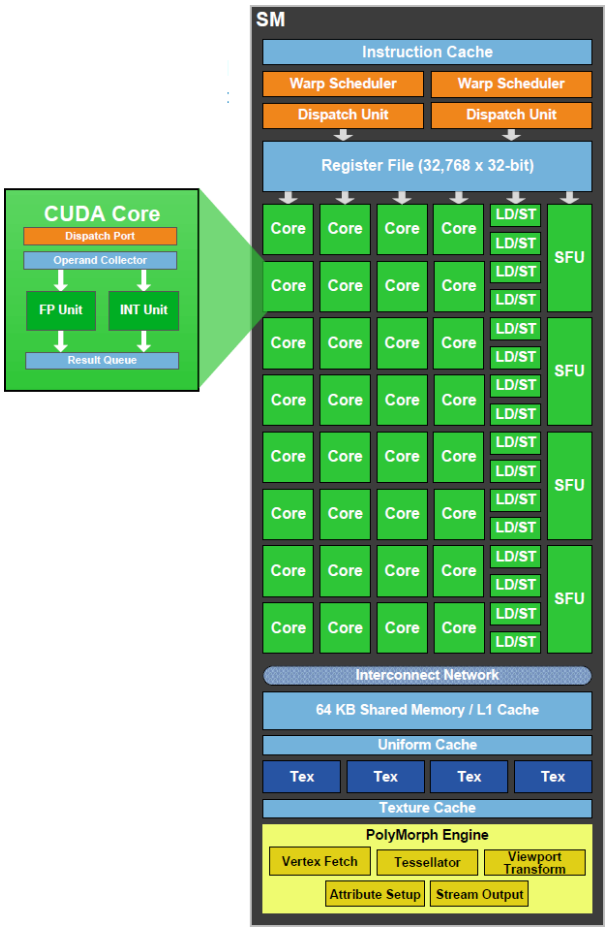

Figura 4.5. Arquitectura de los SM. [NVIDIA]

En la figura 4.6. se listan las características técnicas de la GPU.

	Graphics Card	GeForce GTX 480
Processing Units	Graphics Processing Clusters	4
	Streaming Multiprocessors	15
	CUDA Cores	480
	Texture Units	60
	ROP Units	48
Clock Speeds	Graphics Clock (Fixed Function Units)	700 MHz
	Processor Clock (CUDA Cores)	1401 MHz
	Memory Clock (Clock rate / Data rate)	924 MHz / 3696 MHz
Memory	Total Video Memory	1536 MB
	Memory Interface	384-bit
	Total Memory Bandwidth	177.4 GB/s

Figura 4.6. Características técnicas de la NVIDIA GeForce GTX 480. [NVIDIA]

Por último, en la figura 4.7. se indican los recursos de las GPU en función de su *compute capability* (generación GPU en la tabla). La NVIDIA GeForce GTX 480 posee *compute capability* Fermi.

Parámetro	Valor según gener. GPU			Limitación	Impacto
CUDA Compute Capabilities	1.0 y 1.1	1.2 y 1.3	Fermi		
Multiprocesadores / GPU	16	30	16	HW.	Escalabilidad
Procesadores / Multiprocesador	8	8	32	HW.	Escalabilidad
Hilos / Warp	32	32	32	SW.	Throughput
Bloques de hilos / Multiprocesador	8	8	8	SW.	Throughput
Hilos / Bloque	512	512	512	SW.	Paralelismo
Hilos / Multiprocesador	768	1024	1536	SW.	Paralelismo
Registros de 32 bits / Multiproc.	8192	16384	4096	HW.	Working Set
Memoria compartida / Multiproc.	16384	16384	16 K 48K	HW.	Working Set

Figura 4.7. Recursos de las GPU. [NVIDIA]

Capítulo 5. Metodología de implementación

En este capítulo se desarrolla la metodología de alto nivel para implementar el algoritmo *Vertex Component Analysis* en la GPU de NVIDIA.

5.1. Opciones disponibles para el desarrollo de la metodología de alto nivel

A la hora de implementar el algoritmo *Vertex Component Analysis* en una GPU existen varios caminos bien diferenciados. De entre los disponibles se ha optado por la implementación mediante MATLAB y CUDA. Ésta explicará en detalle en el presente capítulo de la memoria.

5.1.1. Implementación mediante MATLAB y CUDA

El potencial que ofrece MATLAB y su integración con CUDA lo hace, a priori, muy atractivo para tratar de implementar el algoritmo VCA en una GPU. El *Parallel Computing Toolbox* posee una serie de funciones específicas para programar directamente sobre la GPU, disminuyendo el tiempo de desarrollo necesario si se compara con una implementación directa en CUDA. Esta última es bastante más compleja y tediosa al tener que considerar todas las características especiales de la GPU, las especificidades de su arquitectura y del lenguaje de programación. Este último es similar a C/C++, pero posee una serie de particularidades.

La metodología de alto nivel desarrollada para implementar el algoritmo en una GPU empleando MATLAB/CUDA puede dividirse en las siguientes etapas:

1. Estudio del entorno MATLAB/CUDA e identificación de las funciones disponibles más interesantes.
2. Estudio del algoritmo e identificación de los segmentos susceptibles de ser programados en la GPU.
3. Programación del algoritmo en la GPU.
4. Validación y pruebas.
5. Modificaciones y mejoras.
6. Resultados.

Antes de nada hay que asegurarse de que el sistema dispone de una GPU compatible. Para ello existen dos funciones:

- *D = gpuDevice*: devuelve un objeto que representa la GPU seleccionada actualmente.

- *n = gpuDeviceCount*: devuelve el número de dispositivos GPU presentes en el ordenador.

Al ejecutar la primera de ellas en MATLAB se lee lo siguiente:

```
gpuDevice

ans =

  parallel.gpu.CUDADevice handle
  Package: parallel.gpu

Properties:
      Name: 'GeForce GTX 480'
      Index: 1
      ComputeCapability: '2.0'
      SupportsDouble: 1
      DriverVersion: 4
      MaxThreadsPerBlock: 1024
      MaxShmemPerBlock: 49152
      MaxThreadBlockSize: [1024 1024 64]
      MaxGridSize: [65535 65535]
      SIMDWidth: 32
      TotalMemory: 1.5430e+009
      FreeMemory: 1.3895e+009
      MultiprocessorCount: 15
      ComputeMode: 'Default'
      GPUOverlapsTransfers: 1
      KernelExecutionTimeout: 1
      CanMapHostMemory: 1
      DeviceSupported: 1
      DeviceSelected: 1
```

El diagnóstico del sistema confirma que una GeForce GTX 480 está instalada y que tiene un *compute capability* de 2.0, por lo que puede ser programada desde MATLAB. El resto de datos están relacionados con las características técnicas de la GPU en cuestión, como pueden ser el número máximo de hilos por bloque, la memoria total o el número de procesadores de los que dispone.

Una vez confirmado que se dispone de una GPU compatible con la programación en CUDA desde MATLAB, el punto de partida es el código en MATLAB del algoritmo VCA, que facilitan los propios autores del mismo [10]:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% VCA algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

indice = zeros(1,p);
A = zeros(p,p);
A(p,1) = 1;

for i=1:p
      w = rand(p,1);
      f = w - A*pinv(A)*w;
      f = f / sqrt(sum(f.^2));
      v = f'*y;
      [v_max indice(i)] = max(abs(v));
      A(:,i) = y(:,indice(i));        % same as x(:,indice(i))
end
Ae = Rp(:,indice);

return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% End of the vca function
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Como se puede comprobar, el algoritmo no es demasiado largo, limitándose a seis líneas de código dentro de un bucle que se repite tantas veces como *endmembers* se quieren encontrar.

A continuación se pasará a describir la metodología desarrollada, de la cual se muestra el flujo de diseño en la figura 5.1.

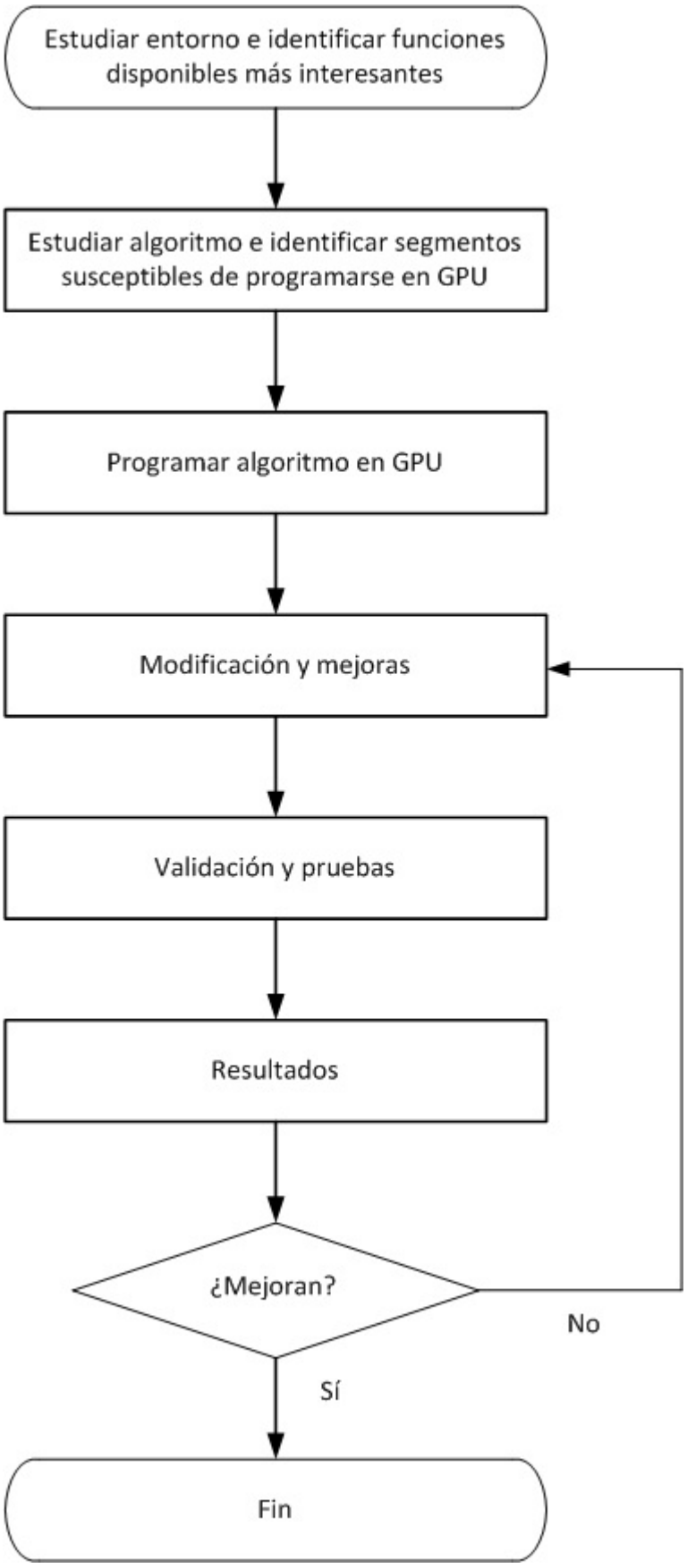

Figura 5.1. Flujo de diseño.

5.1.1.1. Estudio del entorno MATLAB y CUDA e identificación de las funciones disponibles más interesantes

El primer paso es identificar las funciones más importantes e interesantes disponibles en MATLAB/CUDA para programar en GPUs:

- *G = gpuArray(x)*: copia los datos numéricos *x* en la GPU y devuelve un objeto *GPUArray*. Se puede operar en estos datos pasándolos al método *feval* de un objeto kernel de CUDA, o usando uno de los métodos definidos para los objetos *GPUArray*.

- *X = gather(x)*: puede operar dentro de una declaración *spmd*, *pmode*, o tarea paralela para recoger los datos de un array codistribuido, o externa a una declaración *spmd* para recoger los datos de un array distribuido. Si se ejecuta dentro de una declaración *spmd*, *pmode*, o tarea paralela, *X* es un array copia con todos los datos del array en cada laboratorio. Si se ejecuta fuera de una declaración *spmd*, *X* es un array en el área de trabajo local, con los datos transferidos desde los múltiples laboratorios.

- *KERN = parallel.gpu.CUDAKernel(PTXFILE, CPROTO)* y *KERN = parallel.gpu.CUDAKernel(PTXFILE, CPROTO, FUNC)*: crean un objeto kernel que se puede emplear para realizar una llamada a un kernel CUDA en la GPU. *PTXFILE* es el nombre del fichero que contiene el código *PTX*, y *CPROTO* es la llamada C prototipo al kernel que representa a *KERN*. Si se especifica, *FUNC* tiene que ser una cadena de caracteres que defina sin ambigüedades el nombre del kernel apropiado en el fichero *PTX*. Si se omite *FUNC*, el fichero *PTX* sólo puede contener un único punto de entrada.

- *A = arrayfun(FUN, B)*: aplica la función indicada por *FUN* a cada uno de los elementos del *GPUArray B*, y devuelve los resultados en el *GPUArray A*. *A* es del mismo tamaño que *B*, y *A(i, j, …)* es igual a *FUN(B(i, j, …))*. *FUN* es un puntero a

una función que convierte un parámetro de entrada y devuelve un valor escalar. *FUN* debe devolver valores de la misma clase cada vez que se le llama. Los datos de entrada tienen que ser arrays de uno de los tipos siguientes: *single*, *double*, *int32*, *uint32*, *logical* o *GPUArray*. El orden en que *arrayfun* calcula los elementos de *A* no está determinado y no debería confiarse en él.

Las funciones indicadas anteriormente son las más interesantes que ofrece MATLAB/CUDA para interactuar con la GPU. Por otro lado, una serie de funciones internas de MATLAB soportan el uso de *GPUArray*. Cuando cualquiera de estas funciones es llamada con al menos un elemento *GPUArray* como entrada, se ejecuta en la GPU y devuelve un elemento *GPUArray* como resultado.

abs	conj	fft	log	rem
acos	conv	fft2	log10	reshape
acosh	conv2	fix	log1p	round
acot	cos	floor	log2	sec
acoth	cosh	gamma	logical	sech
acsc	cot	gammaln	lt	sign
acsch	coth	gather	lu	sin
all	csc	ge	max	single
any	csch	gt	meshgrid	sinh
arrayfun	ctranspose	horzcat	min	size
asec	cumprod	hypot	minus	sqrt
asech	cumsum	ifft	mldivide *	subsasgn
asin	diag	ifft2	mod	subsindex
asinh	disp	imag	mrdivide *	subsref
atan	display	int16	mtimes	sum
atan2	dot	int32	ndgrid	tan
atanh	double	int64	ndims	tanh
bitand	eps	int8	ne	times
bitcmp	eq	isempty	numel	transpose
bitor	erf	isequal	plot	tril
bitshift	erfc	isequalwithequalnans	plus	triu
bitxor	erfcinv	isfinite	power	uint16
cast	erfcx	isinf	prod	uint32
cat	erfinv	islogical	rdivide	uint64
ceil	exp	isnan	real	uint8
classUnderlying	expm1	isreal	reallog	uminus
colon	filter	ldivide	realpow	uplus
complex	filter2	le	realsqrt	vertcat
		length		

* *mldivide*/*mrdivide* no soportan datos *GPUArray* complejos.

Para obtener ayuda específica sobre estas funciones sobrecargadas, y para conocer cualquier restricción relativa a su uso con objetos *GPUArray*, hay que escribir:

```
help parallel.gpu.GPUArray/functionname
```

Por ejemplo, para recibir ayuda sobre la función *lu* sobrecargada:

```
help parallel.gpu.GPUArray/lu
```

En lo que respecta a *arrayfun*, ésta soporta las siguientes funciones internas de MATLAB y operadores:

```
abs          csc          log10        +
acos         csch         log1p        -
acosh        double       logical      .*
acot         eps          max          ./
acoth        erf          min          .\
acsc         erfc         mod          .^
acsch        erfcinv      NaN          ==
asec         erfcx        pi           ~=
asech        erfinv       real         <
asin         exp          reallog      <=
asinh        expm1        realpow      >
atan         false        realsqrt     >=
atan2        fix          rem          &
atanh        floor        round        |
bitand       gamma        sec          ~
bitcmp       gammaln      sech         &&
bitor        hypot        sign         ||
bitshift     imag         sin
bitxor       Inf          single       Scalar expansion
ceil         int32        sinh         versions of the
complex      isfinite     sqrt         following:
conj         isinf        tan              *
cos          isnan        tanh             /
cosh         log          true             \
cot          log2         uint32           ^
coth

Branching
instructions:
     break
     continue
     else
     elseif
     for
     if
     return
     while
```

A la hora de realizar llamadas a *arrayfun* que se ejecuten en la GPU existen las siguientes limitaciones y restricciones:

- El argumento de la función *arrayfun* tiene que ser un puntero a una función MATLAB, cuyo fichero de función (no un script) define una única función.
- El código sólo puede llamar a una de las funciones soportadas indicadas en la lista anterior, y no puede llamar a scripts. La sobrecarga de las funciones soportadas no está permitida.
- El indexado (*subsagn*, *subsref*) no está soportado.
- Las siguientes características no están soportadas: variables persistentes o globales; *parfor*, *spmd*, *switch* y *try*/*catch*.

- Todos los cálculos en precisión doble se ajustan al estándar IEEE, pero debido a limitaciones del hardware, los cálculos en precisión simple no.
- Los únicos cambios de tipo de datos soportados son *single*, *double*, *int32*, *uint32* y *logical*.
- Las formas funcionales de operadores aritméticos no se soportan, pero los operadores simbólicos sí. Por ejemplo, la función no puede contener una llamada a *plus*, pero sí puede usar el operador *+*.
- Como *arrayfun* en MATLAB, la potencia exponencial de una matriz, multiplicación y división (^, *, /, \) sólo realizan cálculos elemento a elemento.
- No hay una variable *ans* para mostrar los resultados de cálculos sin asignación. Hay que asignar variables para tener resultados de todos los cálculos que se quieran obtener.

Como se ve, la función *arrayfun* se encuentra tiene una serie de limitaciones que no hay que pasar por alto, y es evidente que no es tan versátil como *gpuArray*. Un total de 141 funciones internas de MATLAB son compatibles con *gpuArray*, mientras que 109 lo son con *arrayfun*.

A la hora de crear datos directamente en la GPU existen varias posibilidades. Un número de métodos en la clase *GPUArray* permiten crear arrays en la GPU sin tener que transferirlos desde el espacio de trabajo de MATLAB. Estos constructores sólo requieren del tamaño del array e información acerca de la clase, para crear el array sin elementos provenientes del espacio de trabajo. Para crear arrays directamente en la GPU existen las siguientes opciones:

```
parallel.gpu.GPUArray.ones
parallel.gpu.GPUArray.zeros
parallel.gpu.GPUArray.inf
parallel.gpu.GPUArray.nan
parallel.gpu.GPUArray.linspace
parallel.gpu.GPUArray.eye
parallel.gpu.GPUArray.colon
parallel.gpu.GPUArray.true
parallel.gpu.GPUArray.false
parallel.gpu.GPUArray.losgpace
```

Para comprobar los métodos estáticos disponibles en cualquier versión de MATLAB, hay que escribir:

```
methods('parallel.gpu.GPUArray')
```

Por ejemplo, para crear una matriz identidad de 1.024 x 1.024 de tipo *int32* en la GPU:

```
G = parallel.gpu.GPUArray.eye(1024,'int32')

parallel.gpu.GPUArray:
----------------------
               Size: [1024 1024]
    ClassUnderlying: 'int32'
         Complexity: 'real'
```

Con un argumento numérico se crea una matriz de 2 dimensiones. Para crear una de 3 dimensiones de tipo *double*:

```
G = parallel.gpu.GPUArray.ones(100, 100, 50)

parallel.gpu.GPUArray:
----------------------
               Size: [100 100 50]
    ClassUnderlying: 'double'
         Complexity: 'real'
```

La clase por defecto para los datos es *double*, por lo que no hace falta especificarla.

Para crear un vector columna de 8.192 elementos de valor cero:

```
Z = parallel.gpu.GPUArray.zeros(8192, 1)

parallel.gpu.GPUArray:
----------------------
               Size: [8192 1]
    ClassUnderlying: 'double'
         Complexity: 'real'
```

Para un vector columna, el tamaño de la segunda dimensión es 1.

Con todo lo anterior, se tiene la suficiente información para pasar a la siguiente etapa de la metodología, estudiar el algoritmo e identificar los segmentos susceptibles de ser programados en la GPU.

5.1.1.2. Estudio del algoritmo e identificación de los segmentos susceptibles de ser programados en la GPU

Recuperando el código del algoritmo VCA mostrado anteriormente:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% VCA algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

indice = zeros(1,p);
A = zeros(p,p);
A(p,1) = 1;

for i=1:p
      w = rand(p,1);
      f = w - A*pinv(A)*w;
      f = f / sqrt(sum(f.^2));
      v = f'*y;
      [v_max indice(i)] = max(abs(v));
      A(:,i) = y(:,indice(i));        % same as x(:,indice(i))
end
Ae = Rp(:,indice);

return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% End of the vca function
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A la hora de estudiarlo conviene descomponer aquellas líneas de código donde se realizan varias operaciones para que sea más fácil analizarlo en detalle:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% VCA algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

...

for i=1:p
      w = rand(p,1);
      f = pinv(A);
      f = f*w;
      f = A*f;
      f = w - f;
```

```
        g = f.^2;
        g = sum(g);
        g = sqrt(f);
        f = f/g;
        v = f'*y;
        v = abs(v);
        [v_max indice(i)] = max(v);
        A(:,i) = y(:,indice(i));        % same as x(:,indice(i))
end

...

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% End of the vca function
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A la vista de lo anterior se pueden realizar las siguientes observaciones:

1. El bucle es realimentado, es decir, la iteración “n” depende de datos obtenidos en la iteración “n – 1”.
2. Se producen varias multiplicaciones de matrices, un total de 3.
3. Se realiza el cuadrado de una matriz.
4. Se realiza una normalización de una matriz.
5. La matriz “y” es constante.
6. Intervienen hasta un total de cinco vectores y matrices en el algoritmo.

Limitaciones, restricciones y detalles a tener en cuenta:

1. No se puede crear un array de números aleatorios en la GPU.
2. La función *pinv* de MATLAB no es compatible con la programación en la GPU porque hace uso de una función no soportada por la GPU (*svd* de MATLAB).
3. La forma en la que se realiza la llamada a la función *max* de MATLAB no es compatible con la programación en la GPU.
4. Hay que minimizar las transferencias de memoria a toda costa.

Dando por finalizado el estudio del algoritmo, se pasa a la próxima etapa, la programación del mismo en la GPU.

5.1.1.3. Programación del algoritmo en la GPU

Llegados a este punto se programa el algoritmo en la propia GPU. Como se ha visto en apartados anteriores, teniendo identificadas las funciones y las opciones más interesantes para ello, los cambios a realizar son mínimos. Todo va a girar en torno a:

1. *gpuArray*.
2. *gather*.
3. *parallel.gpu.CUDAKernel*.
4. *arrayfun*.
5. Las funciones sobrecargadas admitidas por la GPU.

Como se habrá deducido, *gpuArray* y *gather* son complementarias. En un caso se envían datos a la memoria de la GPU para que sean procesados allí, mientras que en el otro se traen de vuelta a memoria principal para que sean procesados por la CPU.

En el caso de *parallel.gpu.CUDAKernel* se realiza la llamada a una función creada en CUDA para que sea ejecutada desde MATLAB. Es una opción muy interesante ya que ofrece la posibilidad de explotar al máximo el potencial de la arquitectura GPU desde MATLAB. El único detalle que hay que considerar es que para ello se hace necesario:

1. Disponer de un entorno CUDA operativo.
2. Disponer de un fichero ".cu" (fichero CUDA) además de otro ".ptx" (fichero *Parallel Thread Execution*).
3. Tener el compilador correctamente configurado para poder generar el fichero ".ptx" a partir del fichero ".cu".

Teniendo todo en orden, generar el fichero ".ptx" a partir del ".cu" se consigue mediante la siguiente instrucción:

```
nvcc –ptx myfun.cu
```

Siendo "myfun.cu" el fichero del cual se quiere generar su contrapartida ".ptx". Como se comentó unas líneas más arriba, no hay que olvidar que el compilador tiene que estar correctamente configurado, ya que si no será imposible generar el fichero ".ptx".

Un detalle muy importante a tener en cuenta es que si no se dispone de los ficheros ".cu" y ".ptx" no será posible generar el kernel de CUDA desde MATLAB.

5.1.1.4. Modificaciones y mejoras

En esta etapa el objetivo es conseguir mejoras introduciendo modificaciones en el algoritmo original. Para ello algunas posibilidades son:

1. Agrupar o desagrupar operaciones. En algunos casos interesará tener operaciones agrupadas y realizarlas todas en la GPU, pero en otros podría ser interesante desagruparlas para ejecutar unos segmentos en la CPU y otros en la GPU al existir algún tipo de limitación y/o restricción.
2. Prescindir de operaciones. A veces determinadas operaciones no son útiles y se pueden omitir sin que el resultado obtenido se vea afectado por ello.
3. Modificar algunos tipos de operaciones para optimizar su ejecución. Por ejemplo, cambiar el cuadrado de una matriz por la multiplicación consigo misma.
4. Identificar qué operaciones es mejor realizar en la CPU y cuáles es mejor realizar en la GPU. El caso de la multiplicación de matrices es un claro ejemplo de operación que es mejor realizar en la GPU. Un bucle que se repite muchas veces y sin dependencia de datos en sus iteraciones es otro ejemplo en el cual se puede sacar provecho de la ejecución en GPU.
5. Realizar precargas de datos en la GPU.

En esta etapa la herramienta *profiler* de MATLAB es muy útil ya que sirve para ver de forma numérica y gráfica la ejecución de la función a estudiar. Basta con insertar unas líneas de código en los segmentos que se quieren monitorizar (*profile on*, *profile off* y *profile viewer*), y una vez ejecutado el algoritmo se despliega un menú con bastante información interesante. También existe la opción de abrir el propio *profiler* y ejecutar el

algoritmo desde allí, pero en casto se monitorizará todo el algoritmo, cuando, en ocasiones, podría interesar centrarse en una pequeña parte del mismo. En la figura 5.2. se muestra el menú resultante de una llamada al *profiler*.

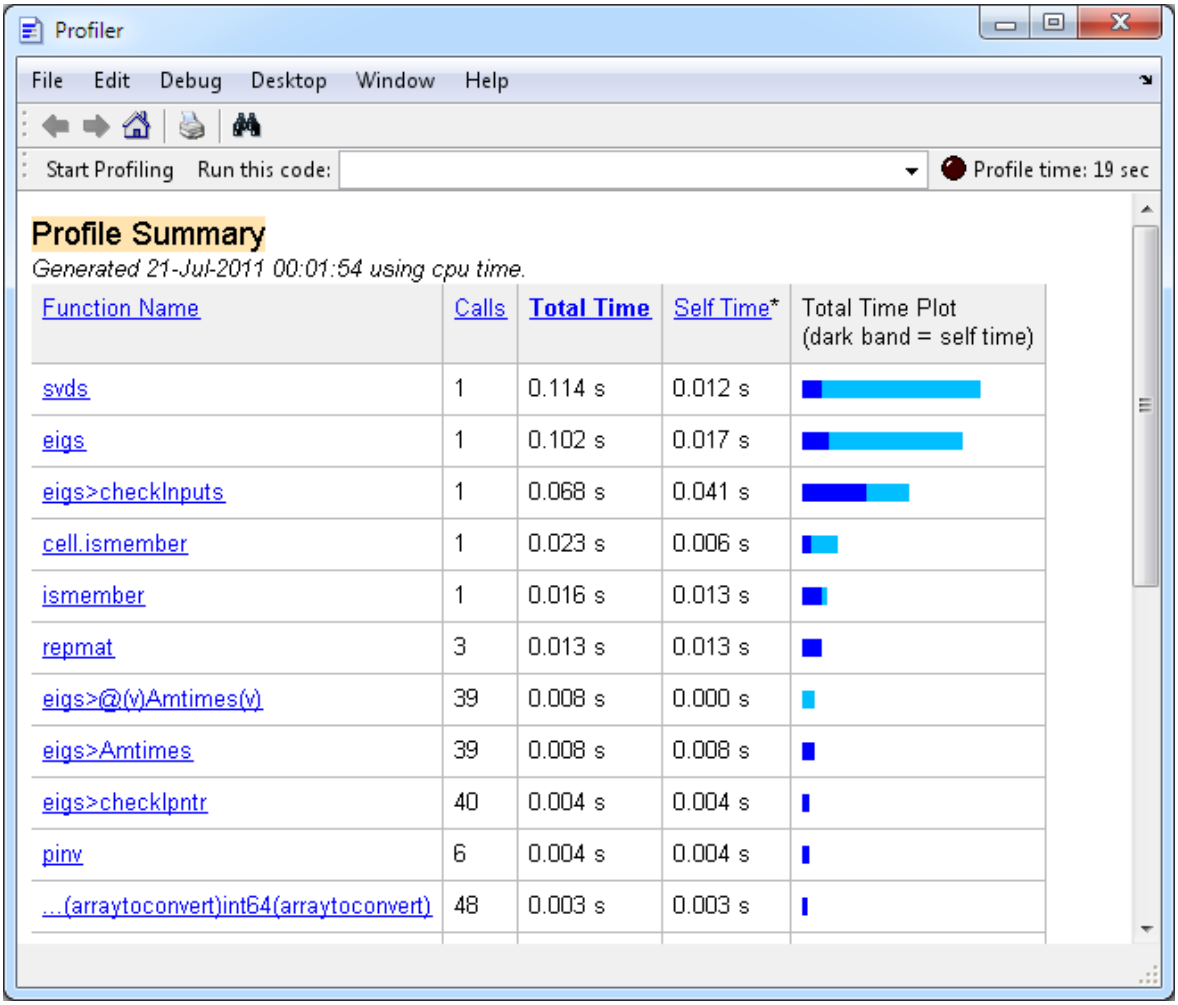

Figura 5.2. Menú del *profiler de* MATLAB.

En la figura 5.2. puede verse la información que devuelve el *profiler*, las funciones que intervienen en la ejecución del código, el número de llamadas que se realiza a cada una de ellas, medidas de tiempo, etc. Además, da la opción de explorar en detalle la ejecución de algunas funciones de forma individual.

Como se constata, es una herramienta muy interesante a la hora de intentar implementar mejoras en un algoritmo, ya que da una idea de en qué sitios podría sacarse más partido con posibles optimizaciones. De un vistazo rápidamente se pueden detectar aquellas

funciones que están consumiendo más tiempo de ejecución, aparte de que da la posibilidad de realizar análisis detallados.

5.1.1.5. Validación y pruebas

Una vez el algoritmo se ha programado en la GPU, se han detectado posibles mejoras y se han realizado las correspondientes modificaciones, es necesario hacer validaciones y pruebas. De esta forma se verifica que los resultados obtenidos con las modificaciones son similares o iguales que los obtenidos sin ellas. Bien es cierto que a veces puede existir la necesidad de buscar un compromiso entre mejoras en las prestaciones y los resultados, no siendo posible mejorar las prestaciones sin empeorar los resultados obtenidos. Ahora bien, de nada sirve introducir una serie de modificaciones si los resultados no son los esperados o no son coherentes.

En este caso, con el algoritmo VCA las validaciones y las pruebas se limitan a ejecutar las demostraciones que los propios autores facilitan y comprobar que todo sigue igual que sin haber realizado las modificaciones y mejoras.

5.1.1.6. Resultados

En esta última etapa del flujo de diseño simplemente se comparan los resultados obtenidos con aquellos que se toman como referencia. Si son mejores, se dan por buenos y se convierten en la nueva referencia. Si, por el contrario, no mejoran a los anteriores, se pasa de nuevo a la etapa de modificaciones y mejoras.

Este proceso se repite tantas veces como se crea oportuno, aunque llegará un punto donde conseguir nuevas mejoras será cada vez más complicado.

Capítulo 6. Resultados

En este capítulo se describen los resultados obtenidos con la metodología de alto nivel desarrollada.

6.1. Casos de test

Los desarrolladores del algoritmo *Vertex Component Analysis* facilitan, además del código MATLAB del propio algoritmo, tres demos para verificarlo:

- Demo 1: a partir de una imagen artificial se realiza la búsqueda de un total de 3 *endmembers*. La imagen se construye a partir de 3 firmas espectrales puras y mezclando los píxeles mediante una matriz facilitada por los propios autores.
- Demo 2: se mezclan 3 *endmembers*. En este caso la mezcla se realiza siguiendo una distribución aleatoria de Dirichlet.
- Demo 3: partiendo de una imagen real obtenida por el sensor AVIRIS, llamada Cuprite, se realiza la búsqueda de los *endmembers* deseados en la misma. Conviene indica que de esta imagen se conocen muy bien los materiales que existen en la escena, y por tanto, se puede comprobar si el método funciona.

Por la mayor complejidad de la tercera demo en cuanto a la búsqueda de *endmembers*, será la que se emplee para verificar el algoritmo con la metodología desarrollada.

6.2. Resultados obtenidos

En primer lugar, conviene realizar una serie de aclaraciones:

1. Se denominará “Intel i5” a aquella ejecución que se realice sólo en la CPU, sin pasar por la GPU.
2. Se denominará “GeForce” a aquella ejecución que se realiza parcialmente en la GPU, quedando el resto para la CPU. No hay que olvidar que el algoritmo no puede ejecutarse en su totalidad en la GPU porque existen una serie de funciones de MATLAB (*pinv* y *max*), que no se pueden programar en la misma.

En primer lugar, se midió el tiempo de ejecución del algoritmo original y de una modificación que prescindía de la normalización (operación que se reveló innecesaria en

todos los casos en los que se probó), siendo los resultados obtenidos en un caso y otro, idénticos. Los tiempos de ejecución se muestran en la figura 7.1.

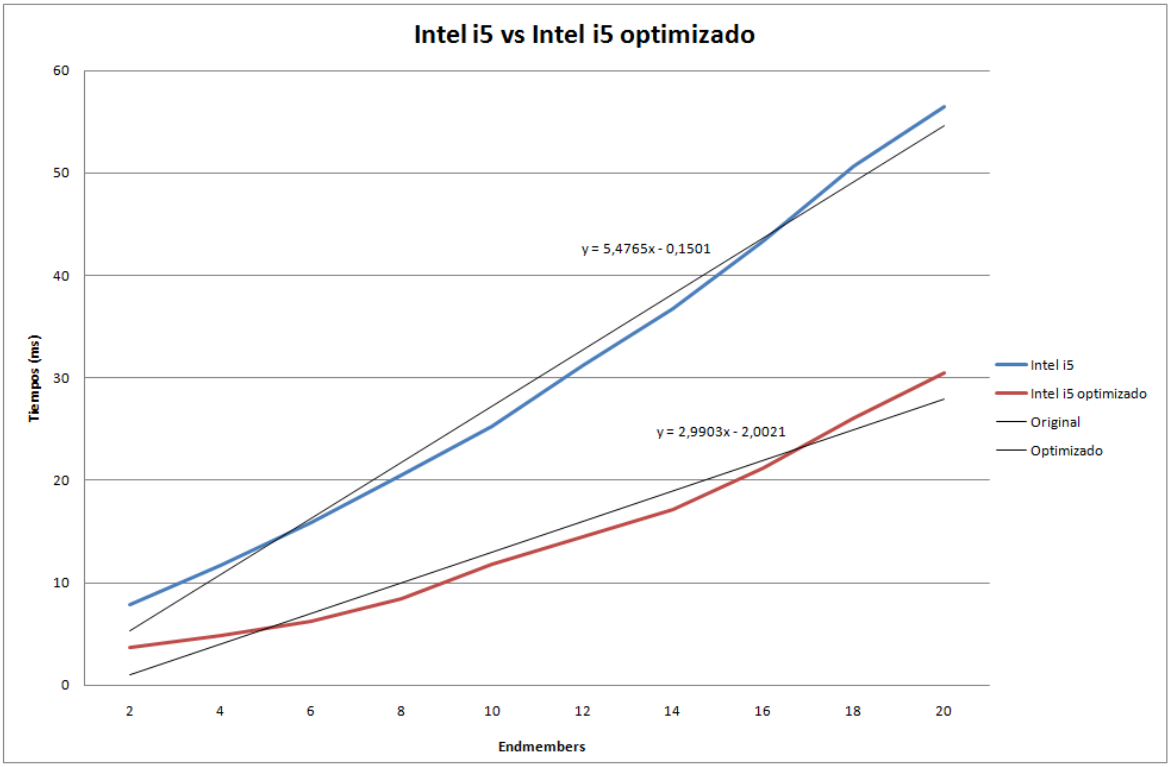

Figura 7.1. Gráfica con los tiempos de ejecución del algoritmo original y del optimizado.

Prescindir de la normalización produce un ahorro notable de tiempo. En el caso de 10 *endmembers* se acerca a los 15ms, mientras que para 20 sobrepasa los 25ms. También queda claro que, la pendiente en el caso del algoritmo optimizado, es bastante menor, siendo un 54,6% de la original. En la tabla 7.1. se ve el ahorro de tiempo que se produce.

Endmembers	***Intel i5***	***Intel i5 (optimizado)***	***Ahorro***
2	7,822 ms	3,677 ms	4,145 ms
4	11,636 ms	4,859 ms	6,777 ms
6	15,896 ms	6,293 ms	9,603 ms
8	20,533 ms	8,439 ms	12,094 ms
10	25,303 ms	11,750 ms	13,553 ms
12	31,162 ms	14,481 ms	16,681 ms
14	36,775 ms	17,179 ms	19,596 ms
16	43,374 ms	21,186 ms	22,188 ms
18	50,678 ms	26,071 ms	24,607 ms
20	56,528 ms	30,510 ms	26,018 ms

Tabla 7.1. Tiempos de ejecución en la CPU del algoritmo original y del optimizado.

Prescindiendo de la normalización, el tiempo de ejecución del algoritmo prácticamente se reduce a la mitad, un hecho que puede constatarse al ver que la magnitud de las cifras en la columna de "Ahorro" es prácticamente idéntica a la de aquellas en la columna "Intel i5 (optimizado)". Para confirmar que los resultados no cambian, se procede al cálculo de 3 *endmembers*, mostrándose las reflectancias obtenidas en la figura 7.2.

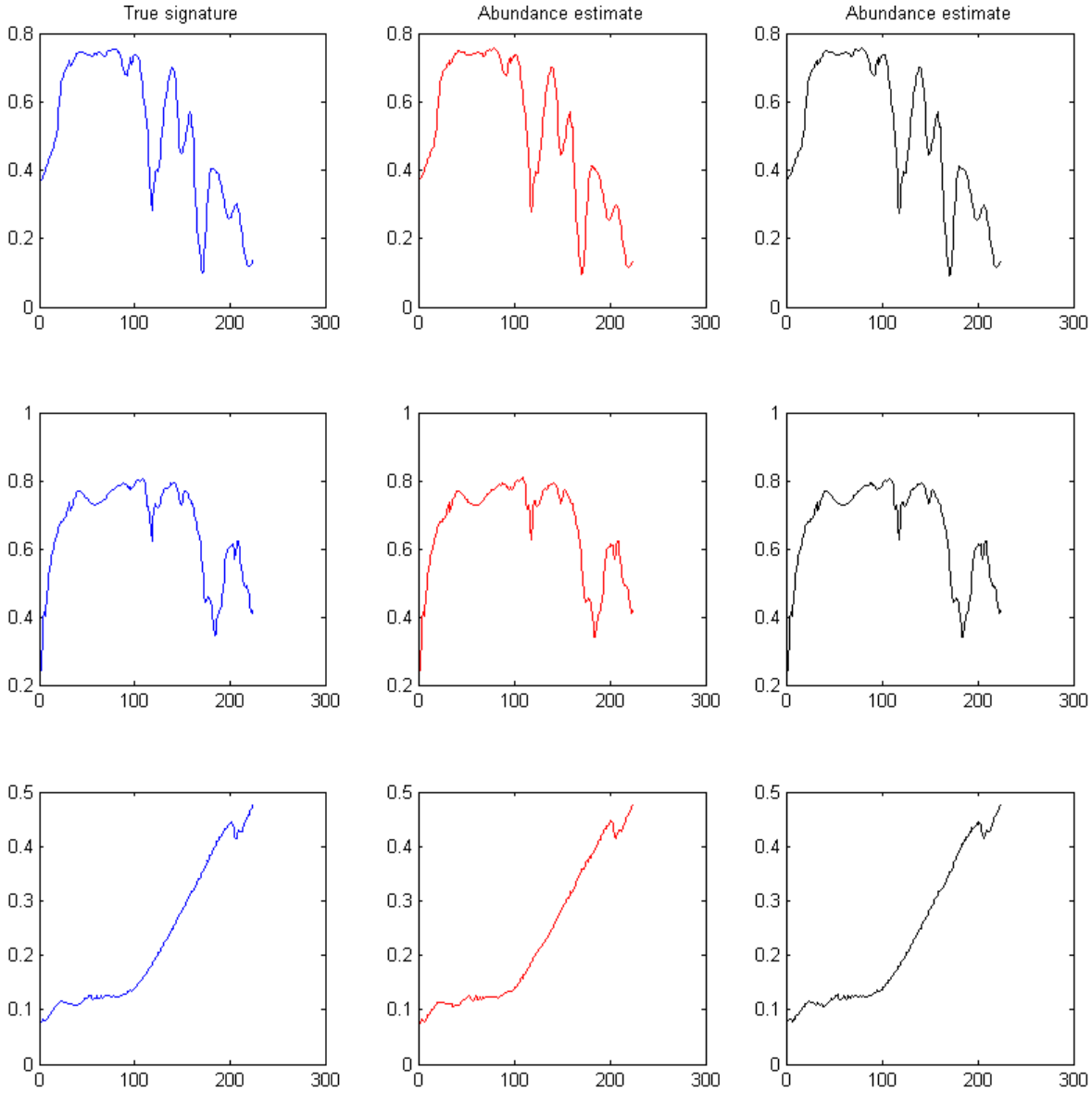

Figura 7.2. Algoritmo original (en rojo) y algoritmo modificado (en negro).

Como puede observarse, las curvas roja y negra son prácticamente idénticas. De esta forma se verifica que los resultados obtenidos mediante un algoritmo y otro son equivalentes.

De aquí en adelante se tomará como referencia el algoritmo optimizado, realizándose las subsiguientes medidas con éste y no con el original.

Antes de pasar a realizar las comparaciones entre el "Intel i5" y la "GeForce", resulta interesante desglosar los tiempos de ejecución:

$$\text{Intel i5}_{total} = \text{Transferencia} + \text{Intel i5}_{cálculo} \quad \text{(Ecuación 7.1.)}$$

$$\text{GeForce}_{total} = \text{Transferencia} + \text{GeForce}_{transferencia} + \text{GeForce}_{cálculo} \quad \text{(Ecuación 7.2.)}$$

Ojo, en los tiempos contemplados en "$\text{GeForce}_{cálculo}$", se incluyen tanto aquellos que se realizan en la propia "GeForce", como aquellos que obligatoriamente tienen que realizarse en el "Intel i5". Como se indicó anteriormente, no es posible ejecutar el algoritmo entero en la GPU, y, al medirse los tiempos de ejecución del algoritmo en su totalidad, los tiempos en el caso de la "GeForce" contemplan tanto los segmentos ejecutados en ella misma como aquellos que no queda más remedio que ejecutar en el "Intel i5".

Volviendo a las ecuaciones anteriores, puede observarse como en ambos casos tiene que producirse una transferencia de datos, denominada "Transferencia". Esta transferencia siempre corresponde a la carga en memoria de la imagen de Cuprite y a la inicialización de varias matrices, acción que se realiza una única vez, es independientemente del número de *endmembers* a buscar, por lo que las expresiones anteriores pueden simplificarse por:

$$\text{Intel i5}_{total} = \text{Intel i5}_{cálculo} \quad \text{(Ecuación 7.3.)}$$

$$\text{GeForce}_{total} = \text{GeForce}_{transferencia} + \text{GeForce}_{cálculo} \quad \text{(Ecuación 7.4.)}$$

Este último detalle queda reflejado en las figuras 7.3. y 7.4.

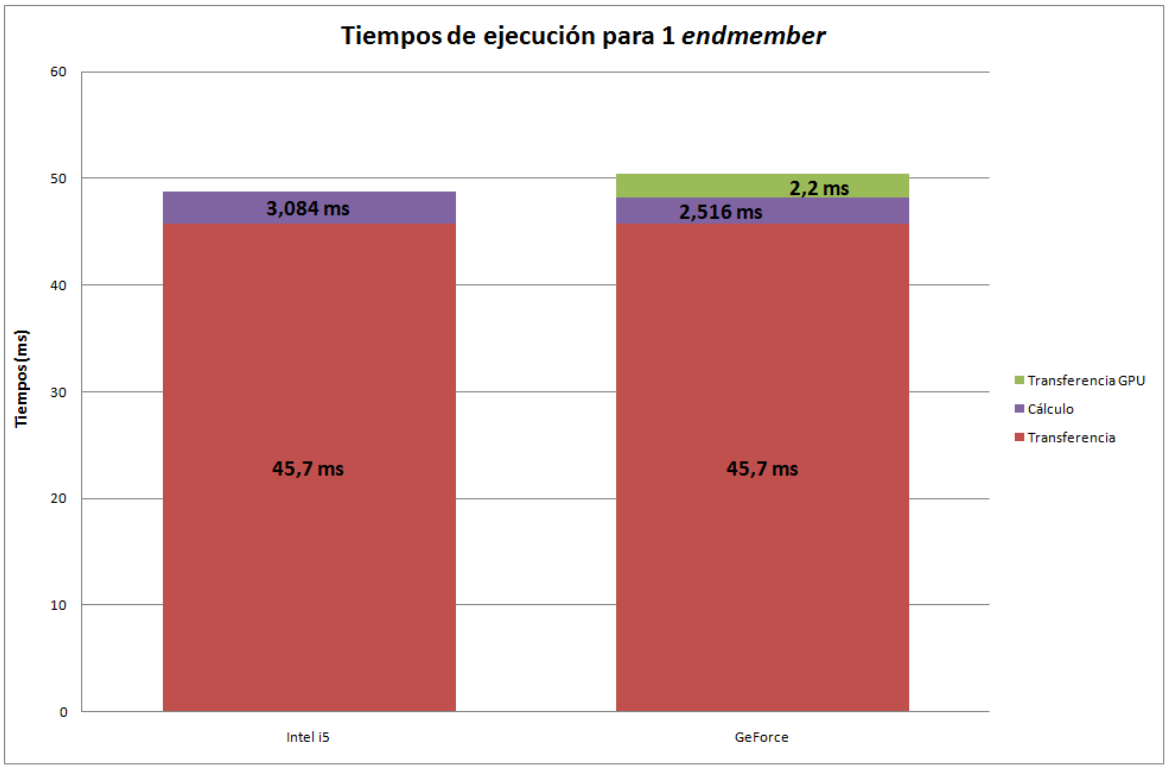

Figura 7.3. Tiempos de ejecución del algoritmo para 1 *endmember*.

Los tiempos de transferencia en ambos casos son idénticos, aunque la "GeForce" tiene una transferencia de datos adicional entre la memoria y ella misma, llamada "Transferencia GPU" en el gráfico. Sin embargo, los tiempos de cálculo son ligeramente inferiores en la "GeForce" comparados con el "Intel i5". Esto último es más evidente en la figura 7.4., donde se contemplan 20 *endmembers*.

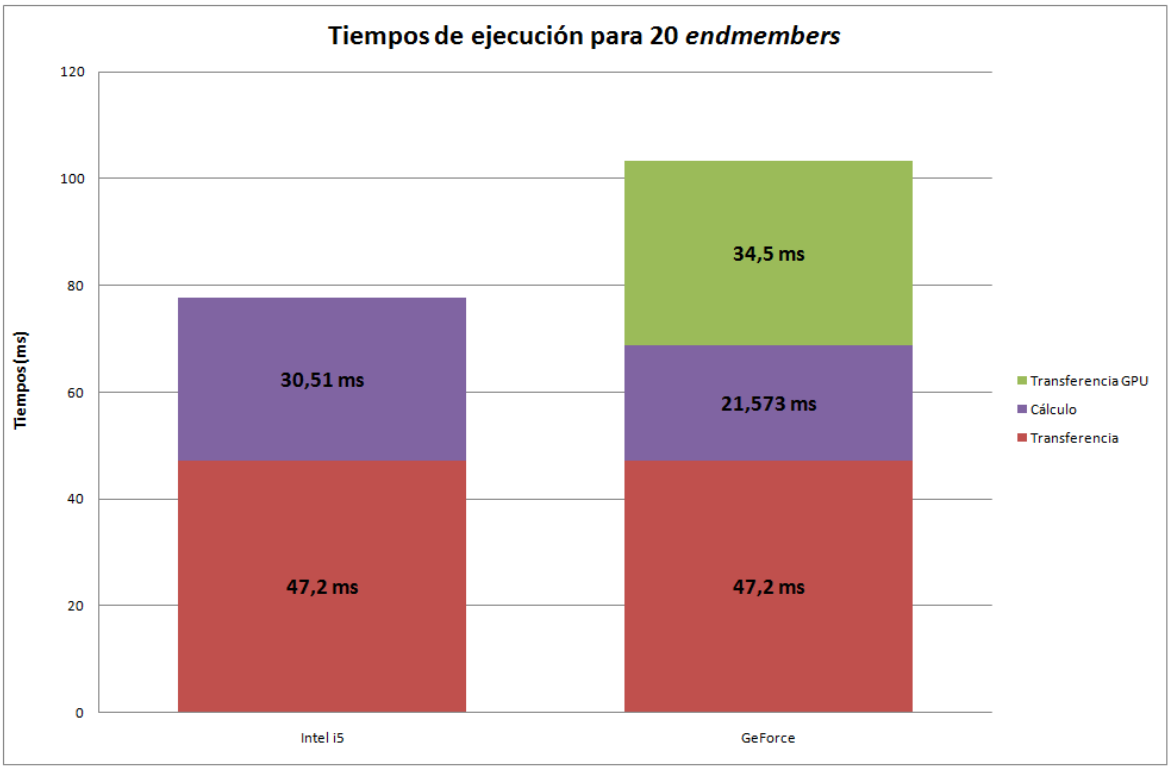

Figura 7.4. Tiempos de ejecución para 20 *endmembers*.

Para 20 *endmembers* los tiempos de cálculo muestran diferencias a tener en cuenta, aunque, simultáneamente, las transferencias necesarias entre memoria principal y la tarjeta gráfica también aumentan. Otra vez los tiempos de transferencia iniciales vuelven a coincidir, quedando demostrado que en ambos casos son idénticos.

A continuación se pasará a desglosar el tiempo de ejecución de los algoritmos. Como se vio en la ecuación 7.1., el tiempo total de ejecución en el "Intel i5" puede dividirse entre tiempo de transferencia y tiempo de cálculo, tal y como se muestra en la figura 7.5.

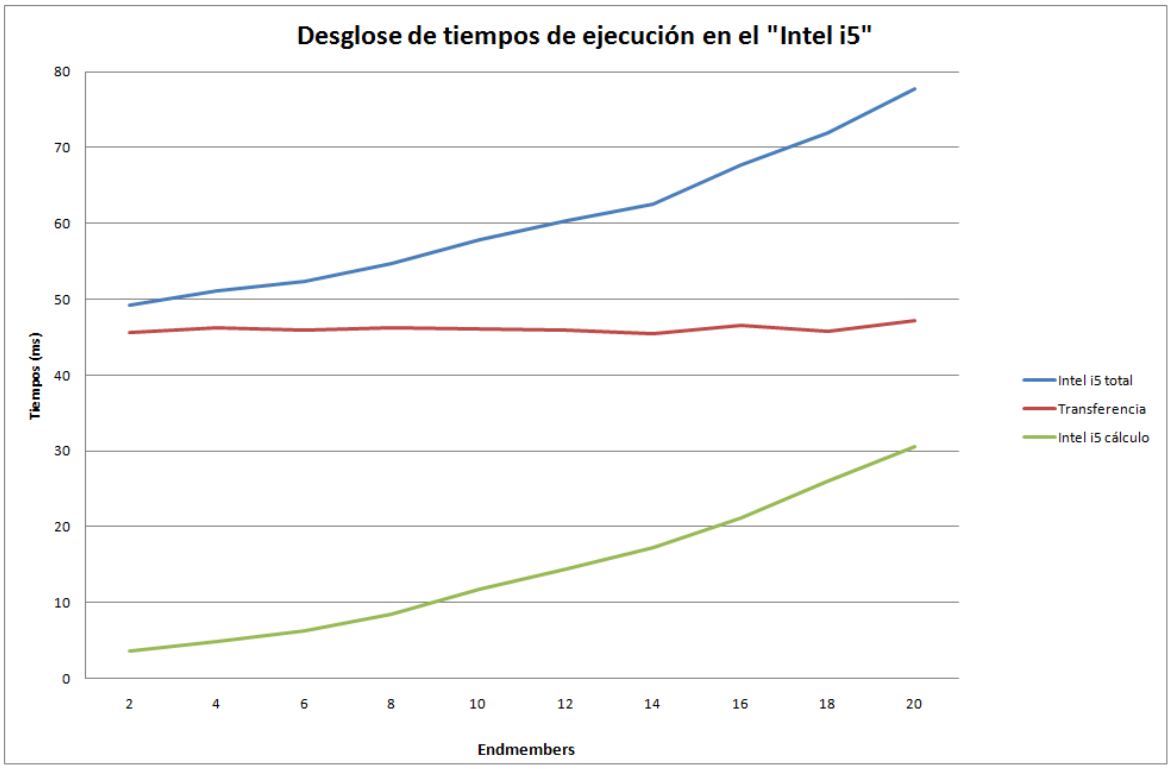

Figura 7.5. Desglose de tiempos de ejecución en el “Intel i5”.

En la figura se observa que el tiempo de transferencia permanece constante a medida que aumenta el número de *endmembers*, mientras que el tiempo de cálculo sí va creciendo. Los valores numéricos se muestran en la tabla 7.2.

Endmembers	***Intel i5 total***	***Transferencia***	***Intel i5 cálculo***
2	49,2770 ms	45,6000 ms	3,677 ms
4	51,0590 ms	46,2000 ms	4,859 ms
6	52,2930 ms	46,0000 ms	6,293 ms
8	54,6390 ms	46,2000 ms	8,439 ms
10	57,8500 ms	46,1000 ms	11,750 ms
12	60,3810 ms	45,9000 ms	14,481 ms
14	62,5790 ms	45,4000 ms	17,179 ms
16	67,7860 ms	46,6000 ms	21,186 ms
18	71,8710 ms	45,8000 ms	26,071 ms
20	77,7100 ms	47,2000 ms	30,510 ms

Tabla 7.2. Desglose de tiempos de ejecución en el “Intel i5”.

Cuando son pocos los *endmembers* a buscar, los tiempos de cálculo son inapreciables comparados con las transferencias. A medida que el número de *endmembers* va en aumento, el tiempo de cálculo crece en consonancia.

En la figura 7.6. se muestra el desglose de los tiempos de ejecución en la “GeForce”.

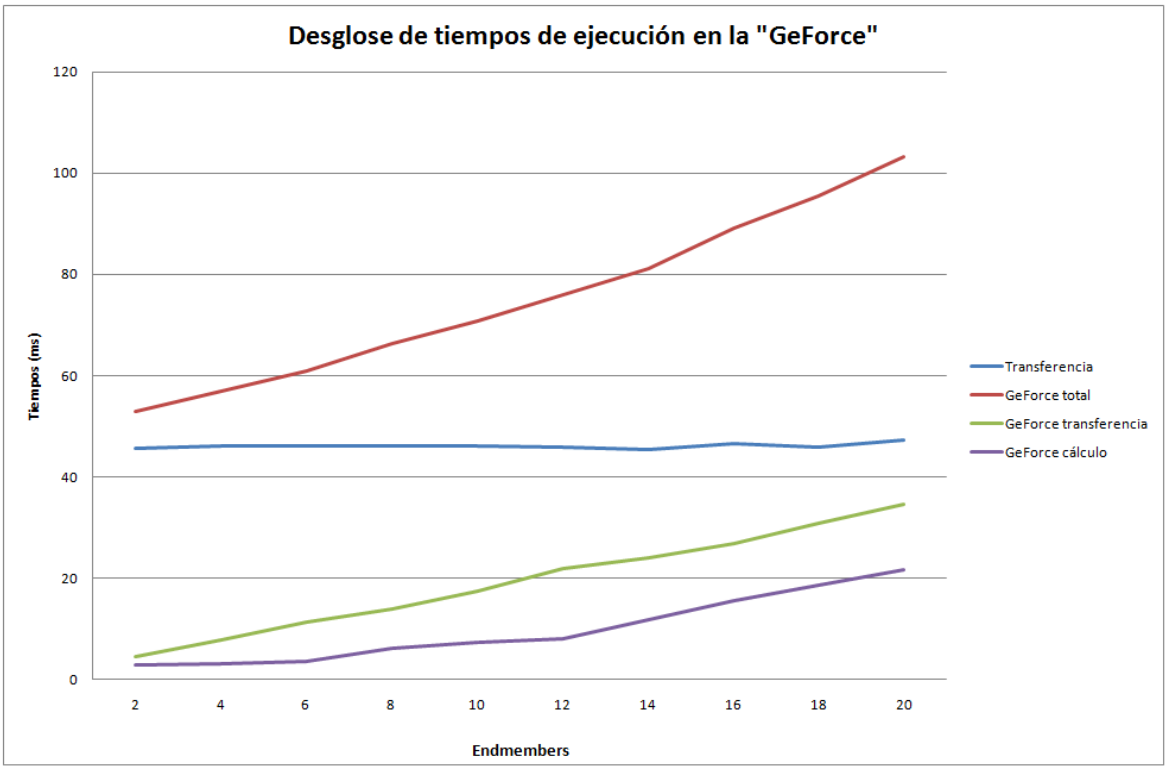

Figura 7.6. Desglose de tiempos de ejecución en la “GeForce”.

A diferencia del “Intel i5”, en este caso hay que añadir un tiempo de transferencia entre memoria principal y la de la “GeForce”, que no es constante, sino que va aumentando a medida que se buscan más *endmembers*. La explicación es que con cada iteración realizada es necesario enviar datos nuevos a la “GeForce”, y, en consecuencia, crece dicho tiempo de transferencia. Esto se debe a que el bucle del algoritmo es realimentado, dependiendo la iteración “n” de datos calculados en la iteración “n – 1”. Los valores numéricos quedan reflejados en la tabla 7.3.

Endmembers	*GPU total*	*Transferencia*	*GPU transferencia*	*GPU cálculo*
2	52,817 ms	45,6000 ms	4,400 ms	2,817 ms
4	56,889 ms	46,2000 ms	7,700 ms	2,989 ms
6	60,893 ms	46,0000 ms	11,300 ms	3,593 ms
8	66,301 ms	46,2000 ms	14,000 ms	6,101 ms
10	70,825 ms	46,1000 ms	17,500 ms	7,225 ms
12	75,936 ms	45,9000 ms	22,000 ms	8,036 ms
14	81,172 ms	45,4000 ms	24,000 ms	11,772 ms
16	89,117 ms	46,6000 ms	26,900 ms	15,617 ms
18	95,347 ms	45,8000 ms	30,900 ms	18,647 ms
20	103,273 ms	47,2000 ms	34,500 ms	21,573 ms

Tabla 7.3. Desglose de tiempos de ejecución en la “GeForce”.

Los tiempos de cálculo representan una pequeña fracción del total, llevándose la mayor parte de la ejecución las transferencias.

Lo verdaderamente interesante de realizar una implementación en una GPU (“GeForce”) es ver si es posible reducir los tiempos de cálculo si se comparan con aquellos obtenidos en la CPU (“Intel i5”). Dicha gráfica se muestra en la figura 7.7.

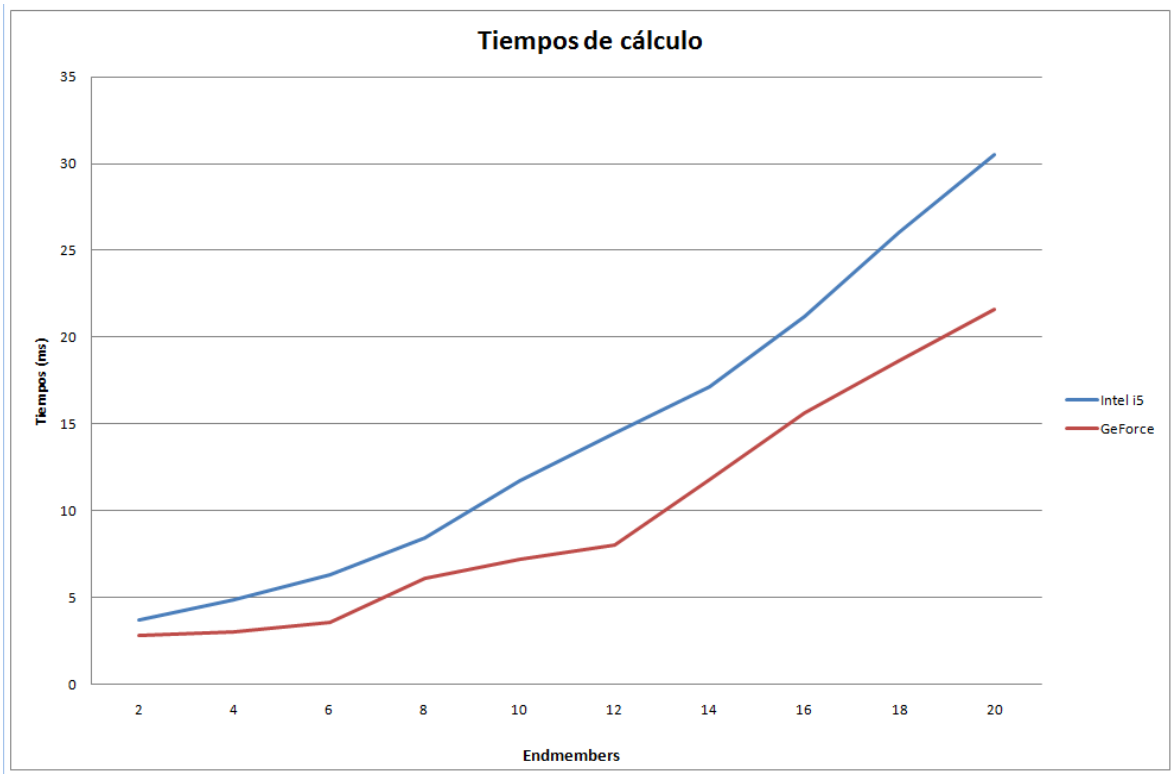

Figura 7.7. Tiempos de cálculo.

En la gráfica anterior resulta evidente que la "GeForce" gana con claridad al "Intel i5" en lo que a tiempos de cálculo se refiere; en todos los casos contemplados se alza como vencedora. Los valores numéricos se encuentran en la tabla 7.4.

Endmembers	***Intel i5***	***GeForce***	***Diferencia Intel i5 – GeForce***
2	3,677 ms	2,817 ms	0,860 ms
4	4,859 ms	2,989 ms	1,870 ms
6	6,293 ms	3,593 ms	2,700 ms
8	8,439 ms	6,101 ms	2,338 ms
10	11,750 ms	7,225 ms	4,525 ms
12	14,481 ms	8,036 ms	6,445 ms
14	17,179 ms	11,772 ms	5,407 ms
16	21,186 ms	15,617 ms	5,569 ms
18	26,071 ms	18,647 ms	7,424 ms
20	30,510 ms	21,573 ms	8,937 ms

Tabla 7.4. Tiempos de cálculo para el "Intel i5", la "GeForce" y diferencia entre ambos.

El ahorro entre el cálculo en el "Intel i5" y entre la "GeForce" ronda los 0,45ms por *endmember*, quedando patente la mayor potencia de cálculo que atesora esta última sobre el primero. Sin olvidar que no ha sido posible implementar el algoritmo en su totalidad en la "GeForce", sino una parte de él. Aún así, la GPU tiene unos tiempos de cálculo más reducidos que la CPU.

A la hora de realizar comparaciones, resulta muy interesante representar las pendientes para hacerse una idea de la bondad de un sistema en comparación con otro. En la figura 7.8. se muestran las pendientes de los tiempos totales de ejecución.

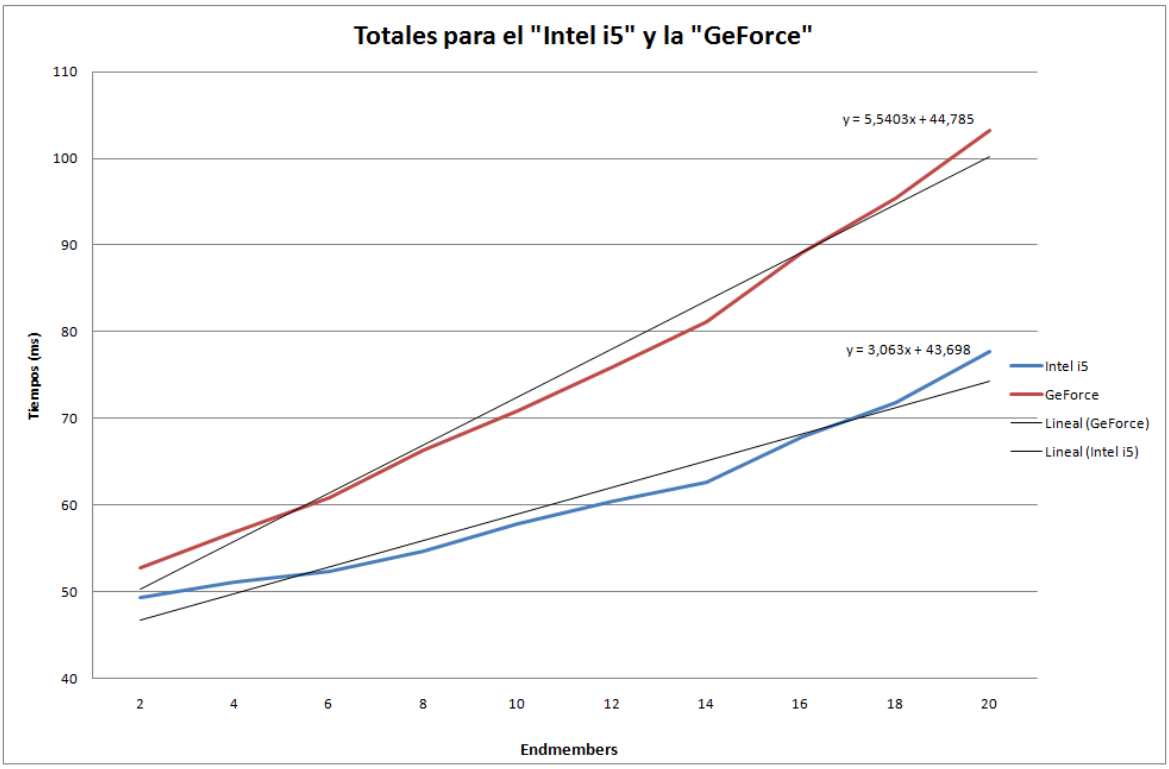

Figura 7.8. Tiempos totales de ejecución.

Como es de esperar, las pendientes para los tiempos totales de ejecución en el caso de la "GeForce" son mayores que las del "Intel i5". Como ya se ha comentado, la razón no es otra que las transferencias adicionales que hay que realizar, que aumentan en consonancia al número de iteraciones que se realizan en al algoritmo.

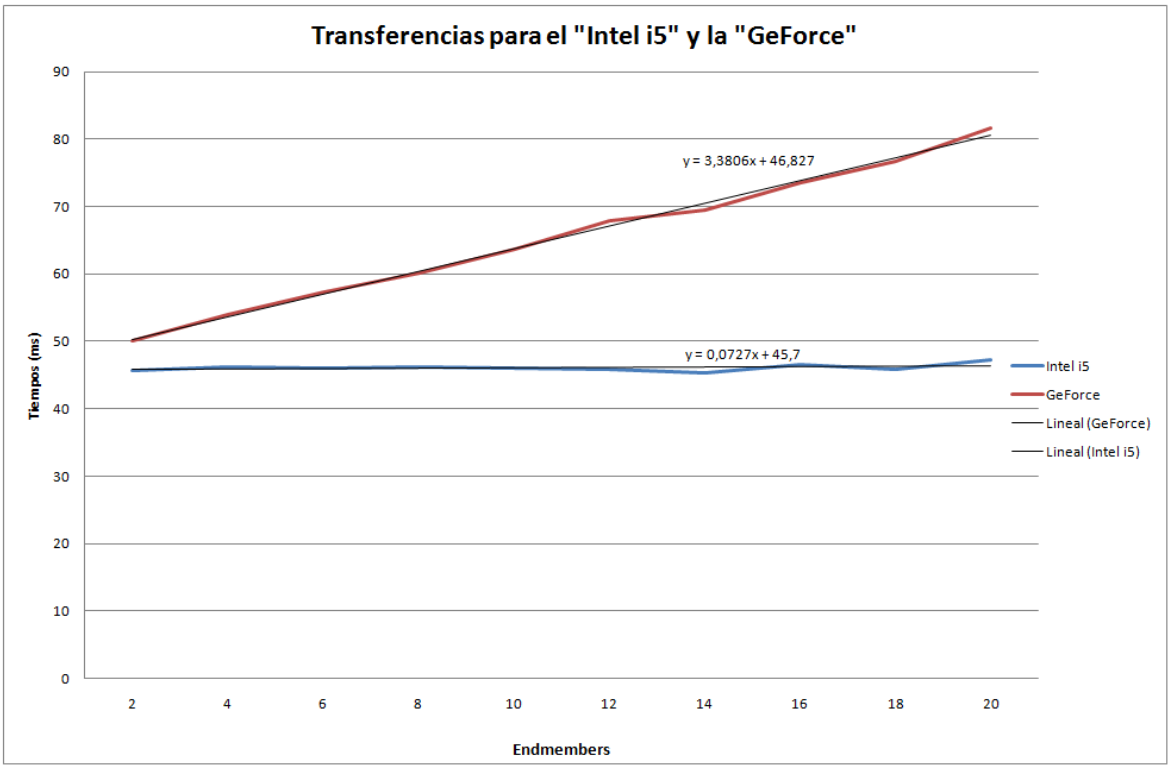

Figura 7.9. Tiempos de transferencia.

En lo referente a las transferencias, en el caso del “Intel i5” la pendiente es prácticamente nula, mientras que para la “GeForce” sí que existe una pendiente creciente, resultante, como se explicó previamente, de las transferencias adicionales en las que se incurre.

La progresión en los tiempos de cálculo es radicalmente distinta, tal y como se observa en la figura 7.10.

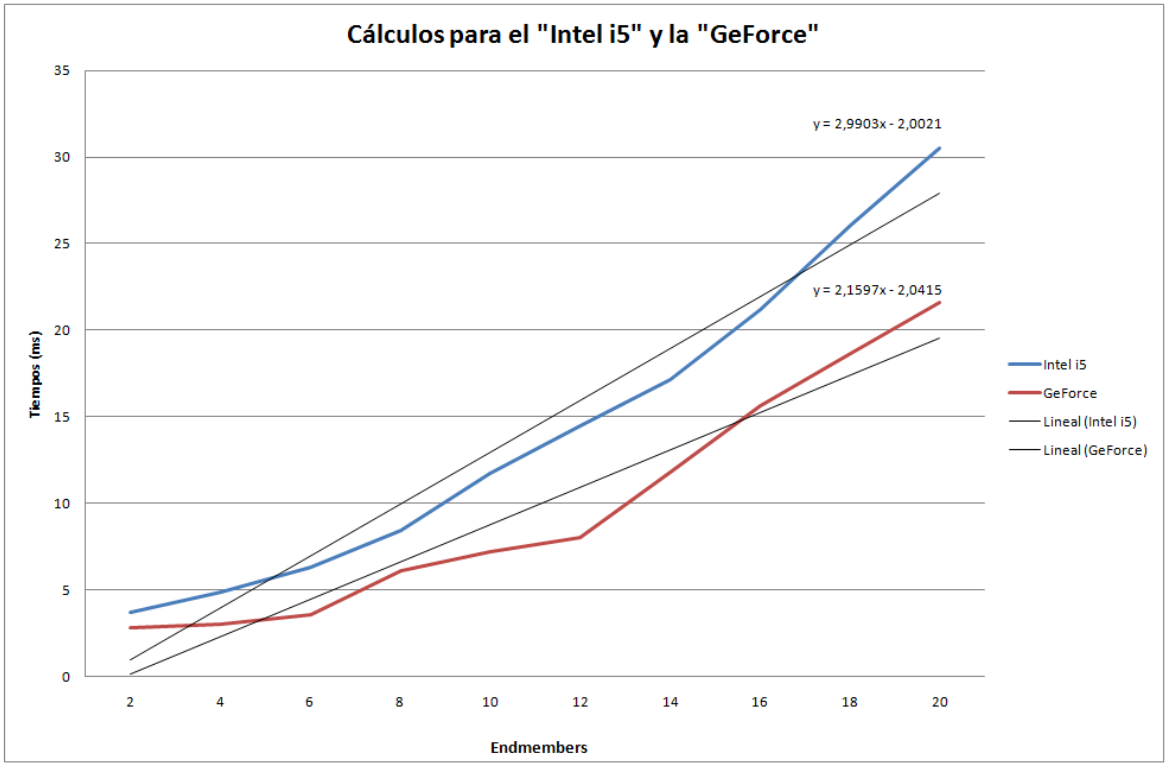

Figura 7.10. Tiempos de cálculo.

La pendiente para la “GeForce” es algo menor que la del “Intel i5”, un 72,2% de la original para ser exactos. Este dato viene a demostrar el potencial que ofrecen las GPU a la hora de cálculos computacionalmente intensivos.

A la vista de todos los resultados obtenidos, es más que obvio el potencial que ofrecen las GPU a la hora de reducir los tiempos de cálculo en comparación con las CPU. Un dato que también conviene considerar es que, en este caso concreto, la GPU empleada, una GeForce GTX 480, tiene una frecuencia de reloj 1,4 GHz en comparación con los 3,20 GHz de la CPU usada, una Intel i5 650.

Capítulo 7. Conclusiones y líneas futuras

En este capítulo se revisan los objetivos iniciales y se exponen las conclusiones alcanzadas tras la realización del Trabajo Fin de Máster, así como algunas de las dificultades más relevantes. También se proponen las posibles líneas futuras de trabajo.

7.1. Revisión de objetivos y conclusiones

La línea de trabajo principal de este proyecto ha estado orientada al desarrollo de una metodología de alto nivel para implementar el algoritmo *Vertex Component Analysis* en una GPU. En este último capítulo se quieren comentar los logros obtenidos, los problemas a los que ha habido que hacer frente y las posibles líneas futuras de investigación.

De cara a cumplir con los objetivos propuestos, se desarrolló dicha metodología de alto nivel para implementar el algoritmo en una GPU. Para ello se hizo uso de MATLAB y su *Parallel Computing Toolbox*. Se identificaron las funciones más interesantes disponibles para el propósito de este trabajo, así como las partes del código del algoritmo susceptibles a ser ejecutadas desde la propia GPU.

En la implementación en la GPU mediante MATLAB y su *Parallel Computing Toolbox* se detectaron una serie de limitaciones que causaron que la ejecución del algoritmo en la propia GPU no fuera tan rápido como en la CPU. No obstante, simultáneamente se comprobó como determinadas funciones sí se ejecutan de forma bastante más rápida en la GPU que en la CPU. El problema viene a la hora de las transferencias de memoria entre la CPU y la GPU, que suponen una penalización considerable en un caso como éste, más aún cuando ejecutar el algoritmo en la propia CPU supone tan poco tiempo.

Una vez logrados los objetivos principales, se sondearon alternativas para acelerar la ejecución del algoritmo en la GPU. La primera de ellas fue realizar una llamada a CUDA desde MATLAB para ejecutar las partes más propensas a ser paralelizadas. El problema encontrado en esta fase fueron las incompatibilidades entre Visual Studio 2008 y Visual Studio 2010 y CUDA Toolkit. Este problema a la hora de tratar de generar el fichero ".ptx" necesario para llamar a una función CUDA desde MATLAB, imposibilitó tratar de acelerar el algoritmo por esta vía. Hubo que generarlo en un entorno Linux instalado específicamente para la ocasión, y aún así no pudo ponerse a prueba porque no se disponía de una copia de MATLAB para dicho sistema operativo. El fichero ".ptx" generado en Linux no es compatible con la ejecución en Windows, y viceversa.

Una vez descartada la vía anterior, se buscó la alternativa de migrar el código desde el entorno MATLAB a un entorno C++, empleando para ello Embedded MATLAB. No hay que olvidar que desde C++, al igual que desde MATLAB, se pueden realizar llamadas a CUDA para optimizar la ejecución del algoritmo. Otra vez volvieron a presentarse problemas con los compiladores de Visual Studio 2008 y Visual Studio 2010. El código generado se compilaba y ejecutaba sin problemas en CodeBlocks, pero como CUDA se integra en Visual Studio, no quedó más remedio que tratar de pasarlo a este último, también sin éxito.

Debido a lo anterior, fue imposible buscar vías alternativas para mejorar los resultados en el tiempo contemplado para el desarrollo de este trabajo de fin de máster. Bien es cierto que gracias a las investigaciones llevadas a cabo, son obvias las ventajas que ofrecen las GPUs a la hora de reducir los tiempos de cálculo en tareas de cómputo intensivo, en comparación con las CPUs.

7.2. Líneas futuras de investigación

Los algoritmos para imágenes hiperespectrales tienen muchas aplicaciones y aún quedan varias vías por explorar de cara al futuro. El principal problema que tienen muchos de ellos es la gran carga computacional que presentan. Es por ello que a la hora de acelerar la ejecución de algoritmos de análisis hiperespectral la implementación en GPUs es una opción a considerar. Por este motivo el procesamiento en GPUs presenta, a priori, grandes ventajas.

Una propuesta interesante de continuación al trabajo que se ha realizado es una implementación pura en CUDA. De esta forma se aprovecharía al máximo todo el potencial que brinda la especial arquitectura de una GPU NVIDIA, ya que con MATLAB sólo se aprovecha una parte de ella. El procedimiento a seguir sería similar al descrito en esta memoria, realizando un estudio en detalle del algoritmo y buscando posibles modificaciones que permitan acelerarlo lo máximo posible.

Con una implementación en CUDA además se evitaría el principal contratiempo de la implementación realizada en este trabajo, las transferencias de memoria. Éstas son el principal caballo de batalla a la hora de tratar de ejecutar los algoritmos en una GPU. Con las implementaciones en MATLAB hay partes del código que no queda más remedio que ejecutar en la CPU, mientras otras se ejecutan en la GPU, con las consiguientes e inevitables transferencias de memoria. Un detalle que penaliza sobremanera la ejecución del mismo. Con CUDA se trataría de sortear esta importante limitación y sacar el máximo partido a la ejecución en la GPU.

En conclusión, la línea futura de investigación que se propone es realizar la implementación en una GPU mediante una programación pura en CUDA aprovechando al máximo la arquitectura y evitando pasar por MATLAB. Bien es cierto que para ello habría que configurar el entorno adecuadamente. La recomendación en este sentido es hacerlo en un sistema operativo Linux de 64 bits, en una máquina de 64 bits y en un entorno de desarrollo KDevelop y NVIDIA CUDA Toolkit de 64 bits; todo de 64 bits para evitar cualquier tipo de posibles incompatibilidades.

Referencias

La última consulta para las referencias web ha sido con fecha 17 de julio de 2011.

[1] J. Nascimento, J. Bioucas, "Vertex component analysis: a fast algorithm to unmix hyperspectral data", IEEE Transactions on Geoscience and Remote Sensing vol 43 nº 4 pp 898–910, 2005.

[2] NVIDIA GeForce GTX 480/470/465 GPU Datasheet (2010).

[3] NVIDIA CUDA Compute Unified Device Architecture – Programming Guide, Versión 1.1 (29/11/07)

[4] NVIDIA CUDA. Installation and Verification on Microsoft Windows XP and Windows Vista (C Edition). (August 2008 | DU–04165–001_v01)

[5] CUDA Technical Training. Volumen 1: Introduction to CUDA Programming (Prepared and Provided by NVIDIA. Q2 2008)

[6] CUDA Technical Training. Volumen 2: CUDA Case Studies (Prepared and Provided by NVIDIA. Q2 2008)

[7] Parallel Computing Toolbox™ 5 User's Guide, The MathWorks Inc. (2010).

[8] MATLAB CUDA
http://developer.nvidia.com/matlab–cuda

[9] Accelerating MATLAB with CUDA™ Using MEX files (NVIDIA Whitepaper) (septiembre 2007).

[10] Home page of José M. Bioucas Dias
http://www.lx.it.pt/~bioucas/

[11] J. C. Harsanyi, C. I. Chang, "Hyperspectral Image Classification and Dimensionality Reduction: An Orthogonal Subspace Projection Approach", IEEE Transactions on Geoscience and Remote Sensing vol 32 nº 4 pp 779–785, julio 1994.

[12] A. Plaza, P. Martínez, R. Pérez, J. Plaza, "A Quantitative and Comparative Analysis of Endmember Extraction Algorithms From Hyperspectral Data", IEEE Transactions on Geoscience and Remote Sensing vol 42 nº 3 pp 650–663, marzo 2004.

[13] C. A. Bateson and B. Curtiss, "A method for manual endmember selection and spectral unmixing," Remote Sens. Environ., vol. 55, pp. 229–243, 1996.

[14] J. W. Boardman, F. A. Kruse, and R. O. Green, "Mapping target signatures via partial unmixing of AVIRIS data," in Summaries of the VI JPL Airborne Earth Science Workshop. Pasadena, CA, 1995.

[15] M. E. Winter, "N–FINDR: An algorithm for fast autonomous spectral end–member determination in hyperspectral data," Proc. SPIE, vol. 3753, pp. 266–275, 1999.

[16] R. A. Neville, K. Staenz, T. Szeredi, J. Lefebvre, and P. Hauff, "Automatic endmember extraction from hyperspectral data for mineral exploration," in Proc. 21st Can. Symp. Remote Sensing, Ottawa, ON, Canada, 1999.

[17] J. Bowles, P. J. Palmadesso, J. A. Antoniades, M. M. Baumback, and L. J. Rickard, "Use of filter vectors in hyperspectral data analysis," Proc. SPIE, vol. 2553, pp. 148–157, 1995.

[18] A. Ifarraguerri and C. I. Chang, "Multispectral and hyperspectral image analysis with convex cones," IEEE Transactions on Geoscience and Remote Sensing vol. 37, pp. 756–770, Mar. 1999.

[19] A. Plaza, P. Martinez, R. Perez, and J. Plaza, "Spatial/spectral endmember extraction by multidimensional morphological operations," IEEE Transactions on Geoscience and Remote Sensing vol. 40, pp. 2025–2041, Sept. 2002.

[20] C. A. Bateson, G. P. Asner, and C. A. Wessman, "Endmember bundles: A new approach to incorporating endmember variability into spectral mixture analysis," IEEE Transactions on Geoscience and Remote Sensing vol. 38, pp. 1083–1094, Mar. 2000.

[21] S. M. Schweizer, J. M. F. Moura, "Efficient Detection in Hyperspectral Imagery", IEEE Transactions on Geoscience and Remote Sensing vol 10 nº 4 pp 584–597, abril 2004.

[22] S. Rosario–Torres, M. Vélez–Reyes, "Speeding up the MATLAB Hyperspectral Image Analysis Toolbox using GPUs and the Jacket Toolbox", en WHISPERS '09, 2009.

[23] V. Fresse, D. Houzet, C. Gravier, "GPU architecture evaluation for multispectral and hyperspectral image analysis", en Conference on Design and Architectures for Signal and Image Processing (DASIP 2010), 2010.

[24] A. Plaza, J. Plaza, H. Vegas, "Improving the Performance of Hyperspectral Image and Signal Processing Algorithms Using Parallel, Distributed and Specialized Hardware–Based Systems", Journal Signal Processing Systems, 2010.

[25] S. Sánchez, A. Plaza, "GPU Implementation of the Pixel Purity Index Algorithm for Hyperspectral Image Analysis", en IEEE Cluster Workshops 2010, 2010.

[26] J. Setoain, M. Prieto, C. Tenllado, F. Tirado, "GPU for Parallel On–board Hyperspectral Image Processing", International Journal of High Performance Computing Applications vol. 22 issue 4 pp. 424–437, noviembre 2008.

[27] S. Sanchez, G. Martin, A. Paz, A. Plaza, J. Plaza, "Near real–time endmember extraction from remotely sensed hyperspectral data using Nvidia GPUs", en SPIE Proceedings of Real–Time Image and Video Processing 2010, 2010.

[28] M. Garland, S. Le Grand, J. Nickolls et al., "Parallel Computing Experiences with CUDA", IEEE Computer Society, 2008.

[29] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, K. Schulten, "Accelerating molecular modeling applications with graphics processors", Journal of Computational Chemistry vol. 28 issue 16 pp 2618–2640, diciembre 2007.

[30] S. Ryoo, C. I. Rodrigues, S. S. Baghsorki, S. S. Stone, D. B. Kirk, W. M. W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA", en Proceedings of the 2008 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2008.

[31] S. Sengupta, M. Harris, Y. Zhang, J. D. Owens, "Scan Primitives for GPU Computing", en Proceedings Graphics Hardware 2007, 2007.

[32] T. Preis, P. Virnau, W. Paul, J. J. Schneider, "GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model, Journal of Computational Physics vol. 228 issue 12 pp 4468–4477, julio 2009.

[33] S. Che, M. Boyer, J. Y. Meng, D. Tarjan, J. W. Sheaffer, K. Skadron, "A performance study of general purpose applications on graphics processors using CUDA", Journal of Parallel and Distributed Computing vol. 68 issue 10 pp 1370–1380, octubre 2008.

[34] S. Samsi, V. Gadepally, A. Krishnamurthy, "MATLAB for Signal Processing on Multiprocessors and Multicores", IEEE Signal Processing Magazine, marzo 2010.

[35] Remote Sensing Tutorial – NASA
http://rst.gsfc.nasa.gov/

[36] C. I. Chang, Hyperspectral Imaging: Techniques for Spectral Detection and Classification, Kluwer Academic/Plenum Publishers, 2003.

[37] A. Plaza, C. I. Chang, Impact of Initialization on Design of Endmember Extraction Algorithms, IEEE Transactions on Geoscience and Remote Sensing, vol. 44, nº 11, pág. 3397–3407, 2006.

[38] Hyperion Instrument – NASA
http://eo1.gsfc.nasa.gov/Technology/Hyperion.html

[39] Earth–Observation 1 – NASA
http://eo1.gsfc.nasa.gov/

[40] ESA Earthnet: Compact High Resolution Camera
http://earth.esa.int/object/index.cfm?fobjectid=4216

[41] ESA Earthnet: PROBA
http://earth.esa.int/proba/

[42] R. O. Green, R.O., "Imaging spectroscopy and the airborne visible/infrared imaging spectrometer (AVIRIS)," Remote Sens. Environ., vol. 65, pp. 227–248, 1998.

[43] D. Landgrebe , "Hyperspectral Image Data Analysis", IEEE Signal Processing Magazine, vol. 19, no. 1, pp. 17–28, 2002.

[44] D. Landgrebe, "Multispectral Data Analysis, A Signal Theory Perspective" http://dynamo.ecn.purdue.edu/~biehl/MultiSpec/documentation.html, 1998.

[45] AVIRIS Home Page
http://aviris.jpl.nasa.gov/

[46] A. F. Goetz, B. Kindel, "Comparison of Unmixing Result Derived from AVIRIS, High and Low Resolution, and HYDICE images at Cuprite, NV", Proc.IX NASA/JPL Airborne Earth Science Workshop, 1999.

[47] J .Boardman, F. Kruse, R. Green, "Mapping target signatures via partial unmixing of AVIRIS data". Proc. Summaries JPL Airborne Earth Sci. Workshop, 23–26, 1995.

[48] R. O. Green, B. Pavri, "AVIRIS In–Flight Calibration Experiment, Sensitivity Analysis, and Intraflight Stability", en Proc. IX NASA/JPL Airborne Earth Science Workshop, Pasadena, CA, 2000.

[49] D. Heinz, C. I. Chang, "Fully constrained least squares linear mixture analysis for material quantification in hyperspectral imagery," IEEE Trans. on Geoscience and Remote Sensing, vol. 39, no. 3, pp. 529–545, March 2001.

[50] A. Plaza, D. Valencia, J. Plaza, C. I. Chang, "Parallel Implementation of Endmember Extraction Algorithms from Hyperspectral Data". IEEE Geoscience and Remote Sensing Letters, vol. 3, no. 3, pp. 334–338, July 2006.

[51] A. Plaza, J. Plaza, D. Valencia, "AMEEPAR: Parallel Morphological Algorithm for Hyperspectral Image Classification in Heterogeneous Networks of Workstations." Lecture Notes in Computer Science, vol. 3391, pp. 888–891, 2006.

[52] J. Setoain, M. Prieto, C. Tenllado, A. Plaza, F. Tirado, "Parallel Morphological Endmember Extraction Using Commodity Graphics Hardware," IEEE Geoscience and Remote Sensing Letters, vol. 43, no. 3, pp. 441–445, 2007.

[53] R. M. Pérez, P. Martinez, A. Plaza, P. L. Aguilar, "Systolic Array Methodology for a Neural Model to Solve the Mixture Problem", in: Neural Networks and Systolic Array Design. Edited by D. Zhang and S.K. Pal. World Scientific, 2002.

[54] CUDA GPUs
http://www.nvidia.com/object/cuda_gpus.html