UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE UPV INGENIEROS DE TELECOMUNICACIÓN

DEPARTAMENTO DE ELECTRÓNICA

Aceleración de algoritmos de Visión hiperespectral mediante GPU

Presentado por:

Jesús Pardal Garcés

Dirigido por:

Joaquín Cerdà Boluda

RESUMEN/ABSTRACT/RESUM

Resumen

Hoy en día la visión hiperespectral es una técnica en crecimiento y con un gran potencial para convertirse en uno de los métodos más importantes para el control de calidad de productos alimenticios. Es posible incorporar esta técnica en multitud de aplicaciones, como pueden ser para la detección de cuerpos extraños, la identificación de podredumbre en alimentos o el control de la calidad de los mismos.

Sin embargo, esta técnica presenta dos inconvenientes: la gran cantidad de datos que procesa un algoritmo hiperespectral y las altas tasas de producción que se requieren en un ámbito industrial. Esto hace que muchas veces una simple CPU sea incapaz de procesar todos estos cálculos hiperespectrales, preprocesado y predicción de las imágenes hiperespectrales, a una velocidad especificada. Por lo tanto, habrá que utilizar algún elemento adicional que ayude a la CPU a realizar todos estos cálculos de una manera que pueda cumplir las especificaciones requeridas respecto a de tiempos o velocidades.

El elemento que se ha utilizado en este proyecto final de carrera para ayudar a la CPU ha sido una Unidad de Procesado de Gráficos (GPU). Por lo tanto, a la GPU se le aplicará el concepto de GPUGPU, Computación de Propósito General en unidades de procesamiento gráfico, para acelerar algoritmos gracias a las ventajas que presenta la programación paralela.

Luego, en el presente proyecto final de carrera se analizará los factores de aceleración obtenidos utilizando este concepto, GPGPU, mediante diferentes GPUs aplicando algoritmos de clasificación en imágenes hiperespectrales.

Para aplicar estos algoritmos de clasificación en imágenes hiperespectrales se adquirieron imágenes de tres muestras de lomo de cerdo con cuerpos extraños mediante una cámara NIR. Una vez adquiridas las imágenes para las tres muestras en tres tomos, se aplicaron dos algoritmos de clasificación de imágenes hiperespectrales para cada

tomo en dos CPUs diferentes con sus respectivas GPUs, con el fin de comparar tiempos de ejecución en cada sistema.

En cuanto a las GPUs se seleccionaron dos dispositivos del fabricante NVIDA, concretamente, la GTX 260 y la GTX 560Ti. LA GTX 260 presenta ocho multiprocesadores, con un total de 192 núcleos, mientras que la GTX 560Ti presenta ocho multiprocesadores , con un total de 384 núcleos.

Se presenta primeramente una versión secuencial de los algoritmos hiperespectrales aplicados a cada CPU y a los que se les mide el tiempo de ejecución para su posterior comparación. A continuación, se introduce una versión paralela de los algoritmos hiperespectrales, aplicada sobre los dos dispositivos mencionados anteriormente y para mediciones de tiempo de ejecución con vistas a la comparación con la versión secuencial. Esta versión paralela fue desarrollada mediante CUDA, una extensión de C/C++ que permiten implementar el paralelismo en el procesamiento de tareas y datos con diferentes niveles de granularidad sobre GPUs de NVIDIA.

A los resultados anteriores se aplica el concepto de *speed up*, el cual indica la mejora de un algoritmo paralelo respecto a un algoritmo secuencial. Los *speeds up* obtenidos están en torno a un factor de 2.000, lo que indica que los algoritmos paralelos se ejecutan mucho más rápido y posibilita llevar esta técnica, visión hiperespectral, mediante esta tecnología, GPU, a un entorno industrial.

Abstract

The hyperspectral vision is nowadays a technology in growth and with a great potential to turn into one of the most important quality control methods of food products. It is possible to incorporate this technology in multitude of applications, such as detection of foreign bodies, identification of rottenness in food or its quality control.

Nevertheless, this technology presents two disadvantages: great quantity of information processed by hyperspectral algorithms and high rates of production needed in industrial areas. For these reasons, simple CPUs are often unable to process such hyperspectral calculations and process and predict hyperspectral images with a given speed. Therefore, it will be required to use additional elements to ensure that CPUs perform these calculations in such a way as to meet their time and speed specifications.

In this Final Project, the element used to ensure a proper CPU performance is a graphics processing unit (GPU). Therefore, the concept of GPUGPU (Computation of General Intention) will be applied to the GPU in units of graphical processing, in order to accelerate algorithms through the advantages of parallel programming.

Afterwards in the present Final Project, factors of acceleration obtained using the concept of GPGPU will be analyzed by means of different GPUs applying classification algorithms in hyperspectral images.

To apply these classification algorithms in hyperspectral images, images of three samples of pork loin with foreign bodies were obtained using a NIR camera. Once images of the samples were obtained in three packages, two classification algorithms of hyperspectral images were applied for each package in two different CPUs with their GPUs, in order to compare execution times in every system.

As for the GPUs, two devices of the manufacturer NVIDA were selected, in particular, GTX 260 and GTX 560Ti. GTX 260 features eight multiprocessors, with a total of 192 cores, while GTX 560Ti features eight multiprocessors, with a total of 384 cores.

First of all, a sequential version of the hyperspectral algorithms applied to each CPU is presented and their execution times are measured for later comparison. Subsequently, a parallel version of the hyperspectral algorithms is introduced. It is applied on both devices mentioned above to measure their execution times in order to compare them with those obtained through the sequential version. This parallel version was developed using CUDA, a C/C++'s extension which allows to implement parallelism in tasks and information processing with different levels of granularity on NVIDIA's GPUs.

The concept of *speed up* is applied to the results above. This concept is an indicator of parallel algorithms improvement in comparison to sequential algorithms. Indicators obtained are factors of about 2.000, which indicates that parallel algorithms run much faster than sequential ones. It makes it possible, therefore, to apply hyperspectral vision to industrial environments by means of this GPU technology.

Resum

Hui en dia la visió hiperespectral és una tècnica en creixement i amb un gran potencial per a convertir-se en un dels mètodes més importants per al control de qualitat de productes alimentaris. És possible incorporar esta tècnica en multitud d'aplicacions, com poden ser per a la detecció de cossos estranys, La identificació de podridura en aliments o el control de la qualitat dels mateixos.

No obstant això, esta tècnica presenta dos inconvenients: la gran quantitat de dades que processa un algoritme hiperespectral i les altes taxes de producció que es requerixen en un àmbit industrial. Açò fa que moltes vegades una simple CPU siga incapaç de processar tots estos càlculs hiperespectrals, preprocessat i predicció de les imatges hiperespectrals, a una velocitat especificada. Per tant, caldrà utilitzar algun element addicional que ajude a la CPU a realitzar tots estos càlculs d'una manera que puga complir les especificacions requerides respecte a de temps o velocitats.

L'element que s'ha utilitzat en este projecte final de carrera per a ajudar a la CPU ha sigut una Unitat de Processat de Gràfics (GPU). Per tant, a la GPU se li aplicarà el concepte de GPUGPU, Computació de Propòsit General en unitats de processament gràfic, per a accelerar algoritmes gràcies als avantatges que presenta la programació paral·lela.

Després, en el present projecte final de carrera s'analitzarà els factors d'acceleració obtinguts utilitzant este concepte, GPGPU, per mitjà de diferents GPUs aplicant algoritmes de classificació en imatges hiperespectrals.

Per a aplicar estos algoritmes de classificació en imatges hiperespectrals es van adquirir tres mostres de llom de porc amb cossos estranys per mitjà d'una cambra NIR. Una vegada adquirides les imatges per als tres toms, es van aplicar dos algoritmes de classificació d'imatges hiperespectrals per a cada tom en dos CPUs diferents amb les seues respectives GPUs, a fi de comparar temps d'execució en cada sistema.

Quant a les GPUs es van seleccionar dos dispositius del fabricant NVIDA, concretament, la GTX 260 i la GTX 560Tu. LA GTX 260 presenta huit multiprocessadors, amb un total de 192 nuclis, mentres que la GTX 560Tu presenta huit multiprocessadors, amb un total de 384 nuclis.

Es presenta primerament una versió seqüencial dels algoritmes hiperespectrals aplicats a cada CPU i als que se'ls mesura el temps d'execució per a la seua posterior comparació. A continuació, s'introduïx una versió paral·lela dels algoritmes hiperespectrals, aplicada sobre els dos dispositius mencionats anteriorment i per a mesuraments de temps d'execució amb vista a la comparació amb la versió seqüencial. Esta versió paral·lela va ser desenrotllada per mitjà de CUDA, una extensió de C/C++ Que permeten implementar el paral·lelisme en el processament de tasques i dades amb diferents nivells de granularidad sobre GPUs de NVIDIA.

Als resultats anteriors s'aplica el concepte de *speed up*, el qual indica la millora d'un Algoritme paral·lel respecte a un algoritme seqüencial. Els *speeds up* obtinguts estan entorn d'un factor de 2.000, la qual cosa indica que els algoritmes paral·lels s'executen molt més ràpid i possibilita portar esta tècnica, visió hiperespectral, per mitjà d'esta tecnologia, GPU, a un entorn industrial.

Agradecimientos

Quisiera dar las gracias a toda la gente que ha hecho posible que hoy este escribiendo estas líneas. Aunque he de reconocer que nunca sería suficiente el agradecimiento por todo lo recibido. Estas líneas son escritas con nostalgia pues soy consciente que dan lugar a la finalización de una gran etapa vivida. Gracias a todos los que os sintáis aludidos y no puedo nombrar aquí.

En primer lugar dar las gracias a mi director de proyecto Ximo, por haber confiado en mí para la realización del proyecto así como por todo el apoyo y confianza brindado durante este periodo, por toda su atención y dedicación, que no ha sido poca. Junto a él dar las gracias a Ainia Centro Tecnológico por confiar en la Universidad Politécnica de Valencia, concretamente, en el Instituto de Instrumentación para la Imagen Molecular para avanzar en sus proyectos de investigación y desarrollo. Tampoco me puedo olvidar de Néstor Ferrando que aporto sus conocimientos y me formo en el entorno de GPGPUs de modo altruista, sin pedir nada a cambio.

Gracias también a todos mis compañeros y amigos, sobre todo a una persona que ahora mismo está un poco más lejos de mí, que me han ayudado y acompañado en este duro camino haciéndolo más agradable y llevadero.

Por último los agradecimientos más especiales para mi familia de la cual siempre he recibido un apoyo incondicional, en particular a mis padres los cuales tienen todo el mérito de que haya llegado hasta aquí, gracias por todo vuestro sacrificio y por todo el apoyo recibido durante toda mi vida.

¡Mil gracias a todos!

Índice General

Índice de Figuras

Índice de Tablas

1. Introducción

Los consumidores de hoy en día se preocupan por ingerir productos de gran calidad, por lo que el sector de la seguridad alimentaria ha experimentado grandes cambios para poder cumplir las exigencias de los clientes. Para ello, se debe controlar el proceso de fabricación y la detección de cuerpos extraños en los productos alimenticios.

La detección de cuerpos extraños es unos de los principales problemas sin resolver que tiene actualmente el sector alimenticio. Habitualmente se realiza por diversas técnicas como pueden ser el detector de metales, rayos X y la visión por ordenador. Sin embargo, todos ellas tienen algunos inconvenientes que las hacen aplicables sólo en ciertos casos muy específicos; los detectores de metales sólo detectan los metales, los rayos X sólo pueden identificar los materiales extraños de mayor densidad que los ingredientes y la visión artificial sólo diferencia elementos que difieren en propiedades como la forma y el color.

¿Y qué ocurre con la detección de plásticos, fragmentos de plantas, cartones e insectos? A día de hoy solo hay estudios y técnicas emergentes.

Entre las técnicas emergentes susceptibles de ser aplicadas al sector de la alimentación para la detección de materiales procedentes del exterior está la visión hiperespectral. Esta técnica es particularmente interesante ya que combina la espectroscopia con el análisis de la imagen de una manera no invasiva. La espectroscopia infrarroja permite la identificación de compuestos en base a su composición química y el análisis de las imágenes posteriores permite la detección de fragmentos de materiales en puntos concretos con una composición diferente a la del producto objetivo con el fin de proceder a su expulsión.

Esta técnica presenta algunos inconvenientes, como el cuello de botella producido por tener que enviar en un instante determinado una gran cantidad de imágenes para que se procesen. Otra desventaja es el elevado tiempo de procesado de los algoritmos en el análisis de las imágenes para la detección de cuerpos extraños. Por

todo esto, esta técnica todavía está en desarrollo ya que su óptima utilización sería en un sistema en tiempo real y no como un post procesado como ocurre en la actualidad.

Por lo tanto, este trabajo tratará de abordar una solución a la problemática de la visión hiperespectral. Donde se procederá a estudiar las diferentes alternativas hardware que permitan acelerar algoritmos hiperespectrales con el fin de reducir tiempos de procesado para poder llevar a cabo un sistema en tiempo real en un ámbito industrial.

1.1 Motivación

El estudio de la aceleración hardware tiene un interés especial no solo en el campo del control de calidad, sino que además es aplicable a otros entornos como dinámica molecular y química computacional [1-2], investigación [2], medicina [3], industria del gas/petróleo [4], finanzas [5], es decir, en todos aquellos campos donde el tiempo de procesado sea crítico para la toma de decisiones en tiempo real. Como ejemplo se puede poner el caso de la toma de imágenes por ultrasonido a alta velocidad con un elevado nivel de detalle en un instante de tiempo reducido con la consecuente toma de decisiones. Otro ejemplo puede ser la obtención de imágenes de profundidad de la corteza terrestre que se utiliza para interpretar los datos sísmicos que determinan la búsqueda de nuevos yacimientos de petróleo.

El trabajo en el que esta embebido este proyecto, Cátedra AINIA, consta de una beca de colaboración en la cual participan un alumno de la E.T.S.I.I y otro de la E.T.S.I.T. Además, este proyecto tiene el aliciente de que con un buen trabajo realizado se afianzaran las relaciones establecidas entre la Universidad Politécnica de Valencia y la empresa AINIA para posibles acuerdos de colaboración que harán posibles que otros alumnos puedan desarrollar actividades formativas como proyectos final de carrera, becas predoctorales, cursos o conferencias en esta empresa.

1.2 Objetivos

El objetivo del presente Proyecto Final de Carrera es el de profundizar en las posibles alternativas para la aceleración eficiente de los algoritmos de los procesados hiperespectrales que realiza AINIA, para más tarde transferir dicha tecnología a la industria donde se procesará los datos adquiridos en tiempo real. Los algoritmos implementados para el procesado de las imágenes hiperespectrales fueron desarrollados por la empresa. El primer objetivo será analizar y estudiar las arquitecturas de cada algoritmo para posteriormente realizar una aceleración eficiente.

El siguiente paso a realizar será la implementación en *Matlab*. Este lenguaje tiene como principales características ser un lenguaje sencillo, contener abundantes herramientas gráficas, contar con una biblioteca matemática amplia y manejar matrices con facilidad. Por esta razón, se utilizará en un primer paso ya que dará facilidades a la hora de obtener unos primeros resultados tanto de representación gráfica como de obtención de datos.

Una vez realizada esta acción, el siguiente paso será llevarlo al lenguaje *C/C++* ya que *MatLab* no genera código de ejecución y su tiempo de ejecución es lento; por lo tanto, no es apto para sistemas en tiempo real que necesitan muchos recursos del sistema como memoria y *CPU*. Además, los comandos de *MatLab* están implementados de forma transparente de cara al programador por lo que no se sabe cuán de eficientes serán estos comandos. Por todo esto, se trasladará a un lenguaje donde se controle todos los pasos a realizar, que sea eficiente, que sea rápido para poder transportarlo a un sistema en tiempo real y a la vez sencillo. Por ello, se elegirá *C++*, ya que se trata de un lenguaje muy utilizado para la realización de software de sistemas en tiempo real y que cumple con lo descrito anteriormente.

Por otra parte, implementarlo en el lenguaje *C++* ayudará a comprender completamente los algoritmos realizados por ANIA, apoyándose en el lenguaje anterior para ir comprobando paso a paso los valores que se van obteniendo en *C++* y así tener certeza de que todos los pasos dados son correctos. Igualmente, permitirá poder comparar tiempos una vez se haya acelerado los algoritmos.

Para lograr el objetivo de acelerar los algoritmos, se estudiará detalladamente las posibilidades tecnológicas que hay actualmente en el mercado. A continuación, se elegirá la alternativa que mejor se adapte y ofrezca mejor aceleración y facilidad de implementación en un entorno industrial, para su posterior puesta a punto en un sistema en tiempo real. Este es el objetivo principal que se marcará siendo el resultado final una atractiva e innovadora solución para el mercado, un factor muy a tener en cuenta en la actualidad.

1.3 Metodología

La resolución de los objetivos propuestos para este PFC ha requerido el uso de una metodología de trabajo que abarca distintas tareas:

- Recopilación de información.
- Análisis de los algoritmos.
- Implementación de los algoritmos en MatLab y C++.
- Estudio de alternativas hardware para la aceleración.
- Elección de la alternativa y estudio de optimización.
- Implementación de la alternativa.
- Verificación, análisis de resultados y comparación de tiempos.

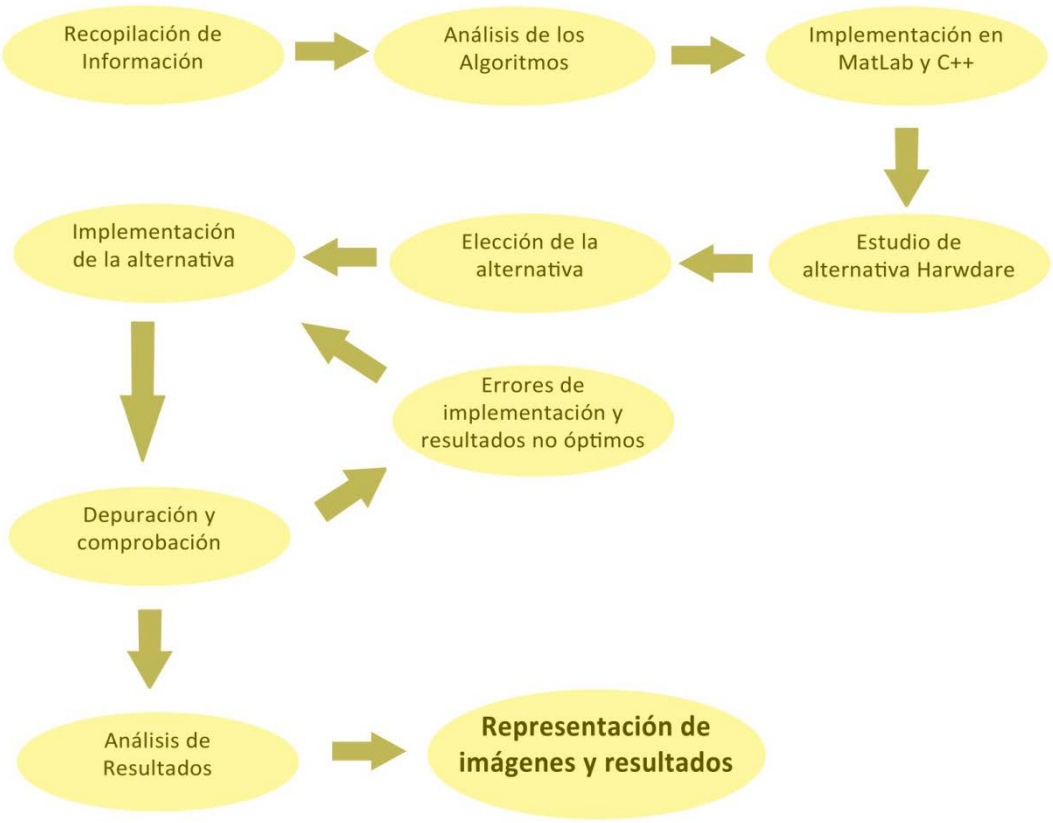


Figura 1. Metodología de trabajo empleada a lo largo del desarrollo del PFC.

La Figura 1. Metodología de trabajo empleada a lo largo del desarrollo del PFC. muestra un esquema de la metodología seguida en el desarrollo del PFC. Tras una etapa inicial de obtención de información bibliográfica con el objetivo de comprender las técnicas usadas, se realizará el análisis de los algoritmos para su total comprensión y así poder empezar a implementarlos. Después de la etapa de implementación se analizará las alternativas, previo rastreo en el mercado de todas las posibles tecnologías que permitan acelerar dichos algoritmos. Tras esto, se elegirá aquella que permita una óptima aceleración y una implementación en un entorno industrial tan sencilla como sea posible. A continuación, se realizará una versión inicial y se entrará en un proceso iterativo notablemente práctico, donde se alternará la realización de optimizaciones y modificaciones para obtener un rendimiento óptimo. Para concluir, se comparará los tiempos empleados en C++ con los tiempos de la alternativa elegida y así poder obtener *speed ups*, que reflejarán la mejora respecto al algoritmo sin acelerar.

1.3.1 Herramientas Utilizadas

A lo largo del proyecto se utilizarán distintas herramientas según en la etapa en la que se esté.

En la etapa de recopilación de información se recurrirá fundamentalmente al portal www.google.es, dada su versatilidad y su potencia. Por otra parte, en el transcurso del desarrollo del PFC se irán utilizando las siguientes herramientas:

- MatLab: es un lenguaje de programación sencillo y de un alto nivel de abstracción, es decir, que cuando se utiliza un comando no se conoce su implementación interna y por lo tanto se desconoce su tiempo de ejecución. Además posee una gran biblioteca de funciones matemáticas que facilitan la representación y el tratamiento de los datos. Igualmente, *MatLab* permite operar con matrices de forma fácil y natural. Por ello, en un primer lugar se ha utilizado esta herramienta para una primera implementación secuencial y así poder estudiar correctamente el funcionamiento de los algoritmos. Gracias a esto, permitirá realizar comprobaciones de los pasos efectuados en las distintas implementaciones a realizar.

- Microsoft Visual C++ 2008 Express Edition: Es el entorno de desarrollo integrado por Microsoft para sistemas operativos Windows. Soporta varios lenguajes de programación tales como Visual C++, Visual C#, Visual J#, y Visual Basic .NET, al igual que entornos de desarrollo web como ASP.NET. Se ha decantado por esta versión ya que es gratuita. En cuanto al lenguaje de programación se ha decido realizarlo en C++ ya que es uno de los más comunes y el más utilizado para realizar software de sistemas y aplicaciones. Además combina la flexibilidad y el acceso de bajo nivel de C con las características de la programación orientada a objetos como abstracción, encapsulación y ocultación, permitiéndome realizar un código eficiente. Asimismo el empleo de C++ permite la integración de bibliotecas ya existentes, al encontrarse muchas de ellas implementadas en dicho lenguaje de programación, con lo que ayudará

en buena medida. Como puede ser el caso de la biblioteca OpenCV que realizará la parte de lectura y representación de las imágenes.

- OpenCV: es una biblioteca libre de visión artificial originalmente desarrollada por Intel. Desde que apareció se ha utilizado en infinidad de aplicaciones (contiene más de 2500 funciones que abarcan distintas aéreas), desde sistemas de seguridad con detección de movimiento, hasta aplicaciones de control de procesos donde se requiere reconocimiento de objetos. Su publicación bajo licencia BSD permite que sea usada libremente para propósitos comerciales y de investigación, siendo la más descargada de su área, con más de dos millones de descargas.

  OpenCV está desarrollada en varios lenguajes de programación, y además es multiplataforma, ya que posibilita su utilización en Linux, Mac OS X, Android y Windows. Actualmente la librería sigue recibiendo aportaciones, por lo que tiene un continuo crecimiento y mejora en sus códigos. En el inicio del proyecto se empezó utilizando la versión OpenCV 2.1 y debido a las actualizaciones lanzadas por la plataforma se modificaron las versiones hasta llegar a la versión 2.4.0.

- Nvidia CUDA SDK v4.1: Usado junto con Microsoft Visual C++ para el desarrollo del código destinado a la alternativa elegida, proporciona las librerías y herramientas necesarias para la compilación, verificación y ejecución del código generado. Al igual que la librería OpenCV, la versión del SDK de CUDA en su inicio fue la 4.1 y durante el transcurso del proyecto se llegó a la versión 5.0.

- Nvidia CUDA Visual Profiler: esta herramienta se ha usado en la etapa de optimización del código. Ya que dicho software es capaz de mostrar gráficamente una gran cantidad de detalles del proceso de ejecución de los algoritmos sobre la alternativa elegida, como pueden ser divergencia de los hilos, patrones de acceso poco eficientes o el alto/bajo número de instrucciones

por ciclo. Todas estas variables ayudará a optimizar el código y verificar el resultado. Si las soluciones obtenidas no son satisfactorias a alguno de los niveles, será necesario volver atrás y rediseñar el algoritmo y a continuación realizar la misma tarea de optimización para ver si ha dado sus frutos.

1.3.2 Planificación

En este subapartado se estimará el tiempo que se necesitará para implementar cada uno de las distintas fases.

En la primera etapa, la de recopilación, el tiempo ha utilizar será alrededor de una semana. A lo largo de esta semana se reunirá información y se procederá a su lectura para su comprensión.

En la fase de análisis e implementación se estipulará entorno a dos meses. Durante estos dos meses se desarrollará el código tanto en *MatLab* como en C++, obteniendo la representación gráfica de los tres tomos de imágenes hiperespectrales correspondientes a las tres muestras cárnicas y sus respectivos tiempos de ejecución.

En la siguiente etapa se estudiará las alternativas existentes en el mercado que mejor se adapte para la aceleración hardware. Tras seleccionar la alternativa se procederá a su estudio, incluyendo unas jornadas de formación impartidas por Néstor Ferrando Jódar. Esta parte se desarrollará en torno a un mes de trabajo. Una vez estudiada la alternativa se seguirá con el desarrollo del código con dicha elección. En esta fase se desarrollará un código inicial y se entrará en un proceso iterativo donde se verificará la eficiencia del código. Si el código resulta que no es lo suficientemente optimo se procederá a su modificación y a continuación, se seguirá verificando su optimización hasta llegar a un punto donde no se pueda optimizar más. Esta parte se estipulará en torno a dos meses.

La última fase corresponderá con la verificación y el análisis de los resultados obtenidos, es decir, se tratará de comparar los tiempos de ejecución en cada unos de los

distintos lenguajes utilizados. Una vez comparados se sacarán conclusiones y se propondrán posibles mejoras. En esta parte se necesitará alrededor de dos semanas.

1.4 Organización del Proyecto

Una vez descrito la introducción a la cual pertenece el presente apartado, donde se exponen motivación, objetivos y metodología, herramientas utilizadas y su planificación. A continuación, se explicará cómo se ha estructurado el resto del proyecto.

En el capítulo dos se describirán métodos matemáticos que se utilizarán en los algoritmos desarrollados por AINIA para su mejor entendimiento y comprensión, como pueden ser qué es la imagen hiperespectral, Centrado y Escalado, Análisis de Principales Componentes (PCA), Análisis Discriminante (AD) y la primera derivada de Savitky-Golay.

Posteriormente, en el tercero capítulo se expondrán las distintas alternativas que hay actualmente en el mercado para la aceleración hardware. A continuación, se describirán de forma breve sus ventajas e inconvenientes. Tras esto, se dará una introducción de los orígenes y fundamentos a la arquitectura elegida, aportando una visión general de sus características y de su modelo e interfaz de programación. A su vez se explicarán las estrategias generales de optimización necesarias para conseguir un buen rendimiento y se introducirán los diferentes subsistemas que la conforman.

Después viene el capítulo cuarto que es la parte fundamental de este proyecto, ya que en él se explicarán los pasos seguidos para la implementación. Por una parte de los algoritmos de un modo secuencial y por otra parte la paralelización de los algoritmos mediante la alternativa elegida.

En el quinto capítulo se presentará los resultados obtenidos tras la finalización del proyecto en forma de gráficos y tablas, incluyendo los *speeds up* conseguidos en los distintos algoritmos.

A continuación, en el sexto capítulo se expresarán las conclusiones extraídas a lo largo del proyecto y posibles líneas futuras que se puedan tomar para mejorar la optimización o aplicarlas a otros ámbitos.

Finalmente, el documento se cerrará con las referencias bibliográficas utilizadas y anexos donde se detalla los códigos utilizados en las implementaciones de los distintos lenguajes de programación empleados para desarrollar los algoritmos.

2. Algoritmo de clasificación de imágenes hiperespectrales

A lo largo de este apartado se definirá conceptos matemáticos necesarios para comprender los algoritmos de clasificación de imágenes hiperespectrales. A continuación, se describirán brevemente los apartados que forman este capítulo:

En el capítulo uno se explicará el concepto de imagen hiperespectral, comentando las características de este tipo de imágenes.

En el siguiente capítulo, el segundo, se realizará una breve descripción de la empresa AINIA y de la visión hiperespectral, donde se explicará la visión hiperespectral y las cámaras que se utilizan y de cómo funcionan.

En el tercer capítulo se describirá dos preprocesados, centrado y escalado y la primera derivada de Savitzky-Golay, para poder tratar las imágenes hiperespectrales de una forma más adecuada.

A continuación vendrá el cuarto capítulo, donde se explicará una técnica, Análisis de Componentes Principales (PCA), como herramienta de análisis exploratorio de datos y para realizar modelos predictivos.

Después viene el quinto capítulo, donde se expondrá una técnica estadística multivariante cuya finalidad es describir las diferencias entre grupos de objetos sobre los que se observan. Este método se denomina Análisis Discriminante.

Y por último, el sexto capítulo se explicará los dos algoritmos utilizados para la clasificación de imágenes hiperespectrales.

2.1 Concepto de Imagen Hiperespectral

La observación de un determinado objeto está basada en la captación, por parte de un sensor, de la radiación electromagnética procedente de la interacción entre el objeto y la fuente de radiación. La radiación electromagnética recibe varios nombres dependiendo de la longitud de onda que la caracteriza, como puede apreciarse en la

Figura 2. Espectro electromagnético.2. Para medir la radiación emitida o reflejada por una determinada superficie es preciso cuantificar la cantidad de flujo energético que procede de la misma. Para ello se utiliza la medida de la radiancia, que depende de factores como la percepción de brillo, reflectancia, ángulos de observación, entre otros.

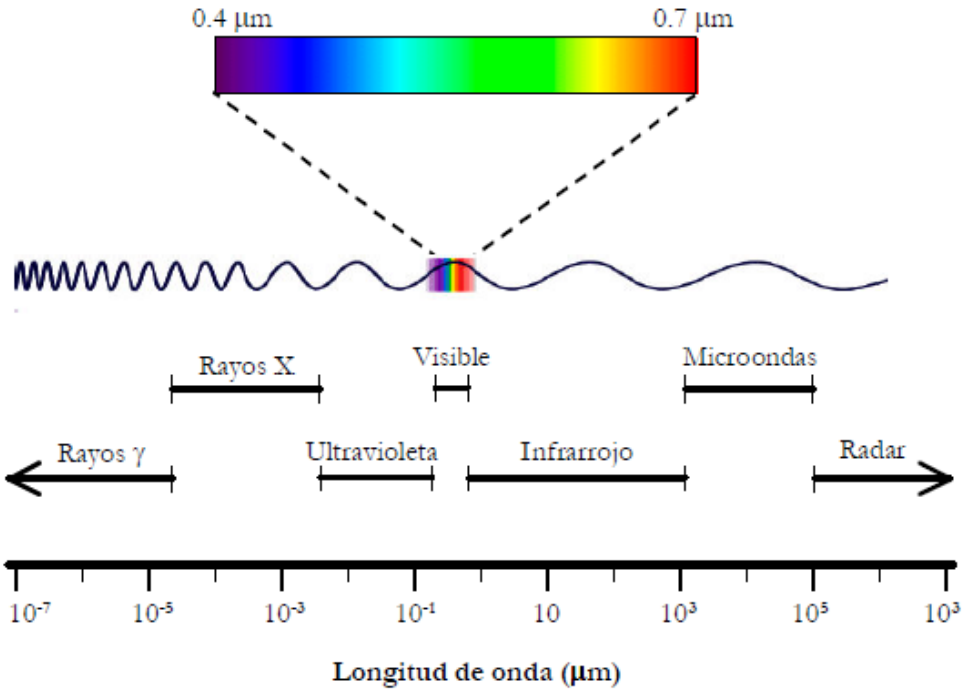


Figura 2. Espectro electromagnético.

En la actualidad, existe un gran abanico de sensores capaces de medir diferentes longitudes de onda a lo largo de una aérea espacial. Gracias al alto rango que cubren estas herramientas es posible la redefinición del concepto de imagen a través de la idea de pixel. Así, en una imagen en escala de grises, se puede decir que un pixel está constituido por un único valor discreto, mientras que, en una imagen hiperespectral, un pixel consta de un conjunto de valores. Estos valores pueden ser entendidos como

vectores N-dimensionales, siendo N el número de bandas espectrales en las que el sensor mide información. El resultado de la toma de datos, vectores N-dimensionales, puede ser representada de 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. En la Figura 3. Ejemplo de imagen multidimensional de cuatro longitudes de onda.3 se muestra la estructura de una imagen hiperespectral. En este caso el orden de magnitud de N permite realizar una distinción a la hora de hablar de imágenes multidimensionales. Así, cuando el valor de N es reducido, típicamente unas cuantas bandas espectrales, se habla de imágenes multiespectrales, mientras que, cuando el orden de magnitud de N es de cientos de bandas, se habla de imágenes hiperespectrales.

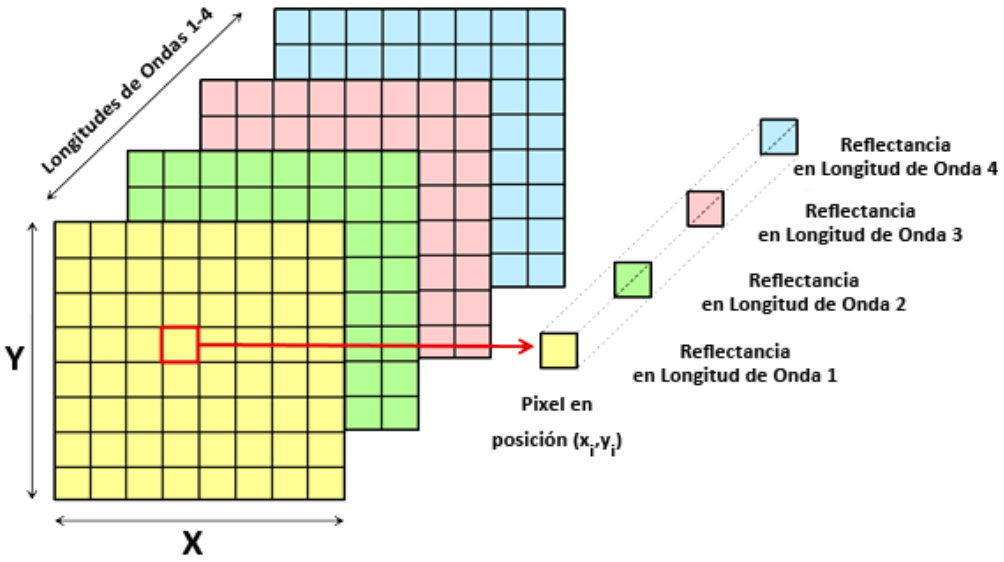


Figura 3. Ejemplo de imagen multidimensional de cuatro longitudes de onda.

2.2 Visión hiperespectral

AINIA es una empresa del sector agroalimentario situada en el Parque Tecnológico de Paterna que se dedica a la investigación, desarrollo tecnológico e innovación para empresas en alimentación, química, farmacia y cosmética.

En 1994 empezó a investigar en el campo de la visión artificial aplicada al control de calidad de los alimentos.

La visión artificial es la ciencia que aborda la adquisición de imágenes, sin contacto y mediante sistemas ópticos, donde se procederá al análisis de las imágenes para controlar un proceso o resolver tareas de inspección y manipulación. A lo largo de esto últimos años se han ido implantando en la industria con el fin de automatizar tareas de inspección, control de calidad y guiado de robots.

Un sistema de visión artificial está formado por la fuente de iluminación, la cámara que transforma la realidad 3D en una representación en 2 dimensiones y el procesador con el software de análisis que procesa las imágenes, extrae la información de interés y toma las decisiones oportunas.

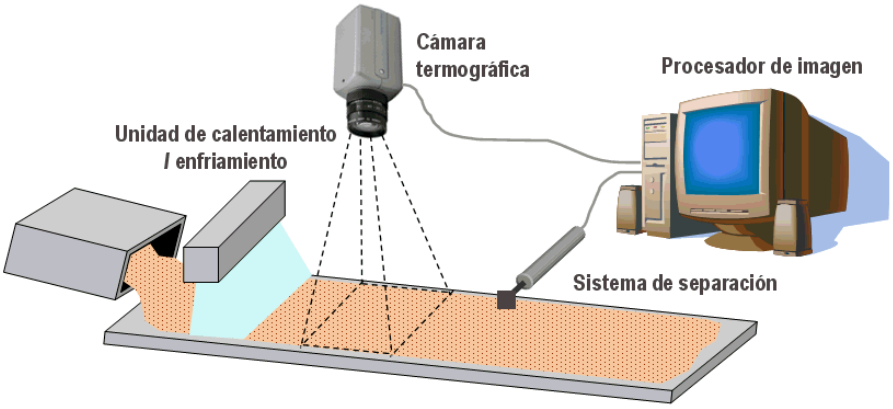


Figura 4. Ejemplo de un sistema de visión artificial

Esta técnica permite resolver diferentes tipos de problemas relacionados con la detección de colores con altísima precisión, medición de ángulos, lecturas de caracteres impresos de cualquier tipo, detección de presencia o ausencia de componentes, control

de calidad del producto obtenido avisando sobre cualquier defecto repetitivo o grave y detección de defectos entre otros.

Cuando los objetos y el entorno a inspeccionar son estables y repetitivos, se trata de una técnica muy adecuada, como por ejemplo en la inspección de productos envasados. En cambio, cuando hay una gran variabilidad, como es el caso de la inspección de numerosas materias primas, semielaborados y productos finales en la industria de los alimentos procesados, la aplicación de la visión artificial es más complicada.

Una desventaja de la visión artificial es que sólo trabaja en el espectro visible y el espectro radioeléctrico es muy amplio. Hay otras bandas espectrales que son de mucho interés porque pueden permitir obtener información no sólo de propiedades físicas, como el color o el tamaño de los objetos a inspeccionar, sino también otras magnitudes físicas, como la temperatura, o de propiedades químicas como, por ejemplo, el contenido en agua o en grasa. La banda ultravioleta (UV) puede ser interesante para tratar de detectar propiedades que tengan efecto de fluorescencia, la banda de rayos X es adecuada para obtener información de la densidad, la banda del infrarrojo cercano (*Near Infrared NIR*) es conveniente para obtener información química y la banda de alta frecuencia nos puede permitir obtener información relacionada con la estructura molecular (*Nuclear Magnetic Resonance Imaging* MRI).

Si bien la visión artificial convencional, basada en cámaras de color, sigue siendo una disciplina con muchas aplicaciones en la industria, en estos últimos años han ido surgiendo nuevas técnicas emergentes capaces de ver más allá de lo que el ojo humano es capaz de ver, abriéndose el campo de aplicaciones no sólo en el ámbito de la medicina, sino también en el sector agroalimentario.

Una tecnología de inspección muy extendida es la basada en rayos X para detección de materias extrañas de mayor densidad que los productos a inspeccionar. También se utiliza cada vez más la visión multiespectral, que combina imágenes tomadas en varias longitudes de onda, y que se ha empleado mucho en la detección de defectos en fruta.

Por otra parte, la termografía se está utilizando para optimizar túneles de secado, tostado o enfriado, ajustándolos a la temperatura de producto deseada. Finalmente, es

reseñable una tecnología especialmente novedosa que tiene un gran potencial: la visión hiperespectral.

La visión hiperespectral es una técnica de inspección que combina las bondades de la espectroscopia del infrarrojo con las ventajas de visión artificial. La espectroscopia del infrarrojo es la ciencia que estudia la interacción de la luz infrarroja con la materia, y en función de cómo es absorbida esta radiación en el espectro, es posible inferir información de la composición de la muestra.

La visión hiperespectral consiste en aplicar un espectrógrafo con una rendija entre el objetivo y el sensor de la imagen. Las lentes del objetivo captan una línea de la imagen que es difractada y proyectada en el sensor descompuesto en todas las longitudes de onda de interés, tal como se puede observar en la Figura 5. Partes de una cámara hiperespectral. De este modo, en cada instante se tiene el espectro de cada punto de la línea captada en una columna del sensor de imagen. Si el producto está en movimiento y este proceso se hace a gran velocidad, se puede ir obteniendo el espectro de cada punto de la superficie del objeto inspeccionado.

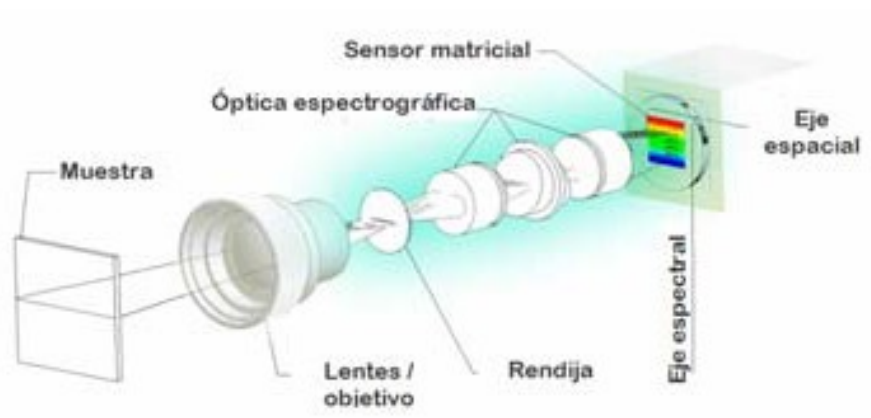


Figura 5. Partes de una cámara hiperespectral.

El espectro infrarrojo de cada compuesto tiene características propias, pues la luz infrarroja es absorbida en diferentes bandas dependiendo de los enlaces covalentes de su estructura molecular. Este fenómeno es explicado por la ley de Beer-Lambert, que indica que hay una relación entre la luz absorbida y la concentración de una sustancia a una determinada longitud de onda. Como en este caso, no hay una única longitud de onda sino múltiples, se recurre a la estadística multivariable para buscar esa relación con la concentración. Mediante esta técnica es posible obtener un modelo de calibración a

partir de un conjunto de muestras con sus concentraciones conocidas y sus espectros. Dicho modelo permite, a partir del espectro de una nueva muestra, identificarla o estimar su concentración.

Es importante resaltar que, cuando se habla de alimentos, hay multitud de ingredientes con distintas concentraciones, por lo que pueden existir efectos matriciales o de interferencia y la capacidad de los modelos generados para identificar un compuesto o estimar una concentración dependerá del tipo de producto y del parámetro a medir. Aplicando esta técnica punto a punto, espectro a espectro, es posible identificar a qué material o compuesto corresponde cada punto según su espectro, o bien estimar la concentración de producto de interés. Las aplicaciones son por tanto inmediatas, el reconocimiento de especies o la medida de concentración de un parámetro de interés en continuo [6].

Vistas las ventajas que presenta la visión hiperespectral, AINIA se decidió a hacer un estudio [7] para evaluar la detección de cuerpos con una composición diferente a la del producto objetivo.

En este estudio los productos usados fueron filetes de lomo de cerdo, productos que a menudo se cortan y embalan en atmósfera protectora y se distribuye a través de la cadena de frío, y un conjunto de materiales de diferentes composiciones y tamaños para ser utilizado como cuerpos extraños. Algunos de los materiales seleccionados pueden estar incorporados en la carne de cerdo como son fragmentos de grasa o hueso, pero otros son absolutamente diferentes y nunca deben aparecer con el producto como son plásticos, metales, insectos, etc.

El conjunto de materiales de distinta composición se eligieron de diferentes tamaños, unos 2x2 mm, 5x5 mm y 10x10 mm, para poder evaluar el nivel de detalle con que se podrían detectar los cuerpos extraños.

A continuación, los materiales se colocaron sobre una superficie de tres filetes de lomo de cerdo para ser caracterizados y clasificados por el sistema hiperespectral.

El sistema hiperespectral consiste en una cámara infrarroja (MCT-507, Xenics, Belgium) a la que se le une un espectrógrafo (Imspector N25E, SPECIM, Finlandia) para adquirir imágenes con una resolución de 320x256 píxeles en la banda del infrarrojo cercano de 1100 a 2500 nanómetros. El sistema de iluminación consiste en cuatro

lámparas infrarrojas halógenas (JCR12V75W, International Light, EE.UU.) colocadas sobre la cinta transportadora para proporcionar una luz homogénea en la medición.

Además, para lograr un control preciso del movimiento de la cinta transportadora, se utiliza un servomotor para adquirir las imágenes hiperespectrales en perfecta sincronía con el avance de la cinta.

El procedimiento de medición consistirá en colocar cada muestra cárnica en la cinta transportadora y adquirir imágenes por medio de una cámara hiperespectral. La cámara capturará solamente una fila del lomo que contiene el espectro de cada punto de la línea escaneada que a continuación será procesada utilizando uno de los dos algoritmos, PCA CE 6CP o PCA 1D15CE 8CP y que tras su análisis se obtendrá la imagen analizada donde se detectarán los cuerpos extraños, así como la composición de las distintas sustancias que forman la muestra cárnica.

2.3 Preprocesado para mejorar la imagen hiperespectral

En este apartado se explicarán conceptos matemáticos necesarios para comprender el preprocesado que se aplicará a los distintos algoritmos de clasificación de imágenes hiperespectrales.

En primer lugar se explicará el pretratamiento por centrado y escalado común en ambos algoritmos.

El siguiente apartado tratará sobre la primera derivada de Savitzky-Golay. Esta técnica se utiliza para suavizar los datos de entrada.

2.3.1 Centrado y Escalado

Para datos espectroscópicos el centrado por columnas es una técnica frecuentemente empleada como un preprocesado para posteriormente poder utilizar otros tipos de métodos como puede ser PCA. En él, la media de cada columna de datos (en este caso, por longitudes de onda) es sustraída de todos los valores de la columna para obtener una matriz de datos donde la media de cada variable procesada es cero.

A continuación, se le resta el valor medio a todos los datos de dicha columna y el resultado que se obtendrá es que todas las columnas de datos (variables) presentarán una media igual a cero [8].

Después de esto, se aplica un escalado por columnas mediante la varianza (también denominado autoescalado o estandarización por columnas) que se lleva a cabo después del centrado y divide los valores de cada columna entre la desviación típica de cada columna. El producto resultante es una matriz donde todas las columnas tienen una media cero y varianza unitaria lo que implica que toda la información restante está relacionada con la correlación entre variables.

$$d_{ij_{nuevo}} = \frac{d_{ij_{original}} - \overline{d_j}}{\sigma_j} \qquad (2.2)$$

El objetivo de esta normalización es la de dar igualdad de escala a cada una de las variables o parámetros que describen cada medida. Este tipo de escalado es muy útil, ya que enfatiza las características con menor variabilidad por lo que resulta útil en el caso de componentes químicos minoritarios cuyos efectos sobre el espectro puedan verse enmascarados [9].

Incluso en el caso de que cada variable represente el mismo parámetro. Por lo que es conveniente que todos ellos trabajen dentro del mismo rango de valores. El éxito de este escalado radica en que, a priori, asigna la misma importancia numérica a cada una de las variables que describen una medida, independientemente de su naturaleza.

El inconveniente de ésta técnica se presenta especialmente en aquellos espectros con un alto nivel de ruido ya que éste también es enfatizado pudiendo llegar a enmascarar la información de interés [9].

2.3.2 Primera derivada de Saviztky-Golay

El filtrado mediante el método conocido con el nombre de Savitzky-Golay, que fueron los autores que inicialmente lo utilizaron en 1964 para establecer con nitidez la intensidad y la anchura de las líneas espectrales obtenidas a partir de datos espectrales con ruido [10].

Actualmente, el método de Savitzky-Golay es una herramienta de trabajo aceptada en el ámbito de la ingeniería química y otras ciencias experimentales, y puede ser utilizada en el filtrado de señales de cualquier proceso [11-13]. El filtro de Savitzky-Golay es un tipo de filtro digital de respuesta impulso finito, FIR, que se aplica directamente sobre los datos en el dominio del tiempo y puede ser considerado como un filtro de media móvil generalizado. En este filtrado, se sustituye cada valor medido por el que se obtiene al hacer una regresión polinomial local con otros puntos de su entorno realizando el cálculo de los coeficientes mediante el método de mínimos cuadrados. Los parámetros a ajustar en este caso son el número de puntos a los que se tiene que ajustar el polinomio así como el orden del mismo, con la restricción de que este último sea menor al primero. También se puede utilizar para estimar las sucesivas derivadas.

El objetivo del filtro Savitzky-Golay es reducir el ruido tanto como sea posible causando la mínima distorsión de la señal, es decir, sin pérdida de resolución. Estas dos características, máxima reducción de ruido y mínima distorsión de la señal, son requisitos que actúan en sentidos opuestos. Por tanto, la elección del tipo de filtro debe ser equilibrada para cada sistema. El ruido se reduce cuanto mayor es el número de puntos del filtrado y menor es el orden del polinomio. Cuando las señales presenten variaciones acusadas en el valor de la señal, entonces el número de puntos del filtro debe ser suficientemente pequeño para evitar el promediado de esos cambios y mantener la resolución.

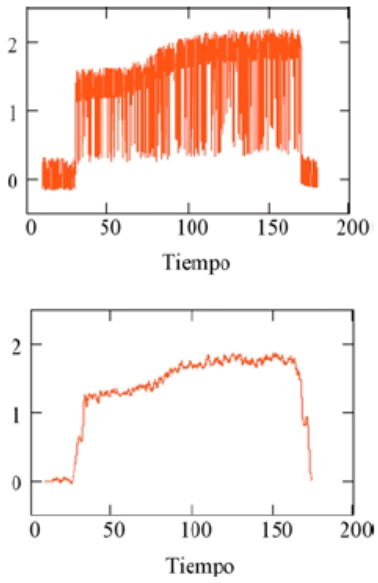


Figura 6. Superior: Señal con ruido. Inferior. Señal filtrada con Savitzky-Golay de 121 puntos. Orden n=1.

2.4 Reducción de Datos Hiperespectrales

En este apartado se estudiará una técnica, PCA, para reducir el número de variables y a su vez extraer la máxima cantidad de información posible.

2.4.1 Análisis de Principales Componentes (PCA)

El análisis de componentes principales es una técnica originalmente propuesta en 1981 [14] que surgió como respuesta a la creciente cantidad de datos que podían ser obtenidos en cada medida gracias a los instrumentos de laboratorio de nueva generación [15]. Así, por ejemplo, un espectrómetro puede proporcionar datos característicos en 10000 longitudes de onda diferentes para cada medida. Esta nueva situación crea una saturación de información cuya consecuencia más probable es la incorrecta extracción de la información que es realmente relevante para la descripción del experimento.

En definitiva, la necesidad que surge con las nuevas herramientas de laboratorio es doble: es necesario comprimir y es necesario extraer toda información relevante del enorme conjunto de datos obtenido ya que en muchas ocasiones la información esencial no depende de variables aisladas sino de la interrelación entre las mismas. El algoritmo PCA aborda estos problemas y por ese motivo es una de las técnicas más utilizadas por los químicos analíticos y, por extensión, por todos aquellos investigadores que trabajan

con las nuevas herramientas de laboratorio, como los espectrómetros, ya que permite reducir, representar y extraer información relevante al mismo tiempo [16].

Para aplicar el análisis de componentes principales se parte de un conjunto de datos dispuestos en forma matricial con una estructura de *m* filas por *n* columnas, suponiendo que se han realizado *m* experiencias y cada una se ha descrito con *n* variables. Así, cada fila corresponde a una descripción completa de un experimento y cada columna a una variable concreta que se utiliza como indicador en *n* experimentos.

Para poder aplicar el algoritmo PCA, las columnas de esta matriz (las variables) deben estar centradas y escaladas.

Por lo tanto, cada experimento o fila es un vector que pertenece a un espacio vectorial de dimensión *n* expresado en base canónica (en la que cada una de las coordenadas está asociada directamente a una de las variables descriptivas que proporciona el instrumento de medida). El análisis de componentes principales es un cambio de base, pasando de la base canónica a una nueva base formada por los autovectores de la matriz de covarianza de los datos.

En definitiva, los *scores* son las coordenadas de cada una de las medidas respecto a la nueva base, por lo que de alguna manera especifican las relaciones existentes entre experimentos. Por el contrario, los *loads* definen la contribución de cada variable original sobre las componentes principales, aportando información sobre la interrelación entre las variables originales.

En principio, la dimensión de la nueva matriz es idéntica a la de la original, pero podemos reducir la dimensionalidad conservando únicamente las coordenadas (“scores”) respecto a las primeras componentes principales, que son las que marcan las direcciones de máxima varianza (y, por tanto, de máxima información). Eso hace posible la representación gráfica en dos dimensiones de datos multidimensionales asegurándonos de que esa representación capturará la máxima varianza posible del conjunto de datos en dos dimensiones.

Por lo tanto, proyectando las variables multidimensionales sobre las dos primeras componentes principales se puede obtener información sobre la relación entre las mismas. La cercanía entre variables suele presuponer una buena correlación entre

ellas. En el caso de que estén en situación completamente opuesta indica que están fuertemente anticorreladas.

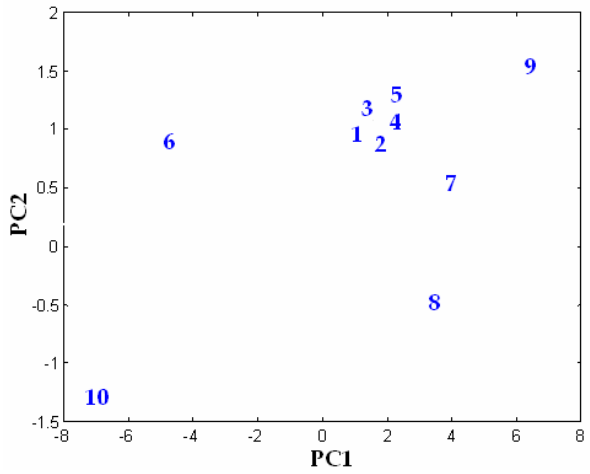


Figura 7. Ejemplo de interpretación de *loads* en un diagrama PCA.

Si se está trabajando con componentes principales, las variables que estén fuertemente correladas o anticorreladas aportan información similar, lo que indica que son redundantes. El eliminar todas menos una puede permitir reducir la dimensionalidad del vector que define la medida sin perder información importante. Sin embargo, cuando las variables son bastante ruidosas puede ser interesante conservar más de una para intentar promediar su efecto. La Figura 7 muestra un ejemplo simulado en el que se observa como las variables 1 a 5 aportan información similar (están fuertemente correladas), mientras que la variable 10 está claramente anticorrelada a la variable 9. Así pues, una posible elección para reducir el número de variables podría incluir la 1, 6, 7, 8, 9, representando la variable 1, a todas las de sus agrupaciones y la 9 a su variable anticorrelada número 10.

2.5 Extracción de características

En este apartado se explicará la extracción de características mediante el análisis discriminante lineal de Fisher.

2.5.1 Análisis Discriminante lineal de Fisher (AD)

Con independencia del área de conocimiento en la que se esté trabajando, es frecuente tener que enfrentarse con la necesidad de identificar las características que permiten diferenciar a dos o más grupos de sujetos. Y, casi siempre, para poder clasificar nuevos casos como pertenecientes a uno u otro grupo: ¿se beneficiará este paciente del tratamiento, o no? ¿Devolverá este cliente el crédito, o no? ¿Se adaptará este candidato al puesto de trabajo, o no?

A falta de otra información, cualquier profesional se limita a utilizar su propia experiencia o la de otros, o su intuición, para anticipar el comportamiento de un sujeto: el paciente se beneficiará del tratamiento, el cliente devolverá el crédito o el candidato se adaptará a su puesto de trabajo en la medida en que se parezcan a los pacientes, clientes o candidatos que se benefician del tratamiento, que devuelven el crédito o que se adaptan a su puesto de trabajo. Pero a medida que los problemas se hacen más complejos y las consecuencias de una mala decisión más graves, las impresiones subjetivas basadas en la propia intuición o experiencia deben ser sustituidas por argumentos más consistentes. El análisis discriminante ayuda a identificar las características que diferencian (discriminan) a dos o más grupos y a crear una función capaz de distinguir con la mayor precisión posible a los miembros de uno u otro grupo.

Obviamente, para llegar a conocer en qué se diferencian los grupos se necesita disponer de la información (cuantificada en una serie de variables) en la que se supone que se diferencian. El análisis discriminante es una técnica estadística capaz de decir qué variables permiten diferenciar a los grupos y cuántas de estas variables son necesarias para alcanzar la mejor clasificación posible. La pertenencia a los grupos, conocida de antemano, se utiliza como variable **dependiente** (una variable categórica con tantos valores discretos como grupos). Las variables en las que se supone que se diferencian los grupos se utilizan como variables **independientes** o variables de

clasificación (también llamadas variables **discriminantes**). Estas variables deben ser variables cuantitativas continuas o, al menos, admitir un tratamiento numérico con significado.

El objetivo último del análisis discriminante es encontrar la combinación lineal de las variables independientes que mejor permite diferenciar (discriminar) a los grupos. Una vez encontrada esa combinación (la función discriminante) podrá ser utilizada para clasificar nuevos casos. Se trata de una técnica de análisis multivariante que es capaz de aprovechar las relaciones existentes entre una gran cantidad de variables independientes para maximizar la capacidad de discriminación.

El análisis discriminante es aplicable a muy diversas áreas de conocimiento. Se ha utilizado para distinguir grupos de sujetos patológicos y normales a partir de los resultados obtenidos en pruebas diagnósticas, como los parámetros hemodinámicos en el ámbito clínico médico o las pruebas psicodiagnósticas en el ámbito clínico psicológico. En el campo de los recursos humanos se aplica a la selección de personal para realizar un filtrado de los curriculums previo a la entrevista personal. En banca se ha utilizado para atribuir riesgos crediticios y en las compañías aseguradoras para predecir la siniestralidad.

2.6 Modelos hiperespectrales

En este apartado se explicarán los dos modelos utilizados, PCA CE 6CP y PCA 1D15CE 8CP, para el análisis de los cuerpos extraños. El primer algoritmo utiliza la técnica multivariante PCA + AD con seis componentes principales (PCA 6CP) y con un pre-procesado de centrado y escalado (CE). El otro algoritmo corresponde al mismo procesado PCA + AD pero con 8 componentes principales (PCA 8CP) y con un pre-procesado distinto. Este trata primero de aplicar la primera derivada de Savitzky-Golay con una venta de 15 puntos (1D15) y a continuación un centrado y escalado (CE).

2.6.1 Modelo PCA CE 6CP

El primer paso para utilizar este modelo será caracterizar las distintas muestras, tanto cuerpos extraños como no. Para ello, se colocará los diferentes materiales en la cinta y se capturará una secuencia de imágenes. Tras procesar estas secuencias, lo que se obtendrá será una matriz donde las filas corresponderán a la caracterización de todos los cuerpos y las columnas a las distintas longitudes de onda que es capaz la cámara de captar. Tras esto, se aplicará un centrado y escalado por columnas, que tratará de calcular la media y varianza por columnas para posteriormente restarle a cada punto de esa columna la media calculada y dividirla por la desviación típica.

El siguiente paso será aplicar un análisis de seis componentes principales para el conjunto del espectro. Los coeficientes de la PCA se llevaron a cabo utilizando PLS-Toolbox 4.0 (vector propio, EE.UU.) para *MatLab* 7.5 (Mathworks, EE.UU.). Dichos coeficientes fueron suministrados por AINIA en un archivo de texto para su utilización.

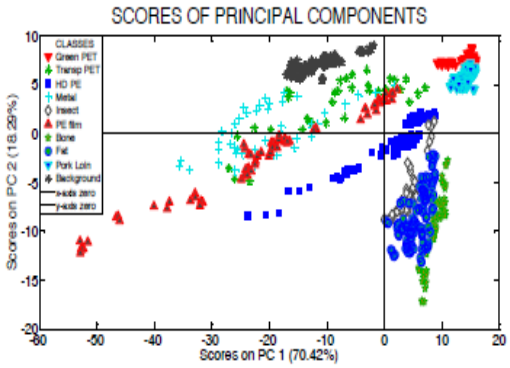


Figura 8. Imagen que representa la 1 ª y 2 ª componentes principales después de la aplicación del PCA.

Como se puede observar en la Figura 8. Imagen que representa la 1 ª y 2 ª componentes principales después de la aplicación del PCA. tras aplicar PCA hay dos agrupaciones que son muy parecidas. Una de ellas es las muestras de lomo de cerdo, triángulo azul, y otra el Green PET, triángulo rojo, por lo que su clasificación puede generar errores.

Por ello, la siguiente acción a realizar será aplicar una técnica de clasificación. Por lo tanto, se utilizará un análisis discriminante que es una de las técnicas más usadas, ya que maximizan la separación entre categorías. La función discriminante, tanto las variables dependientes como independientes, que se han usado en este algoritmo han

sido realizadas utilizando una aplicación software desarrollado en Ainia mediante Visual C + +. NET (Microsoft, EE.UU.).

$$-9.82217 - 0.171096 * CP1 - 1.90192 * CP2 + 0.0129196 * CP3 + 1.45425 * CP4 + 2.02624 * CP5 - 9.60969 * CP6$$

Figura 9. Ejemplo de la función usada para clasificar el primer grupo.

Tanto la función como los coeficientes independientes y dependientes han sido facilitados por AINIA en un documento Word para su correspondiente uso a la hora de desarrollar este algoritmo. En la Figura 9. Ejemplo de la función usada para clasificar el primer grupo. se puede ver los coeficientes usados para la clasificación de la primera clase.

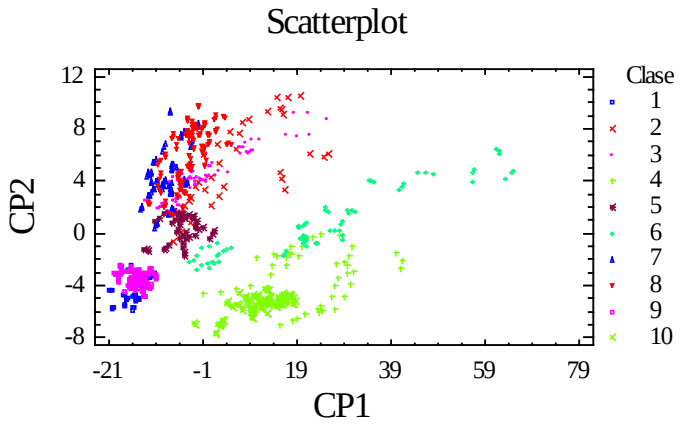


Figura 10. Imagen que representa la clasificación de los coeficientes por grupos.

Una vez clasificados los materiales y obtenidos un modelo de clasificación válido ya se podrá colocar muestras de lomo de cerdo en la cinta transportadora y que el sistema hiperespectral las analice.

El procedimiento para clasificar los pixeles de las imágenes capturadas por el sistema hiperespectral en distintas categorías será el siguiente:

1. Centrado y escalado (CE).
2. Análisis de seis componentes principales (PCA 6CP).

3. Análisis discriminante (AD).

Una vez finalizado el proceso de clasificación se generará una nueva imagen representando cada pixel de un color dependiendo en que grupo haya sido clasificado según la función discriminante. Por lo tanto, los resultados finales se pueden mostrar en la Figura 11. Los resultados obtenidos con la muestra 1 después de la representación de la imagen tras la clasificación de cada píxel., donde es posible comparar la muestra real con la imagen final obtenida.

La imagen de la muestra real es un filete de lomo de cerdo contaminada con los cuerpos extraños en la superficie, mientras que la imagen artificial está construida con los resultados de la clasificación de cada punto en función de su espectro infrarrojo.

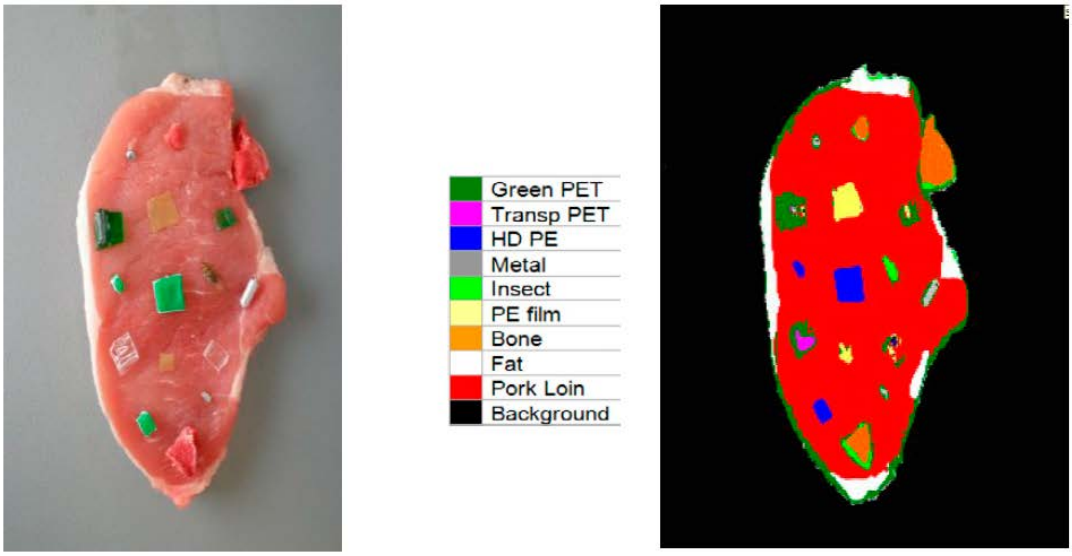


Figura 11. Los resultados obtenidos con la muestra 1 después de la representación de la imagen tras la clasificación de cada píxel.

2.6.2 Modelo PCA 1D15CE 8CP

En primer lugar se colocará los objetos extraños en las muestras cárnicas para caracterizar todos los cuerpos. Luego, se procederá a capturar una secuencia de estas imágenes para a continuación procesarlas y obtener una matriz donde las filas corresponderán a la caracterización de todos los cuerpos y las columnas a las distintas longitudes de onda. Tras esto, se aplicará la primera derivada de Savitzky-Golay con una ventana 15 puntos y un polinomio de segundo orden. Con esta técnica se reducirá el

ruido aleatorio como se explicó en el capítulo 2.3.2 Primera derivada de Saviztky-Golay. A continuación, se le aplica un centrado y escalado por columnas donde se calcula primero la media y varianza por columnas y luego a cada punto de la matriz se le resta la media y se divide por la desviación típica de su correspondiente columna.

Luego se le aplica análisis de componentes principales que en este caso será de ocho y no de seis como en el modelo anterior. Los coeficientes de la PCA como se comentó en el modelo anterior se llevaron a cabo utilizando PLS-Toolbox 4.0 (vector propio, EE.UU.) para *MatLab* 7.5 (Mathworks, EE.UU.). Dichos coeficientes fueron suministrados por AINIA en un archivo de texto para su utilización.

El siguiente paso será aplicar una técnica de clasificación. Para ello se utilizará un análisis discriminante. La función discriminante, tanto las variables dependientes como independientes, fueron realizadas utilizando una aplicación software desarrollado en Ainia mediante Visual C + +. NET (Microsoft, EE.UU.).

$$-8.92318 + 0.130141 * CP1 + 1.39266 * CP2 + 0.00287262 * CP3 - 0.182653 * CP4 - 2.06477 * CP5 + 0.897393 * CP6 - 0.634648 * CP7 - 0.81823 * CP8$$

Figura 12. Ejemplo de la función usada para clasificar el primer grupo.

Tanto la función como los coeficientes independientes y dependientes han sido facilitados por AINIA en un documento Word para su correspondiente uso a la hora de desarrollar el algoritmo. En la Figura 9. Ejemplo de la función usada para clasificar el primer grupo.12 se puede ver los coeficientes usados para la clasificación de la primera clase.

Una vez clasificados los materiales y obtenidos un modelo de clasificación válido ya se podrá colocar muestras de lomo de cerdo en la cinta transportadora y que el sistema hiperespectral las analice.

El procedimiento para clasificar los pixeles de las imágenes capturadas en distintas categorías es el siguiente:

1. Primera derivada de Savitzky-Golay con una ventana 15 puntos y un polinomio de segundo orden.
2. Centrado y escalado.
3. Análisis de ocho componentes principales.
4. Análisis discriminante

Una vez finalizado el proceso de clasificación se generará una nueva imagen representando cada pixel de un color dependiendo en que grupo haya sido clasificado según la función discriminante.

3. Hardware para la aceleración de algoritmos hiperespectrales

Este capítulo estará formado por tres partes; en la primera se introducirá una breve evolución de los dispositivos electrónicos programables hasta la actualidad. A continuación, el segundo sub-apartado corresponderá a la descripción de las distintas alternativas, donde se explicará sus ventajas e inconvenientes. Y por último, se procederá al estudio de la arquitectura seleccionada.

3.1 Evolución de la arquitectura hardware

Hoy en día el desarrollo de circuitos digitales ha llegado a tales magnitudes que gracias a ello se están implementando multitud de complejos algoritmos, los cuales deben realizarse de forma rápida y precisa. Para implementar estos algoritmos tradicionalmente han existido dos maneras: su ejecución en un procesador de propósito general y su realización en hardware a medida. La primera es muy barata y flexible, pero generalmente lenta; la segunda es de muy alta eficiencia, pero por su coste sólo se justifica en aplicaciones de uso masivo (coprocesadores numéricos, chips dedicados de procesamiento de señales, etc.) y generalmente no permite flexibilidad.

Por lo tanto surgió la necesidad de combinar estas dos soluciones. En un primer momento nació los circuitos integrados digitales programables, PLD (Programmable Logic Device), que son circuitos electrónicos que ofrecen a los diseñadores en un solo chip un conjunto de compuertas lógicas y flip-flop’s, que pueden ser programados para

implementar funciones lógicas; y así, de una manera más sencilla reemplazar varios circuitos integrados estándares o de funciones fijas. La mayoría de los PLDs están formados por matrices de conexiones: una matriz de compuertas AND, y una matriz de compuertas OR, como se puede observar en la Figura 13. Estructura interna de un dispositivo programable., y algunos, además, con registros. Estas matrices de conexiones son una red de conductores distribuidos en filas y columnas con un fusible en cada punto de intersección, mediante el cual se seleccionan cuales entradas serán conectadas a las compuertas AND/OR. Con estos recursos se implementan las funciones lógicas deseadas mediante un software especial y un programador. Las matrices pueden ser fijas o programables.

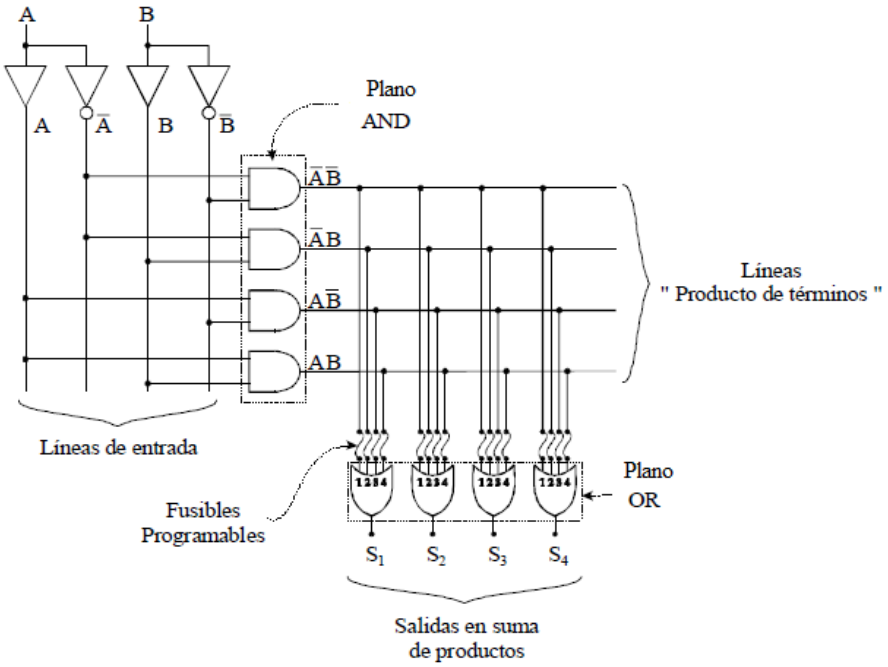


Figura 13. Estructura interna de un dispositivo programable.

Más tarde, aparecieron las FPGAs (Field Programmable Gate Array) que son conjuntos de bloques lógicos programables colocados en una infraestructura de interconexiones programables; es posible programar la funcionalidad de los bloques lógicos, las interconexiones entre bloques y las conexiones entre entradas y salidas. Un FPGA es programable a nivel hardware, por lo que proporciona algunas de las ventajas de un procesador de propósito general y un circuito especializado.

Hoy en día, las fronteras entre FPGAs y PLDs son cada vez más borrosas. En un principio, los PLDs están basados en estructuras de dos niveles lógicos, mientras las FPGAs típicamente usan lookup-tables programables de 4 entradas, ambas estructuras con un registro opcional a la salida.

Otra diferencia es el modo de programación, mientras en las FPGAs la programación es volátil y está basada en una SRAM (*Static Random Access Memory*), los PLDs típicamente utilizan programación no volátil basada en tecnología EPROM o EEPROM ((*Electrically) Erasable Programmable Read Only Memory*). Los retardos internos son otra de las principales diferencias entre FPGAs y PLDs, siendo en las primeras variables y dependientes del enrutamiento de las señales, mientras que en los PLDs los retardos son generalmente fijos y más fáciles de estimar.

Otra solución intermedia que se empleo fue que los procesadores RISC y los DSP convergieran hacia un modelo RISC-DSP programable [17]. Llevando este modelo de DSP programable a distintas soluciones como por ejemplo basadas en FPGAs que hacen uso de bloques reconfigurables y de bloques DSP programables. Este tipo de procesadores contiene las siguientes unidades funcionales: multiplicadores y acumuladores, unidad aritmético lógica; la suma, resta, and, or, not, desplazadores; para escalar los datos antes o después de una operación sobre ellos, conversores de entrada y salida, memoria de datos y memoria de programa. Además estos procesadores utilizan la arquitectura Harvard que tiene la unidad central de proceso (CPU) conectada a dos memorias (una con las instrucciones y otra con los datos) por medio de dos buses diferentes, como se pude observar en la Figura 14.Arquitectura Harvard.

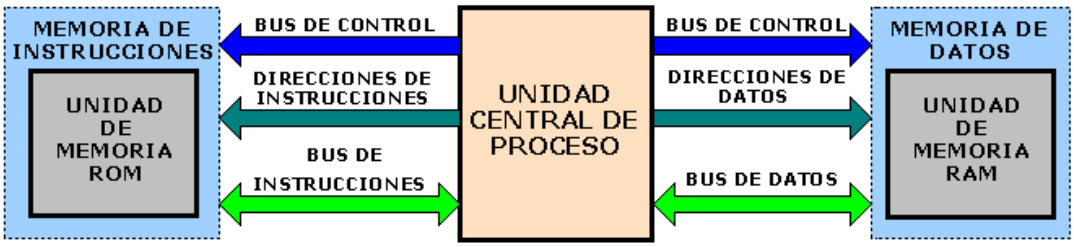


Figura 14.Arquitectura Harvard

Una de las memorias contiene solamente las instrucciones del programa, memoria de programa, y la otra, sólo almacena datos, memoria de datos. Ambos buses son totalmente independientes lo que permite que la CPU pueda acceder de forma

independiente y simultánea a la memoria de datos y a la de instrucciones. Como los buses son independientes éstos pueden tener distintos contenidos en la misma dirección y también distinta longitud. También la longitud de los datos y las instrucciones puede ser distinta, lo que optimiza el uso de la memoria en general.

Recientemente, se ha enfocado de una manera diferente la utilización de las FPGAs, aprovechando el hecho de que estos dispositivos permiten infinitas reprogramaciones en forma dinámica y que el tiempo que lleva la reconfiguración de los chips es muy pequeño. De esta manera se puede redefinir el hardware en tiempo real.

Este nuevo enfoque se ha llamado Lógica Reconfigurable o configurable. Las mismas celdas lógicas pueden ser en un determinado momento un contador, luego una ALU o un filtro digital. Cada compuerta puede realizar una función durante un tiempo y cuando esa función ya no es necesaria ser utilizada para realizar otra. La tecnología disponible posibilita la realización de funciones de gran tamaño dentro de una FPGA, o más aún la implementación de un algoritmo completo o parte de él dentro de un único integrado.

Actualmente, resulta de gran interés la integración de una computadora de propósito general con dispositivos programables como forma de conseguir una arquitectura híbrida que permita que parte de un algoritmo se ejecute en el procesador central, y parte en hardware programable y de esta forma obtener un aumento significativo en la velocidad de ejecución. Además de que posibilita la reutilización del hardware cambiando exclusivamente su programación interna [18].

3.2 Arquitecturas hardwares

Una vez llegado en este punto se elegirá que hardware se utilizará para acelerar los algoritmos explicados en el capítulo 2.1 Concepto de Imagen Hiperespectral. Para ello, se elegirá entre tres distintas arquitecturas igual de validas todas ellas para resolver la reducción de tiempos.

Este apartado contará con tres subapartados donde en cada uno de ellos se hará una pequeña introducción y se explicarán las ventajas y desventajas de las distintas arquitecturas seleccionadas, en el primer subapartado irá FPGAs, en el segundo DSPs y

por ultimo GPUs. Con esto, se tendrá una idea de lo que aportarán y lo que perjudicarán cada uno de los hardwares para resolver la problemática de la reducción de tiempos en un ámbito industrial.

3.2.1 FPGA

Los dispositivos FPGA (que proviene de la abreviación del Inglés: Field Programmable Gate Array), como el de la Figura 15. Imagen exterior de una FPGA., surgen como una evolución de los conceptos desarrollados en los dispositivos PLD y en los circuitos semi-custom. Una FPGA puede alcanzar una alta densidad de integración en un solo circuito integrado (hasta 1.000.000 de puertas lógicas equivalentes aproximadamente, y cada año aumenta su complejidad), y con velocidades de tratamiento de la información de entrada muy altas; cada día se continua mejorando la eficiencia de estos dispositivos tanto en velocidad como en complejidad [19].

Figura 15. Imagen exterior de una FPGA.

Aunque su estructura está completamente fijada a nivel de silicio, la flexibilidad en su programación es grande ya que un FPGA está formado por células independientes que se pueden programar para realizar funciones sencillas, pero debido a los amplios recursos de interconexión de que disponen, estas células se pueden conectar entre sí con celdas de entrada-salida mediante canales de conexiones verticales y horizontales para generar unas funciones lógicas de salida más complejas, tal como muestra la Figura 16. Arquitectura básica de una FPGA.

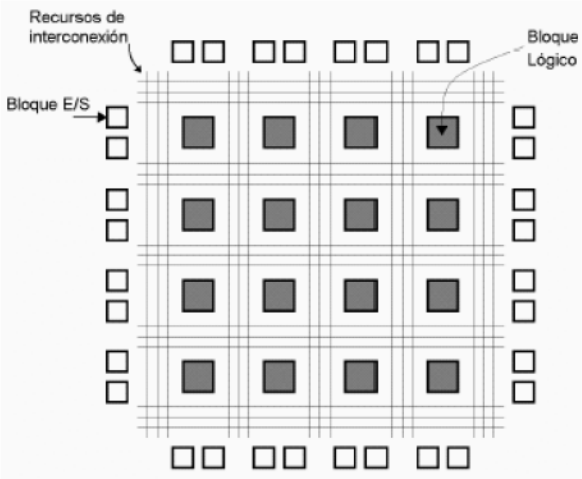


Figura 16. Arquitectura básica de una FPGA

La programación de este tipo de dispositivos se realiza en cuestión de minutos, con lo que se hacen altamente recomendables para prototipos o bajas producciones. Los problemas de un FPGA frente a la tecnología semicustom son principalmente la menor velocidad y optimización del chip, pues a menudo, en un FPGA quedan recursos sin utilizar debido a la falta de canales de conexión; además, los canales introducen retardos en la señal, y a veces es necesario un canal de gran longitud (mayor retardo) para interconectar las células requeridas. Todo esto hace que el dispositivo pierda eficiencia, pero estos factores se ven subsanados por la rapidez con la que pueden ser reprogramados, adaptándose rápidamente a las necesidades del mercado.

En cuanto a la configuración de la FPGA se realiza mediante una comunicación serie denominada bitstream [20], que puede estar almacenada en una memoria externa (PROM, EEPROM, RAM) o provenir de otro sistema (PC, Microcontrolador, otra FPGA).

Actualmente, los dispositivos FPGA son empleados en todo tipo de aplicaciones tanto científicas como de consumo; por ejemplo, actualmente se emplean en reproductores de CD, tarjetas de expansión para PC’s, dispositivos para telecomunicaciones, tareas de control de dispositivos, etc.

Algunas de las ventajas de las FPGA ya se han citado anteriormente y a continuación expondremos algunas de las más importantes [21].

- Hardware a la medida. El diseñador no tiene que buscar los productos del mercado que mejor se adapten a sus diseños, sino que se los diseña a la medida de sus necesidades, o reutiliza o modifica los diseños ya existentes. En robótica, es muy común emplear un microcontrolador u otro en función de los periféricos que traiga integrados, y es muy común que sólo se utilicen unos pocos de ellos. En un sistema que incorporen FPGAs, es el diseñador el que implemente sólo los controladores necesarios.
- Acortamiento del ciclo de diseño. El modelo de diseño hardware basado en HDL contiene muchas de las ventajas del diseño software. El circuito es ahora un fichero de texto, que se puede editar, simular, modificar y finalmente sintetizar. Se pueden crear repositorios hardware, con colecciones de diseños ya probados: controladores de VGA, UARTs, temporizadores, CPUs, etc. El diseñador puede crear prototipos muy rápidamente, probarlos, medirlos y modificarlos, es decir, el tiempo de programación y puesta en el mercado se reduce considerablemente.
- Flexibilidad. Con el mismo hardware físico, conseguimos tener hardware con comportamientos diferentes.
- Diseños hardware libres. Posibilidad de realizar diseños hardware libres que se compartan dentro de la comunidad hardware y que cualquier diseñador pueda utilizarlos, modificarlos y distribuir las modificaciones. Esto es especialmente útil en el campo de la docencia y la investigación. Esto se potencia si el hardware físico en el que se prueban los diseños es también libre. Desaparece la dependencia con el fabricante de la placa y cada Universidad o diseñador puede fabricarse las placas que considere necesarias.

Los inconvenientes de las FPGA´s son debidos principalmente a su flexibilidad, lo que las hace que en ocasiones sean inapropiadas:

- En primer lugar es más cara que su equivalente programable por máscara, esto es debido a que al tener que dejar los canales de rutado ya delimitados ocupa una mayor área y en una oblea se pueden fabricar menos.
- Es un dispositivo más lento que otros sistemas de propósito específico, debido principalmente a los transistores y matrices de interconexión que utiliza. Para

hacernos una idea aproximada los mecanismos de interconexión de una FPGA introducen aproximadamente entre el 30 y el50 % del retardo total del circuito.

- En ocasiones se desaprovecha parte de la lógica para poder realizar el rutado completo del sistema.
- Los programadores deben tener conocimientos sobre los lenguajes de descripción de hardware (HDL) como Verilog o VHL para describir la funcionalidad de estos dispositivos.
- El tiempo de reconfiguración en la FPGAs es elevado entorno al orden de los milisegundos y es proporcional al área del dispositivo a reconfigurar.
- Entornos de desarrollo propietarios, con licencias altas. Para realizar la síntesis del hardware hay que utilizar las herramientas del fabricante de las FPGA, que son caras y al día de hoy no existen alternativas libres.

3.2.2 DSP

Los rápidos avances en la electrónica, particularmente en las técnicas de fabricación de circuitos integrados, han tenido, y sin duda continuarán teniendo, un gran impacto en la industria y la sociedad. El desarrollo de la tecnología de circuitos integrados, empezando con la integración a gran escala (LSI, Large Scale Integration), y ahora con la integración a muy gran escala (VLSI, Very Large Scale Integration) de circuitos electrónicos ha estimulado el desarrollo de computadores digitales más potentes, pequeños, rápidos y baratos y de hardware digital. Estos circuitos digitales baratos y relativamente rápidos han hecho posible construir sistemas digitales altamente sofisticados, capaces de realizar funciones y tareas del procesado de señales digitales que normalmente eran demasiado difíciles y/o caras con circuitería o sistemas de procesado de señales analógicas. De aquí que muchas de las tareas del procesado de señales que convencionalmente se realizaban analógicamente se realicen hoy mediante hardware digital, más barato y a menudo más confiable.

Uno de los hardware digitales más empleados es el DSP,(que proviene de la abreviación del Inglés: Digital Signal Processor), que es un tipo de microprocesador que posee un conjunto de instrucciones, un hardware y un software optimizados para aplicaciones que requieran operaciones numéricas a muy alta velocidad. Debido a esto

es especialmente útil para el procesado y representación de señales analógicas en tiempo real [17].

Figura 17. Imagen exterior de un DSP.

La mayoría de los DSP [22] están optimizados para ejecutar operaciones a muy alta velocidad esto se consigue con operaciones del tipo Multiplier Accumulator (MAC) que se realizan en un solo ciclo de reloj. Esta operación es un paso común que calcula el producto de dos números y añade el producto resultante a un acumulador. Además los DSP's poseen arquitecturas de memoria con un acceso múltiple para permitir de forma simultánea cargar varios operándoos, por ejemplo, una muestra de la señal de entrada y el coeficiente de un filtro simultáneamente en paralelo con la carga de la instrucción. También incluyen una variedad de modos especiales de direccionamiento y características de control de flujo de programa diseñadas para acelerar la ejecución de operaciones repetitivas. Asimismo, la mayoría de los DSP incluyen en el propio chip periféricos especiales e interfaces de entrada salida que permiten que el procesador se comunique eficientemente con el resto de componentes del sistema, tales como convertidores analógicos- digitales o memoria.

Actualmente la tendencia para acelerar las operaciones por ciclo en un DSP se ha enfocado hacia el aumento del paralelismo del sistema, es decir, el “Pipelining”. Es una técnica que incrementa las prestaciones de un procesador, que consiste en dividir una secuencia de operaciones en otras de más sencillas y ejecutar en lo posible cada una de ellas en paralelo. En consecuencia se reduce el tiempo total requerido para completar un conjunto de operaciones. Casi todos los DSP del mercado incorporan el uso de la segmentación en mayor o menor medida. Aunque la mayoría de los DSP utilizan la técnica de segmentación, su profundidad o número de etapas varía de un procesador a

otro. En general, cuanto mayor sea el número de etapas menor tiempo tardará el procesador en ejecutar una instrucción.

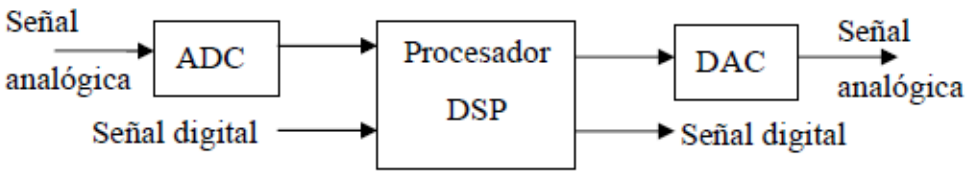


Figura 18. Sistema típico de un DSP

Como se puede ver en la Figura 18. Sistema típico de un DSP las aplicaciones clásicas de los DSP's es trabajar con señales del mundo real, tales como sonido y ondas de radio que se originan en forma análoga. Por lo tanto, el DSP se basa en el hecho de que es posible construir una representación de la señal analógica en forma digital. Esto se realiza mediante el muestreo del nivel de tensión a intervalos regulares de tiempo y convirtiendo el nivel de voltaje en ese instante en un número digital proporcional a la tensión. Este proceso es realizado por un circuito llamado convertidor analógico a digital, ADC. Con el fin de que el ADC se represente como una tensión constante, mientras que se está tomando su muestra, se utiliza un circuito de retención para muestrear la tensión justo antes de la conversión. Una vez terminado, vuelve a estar listo para la siguiente conversión.

Una vez terminada la etapa de conversión analógico-digital, los datos son entregados al DSP el cual está ahora en condiciones de procesarla. Eventualmente el DSP deberá devolver los datos ya procesados para lo cual es necesaria una etapa final que transforme el formato digital a analógico a través de una conversión digital-analógica (DAC).

A continuación describiremos las ventajas más importantes que presentan estos dispositivos, algunas de las cuales ya se han comentado anteriormente.

La primera de ellas y una de las más importantes es que permite realizar, de forma económica, tareas que serían muy difíciles de realizar o imposibles utilizando sistemas analógicos.

Otra de ellas es la insensibilidad ante variaciones ambientales y ante las tolerancias de los componentes. El comportamiento de los circuitos analógicos es fuertemente dependiente de su temperatura y se fabrican con determinadas tolerancia. Esto hace que su respuesta depende de los valores reales que tengan los componentes usados.

La combinación de las dos ventajas anteriores da lugar a una ventaja adicional y es que el su comportamiento es predecible y repetible ya que la respuesta de los sistemas DSP no varía con las condiciones ambientales ni con las variaciones de los componentes, es posible fabricar sistemas que tengan idénticas respuestas y que éstas no varíen a lo largo de la vida del sistema.

Otra ventaja es la reprogramación, es decir, como los sistemas DSP están basado en procesadores programables pueden ser reprogramadas para realizar otras tareas. Por el contrario los sistemas analógicos requieren físicamente componentes diferentes para realizar tareas distintas.

Otra virtud es el tamaño. El tamaño de los componentes analógicos varía con sus valores. Por ejemplo un condensador de 100 microfaradios utilizado en un filtro es de mayor tamaño que un condensador de 10 picofaradios utilizado en un filtro distinto. Por el contrario en un Sistema DSP ambos filtros tendrían el mismo tamaño, utilizarían probablemente los mismos componentes, diferenciándose únicamente en los coeficientes del filtro. Además el sistema DSP sería de menor tamaño que los dos sistemas analógicos.

Otra importante capacidad de estos sistemas, ya mencionada anteriormente, es la de poder ejecutar procesamiento en tiempo real para multitud de aplicaciones ya sean de comunicaciones, control, procesamiento de imagen, multimedia, etc.

Y para acabar las ventajas de los sistemas DSP podemos decir que las señales digitales pueden ser almacenadas en un disco flexible, Disco Duro o CD– ROM, sin la pérdida de fidelidad más allá que el introducido por el convertidor Analógico Digital (ADC). Esto no es el caso para las señales analógicas.

Como todo sistema no es perfecto y existen algunos inconvenientes que explicaremos a continuación.

Uno de los inconvenientes más importantes es que el diseño y desarrollo de este tipo de circuitos requiere un tiempo excesivo tanto desde el punto de vista de validación del circuito, como de fabricación. Esta causa demora notablemente los proyectos por lo que actualmente se está trabajando en el desarrollo de herramientas que realicen la implementación física automáticamente, partiendo de una netlist como la usada por las herramientas de emplazamiento de FPGAs.

Los entornos de las herramientas son poco amigables, aunque la mejora es notable, las herramientas actuales carecen de sofisticación, integrabilidad y manejabilidad para que el usuario haga un uso total de la potencia del DSP.

El rendimiento no es óptimo, ya que se diseña para una aplicación concreta, la falta de flexibilidad para modificar los algoritmos implementados constituye una gran desventaja.

Su naturaleza secuencial y de propósito específico no permite en muchas ocasiones alcanzar un procesamiento en tiempo real.

El rápido avance de la tecnología, es frecuente (por parte del fabricante), el hecho de abandonar la tendencia familiar, es decir la tendencia a conservar las características del dispositivo anterior con el fin de minimizar los costes del aprendizaje, lo cual suele provocar en los usuarios un constante aprendizaje.

La necesidad de conocimientos matemáticos, el DSP es un dispositivo con una base matemática sólida y se requieren buenos conocimientos matemáticos para poder optimizar el uso de determinados algoritmos.

Otro de los inconvenientes es el rango dinámico limitado, es decir, la amplitud del rango dinámico disponible vendrá fijado por el número de bits empleados para representar la muestra. Esto da lugar a fenómenos de saturación o de truncado. Como se deduce fácilmente, cuántos más bits tenga la muestra, mayor será la precisión en los cálculos posteriores y disminuirán los errores generados.

El ancho de banda limitado por la frecuencia de muestreo. Para obtener unos resultados aceptables, dicha frecuencia debe duplicar como mínimo la frecuencia máxima contenida en la señal analógica [23].

Y también limitaciones debido al procesamiento digital, como puede ser el error debido a la cuantificación, que se entiende por cuantificación al proceso de representar una muestra analógica por el entero más próximo que según la escala le corresponde y que lógicamente corresponderá al nivel de la señal más próximo. Este proceso necesariamente introduce un error, diferencia entre el valor real y el valor muestreado de la señal. Cuanto mayor sea el número de bits utilizado para representar la muestra (resolución), menor será este error. Este fenómeno da lugar a una degradación de la señal como consecuencia de la pérdida de información inherente a la representación de una señal analógica mediante una muestra digital con un número finito de valores.

3.2.3 GPU

Una GPU, (que proviene de la abreviación del Inglés: Graphics Processor Unit), es una unidad de procesado, especializada en cálculos relacionados con el renderizado de gráficos 2D y 3D. Dichas unidades son el núcleo de las tarjetas gráficas actuales que cualquier ordenador posee y cuyo objetivo es aligerar la carga de trabajo del procesador central (CPU) en aplicaciones con gráficos, ya que este tipo de cálculos son computacionalmente muy costosos, y así la CPU queda liberada de esta tarea.

Figura 19. Imagen de una GPU.

Debido a la rápida evolución que han llevados estas unidades hacen que actualmente estén desarrolladas bajo la perspectiva many-cores (muchos núcleos) con el objetivo de paralelizar. Esto trata de realizar varios cálculos de forma simultánea basada en el principio de dividir los problemas grandes para obtener varios problemas pequeños, que posteriormente son solucionados en paralelo.

La arquitectura de una GPU está altamente segmentada, lo que indica que posee gran cantidad de unidades funcionales y cuyas características principales son:

- No fluyen las instrucciones, sino los datos.
- Los operadores son unarios, es aquella operación matemática que sólo necesita el operador y un argumento para que se pueda calcular un valor, lo que evita las dependencias y maximiza el paralelismo.
- Se dispone de un menor número de etapas dada la baja frecuencia. Se fomenta el procesamiento vectorial y superescalar, buscando aprovechar el paralelismo de datos.

Este tipo de hardware resulta especialmente útil para algoritmos masivamente paralelos, comportamientos no aleatorios, que puedan que pueden ejecutarse de forma eficiente sobre las GPUs en el cual se ve como un coprocesador adicional a la CPU de gran capacidad de computación.

La relación que existe entre velocidad (capacidad de cálculo) y coste es muy buena debido a que se puede obtener un buen rendimiento a partir de unos 100 €.

Otra gran ventaja es que la mayoría de PCs ya poseen tarjetas gráficas por lo que su inserción en cualquier ámbito sería la solamente la colocación de un PC con una GPU.

El crecimiento del mercado de videojuegos, tanto para PC como para consolas, tiene una evolución exponencial, este hecho hace que sea una gran ventaja ya que ayuda a reducir los precios de las GPUs con pocos meses de vida.

En cuanto a las posibles desventajas que puede tener este tipo de hardware se encuentra en la forma de ejecutarse los algoritmos ya que no todos los algoritmos se pueden ejecutar de forma paralela con lo cual su ejecución no sería la óptima y no se aprovecharía el máximo rendimiento que nos permite la GPU.

Otro factor muy a tener en cuenta es que sobre la GPU la programación es considerablemente más complicada y frustrante en determinadas ocasiones debido a cambio de pensamiento, es decir, de una manera secuencial a una totalmente en paralelo.

Por otra parte, en la mayoría de casos es necesario reescribir completamente los algoritmos debido a que no tiene nada que ver la forma de ejecutarse en la CPU de forma secuencial a la GPU en forma paralela.

El hardware al igual que en DSPs y FPGAs es muy cambiante debido a que cada nueva generación añade nuevas funcionalidades y los programadores deben adaptarse a ellas.

3.3 Alternativa elegida: GPU

Dadas las características de las tarjetas gráficas actuales como su bajo coste, la elevada potencia de cálculo y la gran capacidad de computación paralela hacen que sean realmente muy útiles para diferentes ámbitos de los que originalmente fueron concebidas. Es por esto que la alternativa elegida será la GPU de la familia Nvidia debido a que el algoritmo se puede paralelizar perfectamente ya que al fin y al cabo el algoritmo trata de multiplicaciones de matrices y eso es más o menos lo que se hace cuando se trabaja con imágenes. Además Nvidia proporciona manuales, toolkits, SDK con ejemplos de muestra de forma gratuita con frecuentes actualizaciones y que la arquitectura desarrollada para la utilización de la GPU es una extensión del lenguaje de programación C. Otras de las razones de su elección es la facilidad con la que se puede implantar en cualquier entorno ya que como se ha comentado anteriormente en cualquier lugar dispone de un PC que con el simple hecho de incorporales una GPU se puede facilitar la liberación de trabajo que tenga que soporta la CPU.

3.3.1 Introducción

Los sistemas informáticos de hoy en día están pasando de realizar el "procesamiento central" en la CPU a realizar "coprocesamiento" repartido entre la CPU y la GPU.

Esto es debido a que a partir del 2003, los fabricantes de semiconductores se centraron en dos trayectorias de diseño para sus microprocesadores. Unas de ellas, Intel y AMD, han seguido una trayectoria multicore donde intentan mantener una alta velocidad de ejecución secuencial, permitiendo cierto paralelismo mediante la implementación de varios núcleos (2-8).

Y la otra vía es la manycore, GPU's NVIDIA y AMD, donde intenta maximizar el flujo de operaciones por segundo en algoritmos masivamente paralelos. Estos procesadores, no poseen núcleos tan complejos y poderosos como los de las arquitecturas multicore, pero posee muchos núcleos pequeños, que en cantidad logran superar tanto en desempeño como en consumo energético a arquitecturas multicore (bajo condiciones de programación paralela). Para ello lo que tratan es de integrar una gran cantidad de núcleos que comprometan el rendimiento de un solo núcleo en favor del rendimiento paralelo. Con este diseño basado en muchísimos núcleos son posibles cientos de miles de threads (hilos o hebras) por chip computacional.

Si se compara actualmente el rendimiento de las unidades CPU podemos ver que los resultados obtenidos en algoritmos paralelos en GPU son mucho más rápidos que en la CPU tradicional. Sin embargo en cuanto tenemos procesamientos secuenciales, con muchos saltos condicionales y cambios de contexto, la CPU se comporta mucho mejor. A continuación se puede ver la evolución de este salto que hay entre el rendimiento de la CPU y el de la GPU en los últimos años en la Figura 20 y Figura 21.

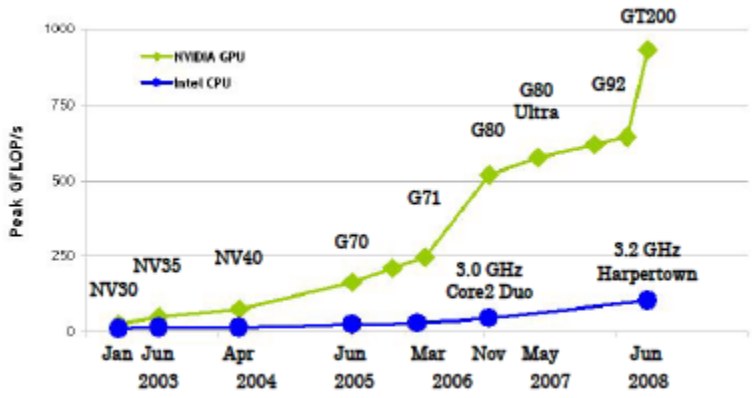


Figura 20. Operaciones por segundo entre distintas GPU's y CPU's.

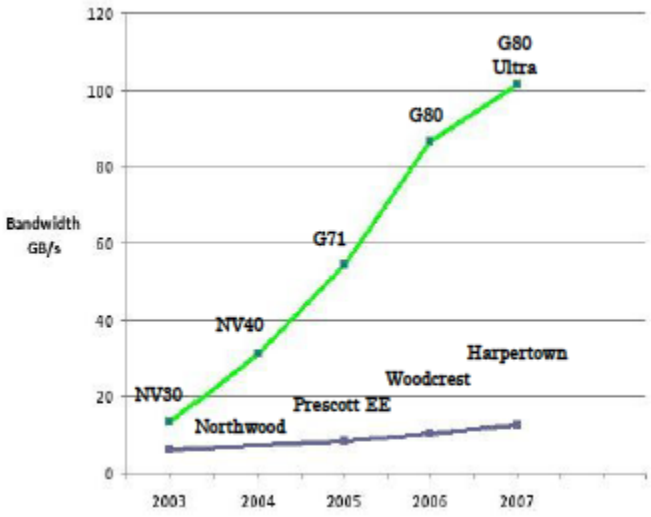


Figura 21. Ancho de banda de GPU's y CPU's.

La razón de la discrepancia en la capacidad de cálculo entre la CPU y la GPU es que la GPU está especializado en un cálculo intensivo, computación altamente paralelizables, exactamente lo que trata de la representación de gráficos, y por lo tanto diseñado de tal manera que un mayor número de transistores se dedican al procesamiento de datos en lugar de almacenamiento en caché de datos y de control de flujo, como se ilustra esquemáticamente en la Figura 22.

Figura 22. Diferencia de transistores entre GPU's y CPU's.

La GPU es especialmente adecuada para abordar los problemas que se pueden expresar como cálculos de datos masivamente paralelos con una alta intensidad

aritmética. Debido a que el mismo programa ejecuta el mismo código pero con elementos de datos distintos, además las latencias de acceso a la memoria se pueden ocultar con cálculos en lugar de cachés de datos grandes.

En la paralelización de los datos se procesan mapas con elementos de datos. Estos mapas contienen varios hilos que a su vez procesan varios elementos de datos a la vez. Muchas aplicaciones que procesan grandes volúmenes de datos pueden utilizar un modelo de programación de datos en paralelo para acelerar los cálculos.

En 3D, grandes conjuntos de píxeles y vértices se asignan a los hilos en paralelo. Del mismo modo, la imagen y los medios de procesamiento de aplicaciones como el procesamiento posterior de las imágenes renderizadas, codificación y decodificación de video, la ampliación de la imagen, la visión estéreo, y el reconocimiento de patrones se puede asignar bloques de la imagen y los píxeles que son paralelos a hilos de procesamiento.

De hecho, muchos algoritmos fuera del campo de procesamiento de imágenes y procesamiento son acelerados por el procesamiento paralelo de datos, por ejemplo procesamiento de señales en general o simulación de la física o biología computacional.

3.3.2 GPGPU

Los chips gráficos empezaron como procesadores gráficos de funciones fijas, pero se hicieron cada vez más programables y potentes desde el punto de vista computacional. Entre los años 1999 y 2000, científicos del sector informático y de otras disciplinas empezaron a utilizar las GPU para acelerar diversas aplicaciones científicas. Fue el nacimiento de un nuevo concepto denominado GPGPU o GPU de propósito general.

El GPU Computing es el uso de la GPU (unidad de procesamiento gráfico) junto con una CPU para acelerar algoritmos no relacionados directamente con el procesado de gráficos como pueden ser operaciones de cálculo científico o técnico de propósito general.

El cálculo en la GPU ofrece un rendimiento de aplicaciones sin igual al descargar en la GPU las partes de la aplicación que requieren gran capacidad

computacional, mientras que el resto del código sigue ejecutándose en la CPU. Desde la perspectiva del usuario, las aplicaciones simplemente se ejecutan más rápido.

Una CPU + GPU constituye una potente combinación porque la GPU está formada por varios núcleos optimizados para el procesamiento en serie, mientas que la GPU consta de millares de núcleos más pequeños y eficientes diseñados para el rendimiento en paralelo. Las partes en serie del código se ejecutan en la CPU mientras que las paralelas se ejecutan en la GPU.

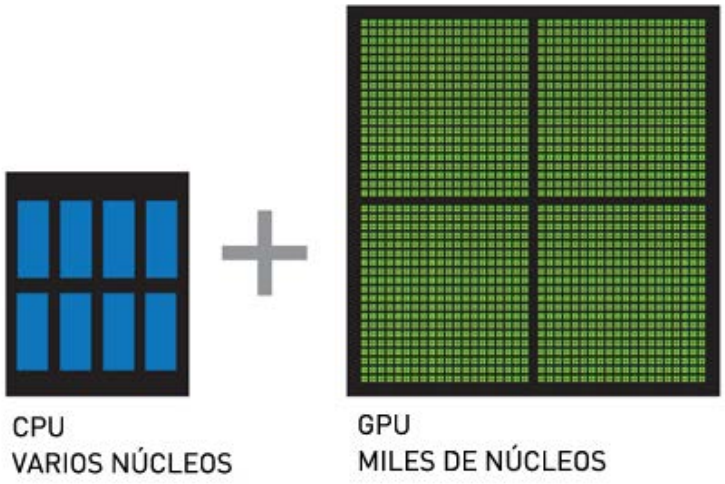


Figura 23. Diferencia de núcleos entre CPU y GPU.

3.3.3 Entornos de desarrollos

A partir del año 2003, las GPUs comenzaron a considerarse procesadores de propósito general. Los primeros lenguajes específicamente orientados a GPGPU fueron desarrollados por universidades: BrookGPU, Stanford University (2004) y Sh, University of Waterloo (2003).

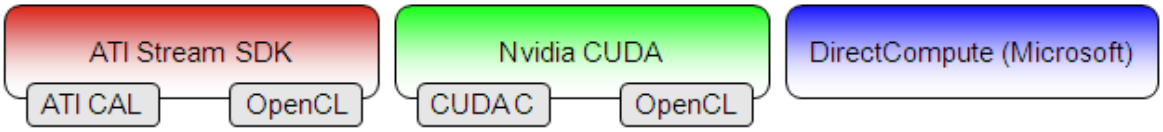


Figura 24. Distintos entornos de desarrollo.

En el año 2006 Nvidia desarrollo una arquitectura de cálculo paralelo llamado CUDA. CUDA es una extensión del lenguaje de programación C que permite

aprovechar la capacidad de procesado paralelo de las GPU multi-núcleo de Nvidia (solo podremos utilizar tarjetas gráficas de esta compañía para utilizar CUDA) con el objetivo de resolver problemas computacionalmente complejos y acelerar aquellas aplicaciones de propósito general que son paralelizables. Además Nvidia nos ofrece una gran cantidad de soporte a la programación GPGPU sobre CUDA: documentación, SDK gratuito, ejemplos de programación, actualizaciones frecuentes...

La competencia de Nvidia, ATI, desarrollo en 2008 ATI Stream que es un conjunto avanzado de tecnologías de hardware y software que permiten que los procesadores gráficos AMD (GPUs), trabajen en conjunto con el procesador central del equipo (CPUs), para acelerar aplicaciones más allá del procesamiento tradicional de gráficos y video. Esto permite que las plataformas equilibradas ejecuten tareas intensas más eficientemente, proporcionando una mejor experiencia con aplicaciones para el usuario final. Para ello ATI ofrece un SDK.

OpenCL (Open Computing Language, en español lenguaje de computación abierto). Es una interfaz estandarizada y multiplataforma basada en el lenguaje C, la cual está diseñada para el desarrollo de aplicaciones paralelas portables para mejorar en gran medida la velocidad y capacidad de respuesta para un amplio espectro de aplicaciones en numerosas categorías del mercado de juegos y entretenimiento a los programas científicos y médicos.

OpenCL fue inicialmente definido por Apple y posteriormente desarrollado por el grupo Khronos, el mismo grupo que maneja el estándar OpenGL. Existen implementaciones de compiladores de OpenCL para AMD ATI, Nvidia, CPUs X86 e incluso FPGAs de Altera.

El modelo de programación de OpenCL es similar al de CUDA C. Hay una correspondencia directa para mucha de sus funciones. Además el rendimiento obtenido con los compiladores de CUDA C es mucho más alto que con los de OpenCL.

Direct Compute de Microsoft es una nueva API orientada a la computación de propósito general sobre GPUs que soporten el estándar DirectX10 o superior.

3.3.4 Introducción a la arquitectura CUDA

En noviembre de 2006 Nvidia sacó al mercado CUDA (del inglés, Compute Unified Device Architecture), una arquitectura de computación paralela de propósito general que permite utilizar la enorme potencia de cálculo de la GPU para resolver problemas complejos y paralelizables de forma más eficiente que una CPU, convirtiendo la GPU en un coprocesador extremadamente rápido de la CPU. Se encuentra disponible desde la serie GeForce 8 de Nvidia en adelante.

3.3.4.1 Arquitectura de GPU según CUDA

CUDA presenta a la arquitectura de la GPU como un conjunto de multiprocesadores MIMD (Múltiple Instrucciones-Múltiple Datos). Cada multiprocesador posee un conjunto de procesadores SIMD (Simple Instrucción-Múltiples Datos) que se encargan de procesar los threads. Habitualmente las tarjetas disponen de entre 12 y 16 multiprocesadores, pudiendo llegar a 30 en algunos modelos.

Estos multiprocesadores denominados Multiprocesador Streaming (del inglés Streaming Multiprocessor, o SM) deben su nombre a que está compuesto de diferentes unidades independientes de procesado y a que trabaja con flujos de instrucciones. Un SM se puede definir como un multiprocesador unificado capaz de realizar tanto tareas de procesado gráfico como de propósito general. La estructura y funcionalidad de los multiprocesadores ha sido el aspecto que más ha cambiado durante la evolución de todo el ecosistema CUDA.

Además la tecnología es escalable, pudiendo agrupar varias tarjetas para conseguir mayores prestaciones. Esta idea también se integra en algunos modelos disponibles como en el Tesla S1070, que internamente está estructurado en 4 dispositivos de 30 multiprocesadores. A efectos prácticos, lo que nos interesa es saber de cuántos procesadores disponemos en la tarjeta, y vemos que se traduce en tener entre 96 y 960 en los modelos habituales, aunque como ya se ha dicho, disponiendo varias tarjetas en paralelo podemos aumentar estas cifras.

No obstante, tampoco se deberá pensar que estas son las cifras por las que se multiplicará el rendimiento de las aplicaciones respecto a su análogo serializado. Los procesadores de los que se habla no son tan potentes como los procesadores de CPUs actuales, teniendo valores medios de 1,3 Ghz. En cualquier caso, sí es determinante la capacidad de cálculo total de estos dispositivos.

Respecto a la memoria, existen numerosos modelos conviviendo en esta arquitectura. Cada procesador SIMD posee una serie de registros a modo de memoria local (sólo accesible por el procesador), a su vez cada multiprocesador posee una memoria compartida o shared (accesible por todos los procesadores SIMD del multiprocesador) y finalmente la memoria global, la cual es accesible por todos los multiprocesadores y, por ende, por todos y cada uno de los procesadores SIMD.

3.3.4.2. Modelo de Programación CUDA

CUDA propone un modelo de programación SIMD (Simple Instrucción-Múltiples Datos) con funcionalidades de procesamiento de vector [24]. La programación de GPU se realiza a través de una extensión del lenguaje estándar C/C++ con constructores y palabras claves. La extensión incluye dos características principales: la organización del trabajo paralelo a través de *threads* concurrentes y la jerarquía de memoria de la GPU con sus diferentes costes de acceso.
Los *threads* en el modelo CUDA son agrupados en *Bloques*, los cuales se caracterizan por:

- El tamaño del *bloque*: cantidad de *threads* que lo componen. Es determinado por el programador.
- Todos los *threads* de un *bloque* se ejecutan sobre el mismo SM.
- Los *threads* de un *bloque* comparten la memoria *shared*, la cual pueden usar como medio de comunicación entre ellos.

Varios bloques forman un Grid y los threads de diferentes bloques de un grid no se pueden comunicar entre sí, esto permite que el administrador de bloques sea rápido y flexible, no tiene en cuenta el número de SM utilizados para la ejecución del programa. Además de las variables en la memoria compartida, los

threads tienen acceso a otros dos tipos de variables: Locales y Globales. Las variables locales residen en la memoria DRAM de la tarjeta, son privadas a cada thread. Las variables globales también residen en la memoria DRAM de la tarjeta, se diferencian de las locales en que pueden ser accedidas por todos los threads aunque pertenezcan a distintos bloques. Esto lleva a una manera de sincronización global de los threads. Como la memoria DRAM es más lenta que la memoria compartida, los threads de un bloque se pueden sincronizar mediante una instrucción especial, la cual es implementada en memoria compartida. La figura 23 muestra la relación de los threads y la jerarquía de memoria.

Además de la memoria local, shared y global, existen dos espacios adicionales de memoria de sólo lectura, ambos pueden ser accedidos por todos los threads de un grid. Éstas son la memoria de constantes y la memoria de texturas, ambas optimizadas para determinados usos.

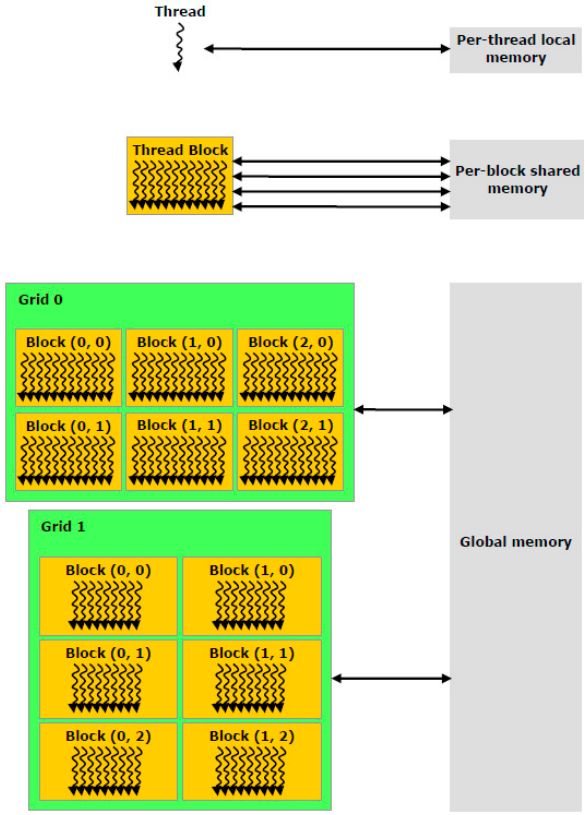


Figura 25. Threads y jerarquía de memoria

Por lo tanto, un programa CUDA está compuesto de una o más fases, las cuales son ejecutadas o en el *host* o en el dispositivo (*device)*. El *host* es el ordenador al cual está conectada la tarjeta gráfica y será quien rija su comportamiento. El *device* es la

tarjeta gráfica. Aquellas partes que exhiben poco o nada de paralelismo se implementan en el código a ejecutar sobre el *host*, no así las que pueden ser resueltas aplicando paralelismo de datos, éstas son implementadas a través de código que se ejecutará en el dispositivo, en este caso la GPU. Si bien en el programa CUDA existen dos partes bien diferenciadas, será el compilador el responsable de su diferenciación.

Para ello, el código desarrollado para ejecutarse en el *host* será compilado con el compilador estándar de C (o el del lenguaje secuencial utilizado) y ejecutado en la CPU como un proceso común. El código a ejecutarse en el dispositivo, escrito en C extendido con palabras claves que expresan el paralelismo de datos y las estructuras de datos asociadas, será compilado con el compilador propio de CUDA (*nvcc* por ejemplo), como puede apreciarse en la siguiente figura.

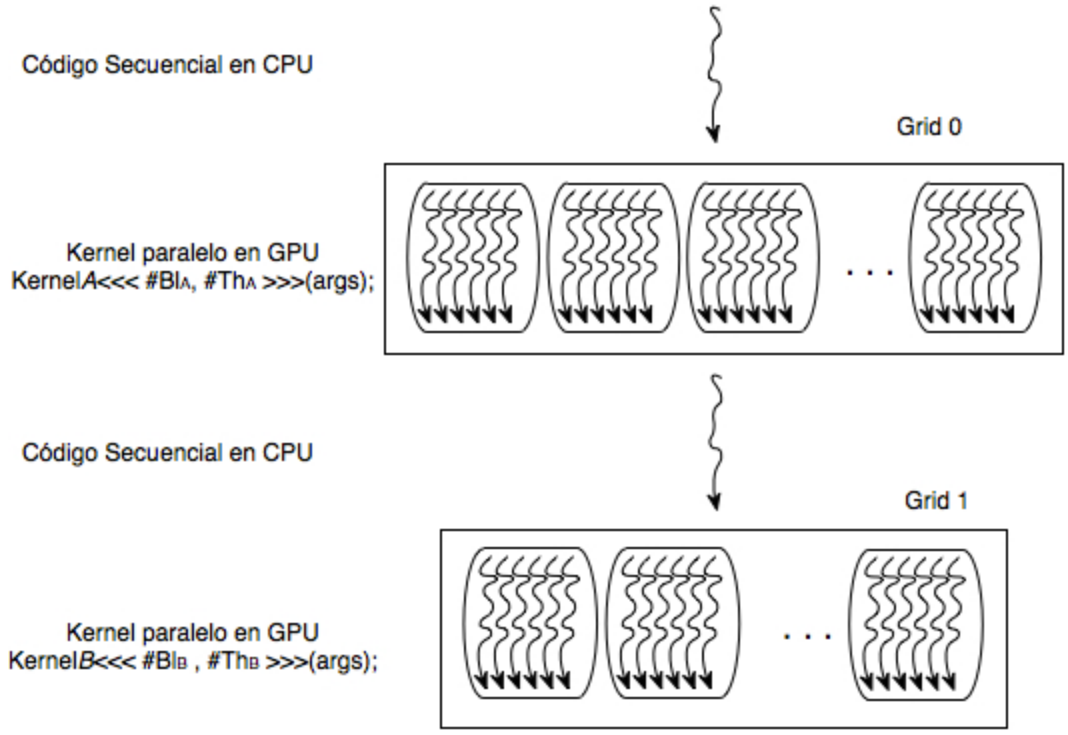


Figura 26. Modelo de programación en CPU y en GPU.

En el sistema CPU-GPU, cada uno de las componentes, host y dispositivo, tienen su propio espacio de memoria, las cuales son independientes. Para resolver un problema en la GPU, el programador necesita transferir los datos de entrada del

programa la GPU y, una vez obtenidos los resultados, transferirlos a la CPU. CUDA provee funciones para realizar estas tareas de transferencia de datos desde la CPU a la GPU y en el sentido inverso. Las transferencias de datos entre la CPU y la GPU se dan a nivel de la memoria principal de cada uno.

En la Figura 27 se muestra la visión general del modelo de memoria CUDA de la GPU, detallando las posibles transferencias y accesos a los distintos tipos de memoria.

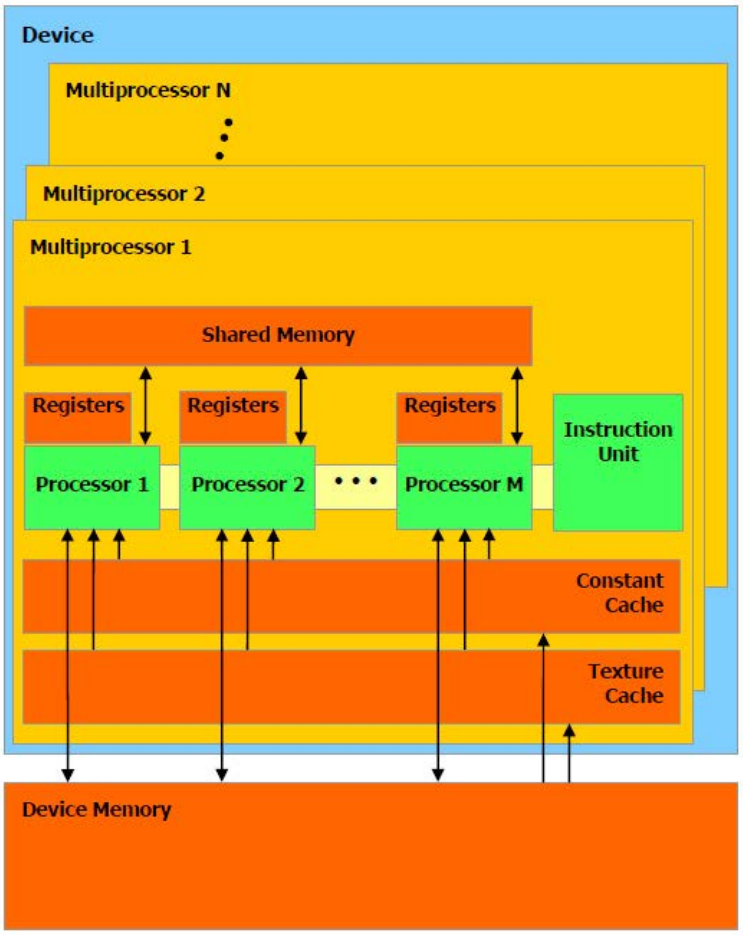


Figura 27. Jerarquía de memoria y accesos.

No todos los problemas pueden ser resueltos en la GPU, los más adecuados son aquellos que pueden resolverse mediante la aplicación del paradigma paralelo de datos, es decir aplican la misma sentencia o secuencia de código a todos los datos de entrada. Se puede decir que una solución de un problema en GPU será más ventajosa respecto a La solución en la CPU si la aplicación tiene las siguientes propiedades:

- El algoritmo tiene un orden de ejecución cuadrático o superior: el tiempo necesario para realizar la transferencia de datos entre la CPU y la GPU tiene un gran costo, el cual no suele verse compensado por el bajo costo computacional de un método lineal.
- Es mayor la carga de cálculo computacional en cada *thread*: de nuevo para compensar el tiempo de transferencia de información es conveniente que cada *thread* posea una carga computacional considerable.
- Es menor la dependencia entre los datos para realizar los cálculos, esto es posible si cada SM sólo necesita de los datos de su memoria local o compartida y no necesita acceder a memoria global, la cual tiene un acceso más lento.
- Es menor la transferencia de información entre CPU y GPU. La situación óptima es cuando la transferencia sólo se realiza una vez, al comienzo y al final del proceso. Esto significa una transferencia de los datos de entrada, desde la CPU a la GPU, y una al final, desde la GPU a la CPU, para obtener los resultados. Es bueno no tener transferencias intermedias, ya sea de resultados parciales o datos de entradas intermedios.
- No existen secciones críticas, es decir, varios procesos no necesitan escribir en las mismas posiciones de memoria, las lecturas de memoria global y compartida puede ser simultánea, pero las escrituras en la misma posición de memoria plantea un acceso a un recurso compartido, lo cual implica contar con mecanismo de acceso seguro. Este proceso hace más lenta la solución del proceso global.

Además, es necesario que las estructuras de datos en la aplicación que ejecuta en la CPU se adapten o puedan transformarse a estructuras más simples del tipo matriz o vector a fin de poder ser compatibles con las estructuras que maneja la GPU.

El modelo de programación CUDA asume que los *threads* CUDA se ejecutan en una unidad física distinta, la cual actúa como coprocesador (*device*) al procesador (*host*). Como CUDA C es una extensión del lenguaje de programación C, permite al

programador definir funciones C, llamadas *kernels*, las cuales al ser invocadas son ejecutadas en paralelo por *N threads* diferentes en la GPU.

Los *kernels* son el componente principal del modelo de programación de CUDA, son funciones invocadas desde el *host* y ejecutadas en el *device*. Cuando se invoca un *kernel*, éste se ejecuta *N* veces en *N threads* diferentes. Cada *threads* se diferencia de los demás por su identificador, el cual es único y accesible en el *kernel* a través de una variable interna y predefinida de CUDA (*built-in*) llamada *thread*Idx. A través de *thread*Idx se puede definir el comportamiento específico de cada uno de los *threads*.

Para la definición de un *kernel* se deben respetar varias condiciones, las cuales se enuncian a continuación:

- El tipo de la función *kernel* es void.
- Debe llevar la etiqueta __global__, la cual identifica a un *kernel* y determina que la función es invocada desde el *host* y ejecutada en el *device*.
- Todos los *threads* que se activen durante la ejecución del *kernel*, ejecutan el mismo programa, el cual coincide con el *kernel* que lo activó.
- El número de *threads* es conocido antes de la ejecución del *kernel*, ellos serán agrupados, según se indica en la invocación, en grupos denominados *bloques*. Todos los *bloques* tienen igual número de *threads*.

Existe una jerarquía perfectamente definida sobre los *threads* de CUDA. Los *threads* se agrupan en *bloques*, los cuales se pueden ver como vectores (una dimensión) o matrices (dos o tres dimensiones). Como se mencionó antes, los *threads* de un mismo *bloque* pueden cooperar entre sí, compartiendo datos y sincronizando sus ejecuciones. Sin embargo, *threads* de distintos *bloques* no pueden cooperar entre sí. Los *bloques* a su vez, se organizan en *grid*, el cual pueden ser de una o dos dimensiones (En las nuevas arquitecturas se admiten tres dimensiones). Cuántos *bloques* y *threads* por *bloque* tendrá un *grid* son valores establecidos antes de la invocación, los cuales permanecen invariables durante toda la ejecución del *kernel*.

Dada la organización que provee CUDA para los *threads* y como cada uno de ellos tiene un identificador único: *thread*Idx, esta es una variable de tres dimensiones, dim3 en CUDA (Ver apéndice A). Más específicamente *thread*Idx tiene tres componentes (*x*, *y*, *z*), permitiendo según la dimensión del *bloque*, identificar unívocamente a cada *thread*. Cuando el *bloque* es de una dimensión, las componentes *y* y *z* tienen el valor 1, en el caso de un *bloque* de dos dimensiones sólo la componente *z* tiene el valor 1. Para un bloque de dos dimensiones (Dx, Dy), el ID del hilo de índice (x, y) es (x + yDx) y para un bloque de tres dimensiones de tamaño (Dx, Dy, Dz) el ID de un hilo de índice (x, y, z) es (x + yDx + zDxDy).

Lo mismo ocurre con los *bloques* y los *grids*, pero para ellos CUDA tiene definida tres variables: blockIdx y blockDim para *bloques*, y gridDim para *grid*, todas de tipo dim3. BlockIdx permite identificar a los *bloques* y las variables blockDim y gridDim contienen el tamaño de cada *bloque* y de cada *grid*, respectivamente. Todas estas variables son variables *built-in*, sólo accesibles dentro del *kernel*. Al igual que *thread*Idx, tienen tres componentes. Los campos de las dimensiones (identificador o tamaño) no definidas tienen el valor por omisión 1.

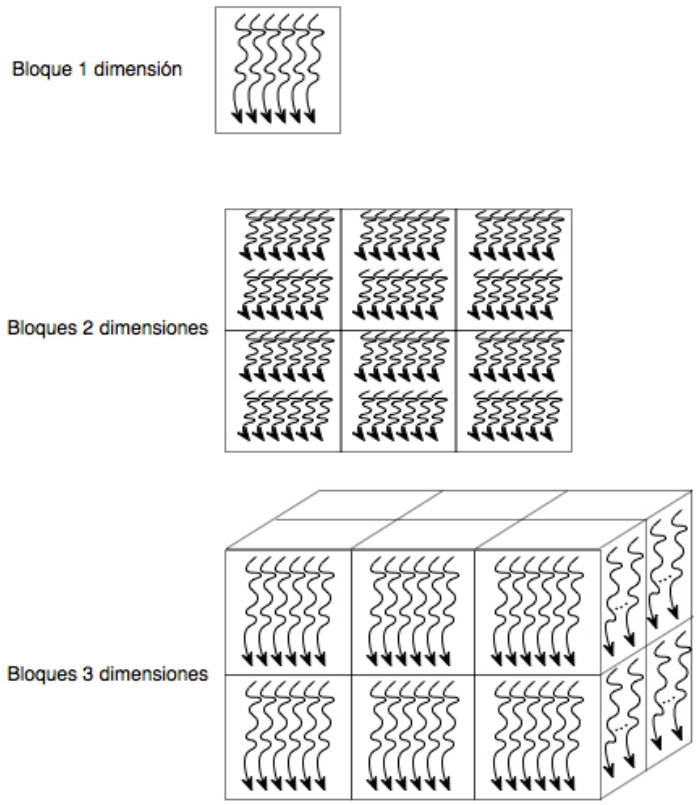


Figura 28. Organización de los threads en un bloque.

Al invocarse un kernel, se genera un grid con tantos threads como son indicados en la invocación invocarlo y se diferencia de una llamada a una función común en el código secuencial porque además de las especificaciones enunciadas anteriormente, en su invocación se establecen los parámetros necesarios para configurar la ejecución. La cantidad de threads son, generalmente, los suficientes como para aprovechar las características del hardware y expresar el máximo paralelismo del problema a resolver. A su vez los bloques pueden tener hasta 512 threads en las GPU Nvidia de la generación de G80 y 1024 en las nuevas generaciones (GT200 y GF100).

La invocación a un *kernel* sigue la siguiente estructura:

```
nombre_del_kernel <<< dim_grid, dim_block >>>(/*arguments*/);
```

En la Figura 28 se muestran bloques de una, dos y tres dimensiones. La Figura 29 muestra un grid bidimensional con bloques tridimensionales.

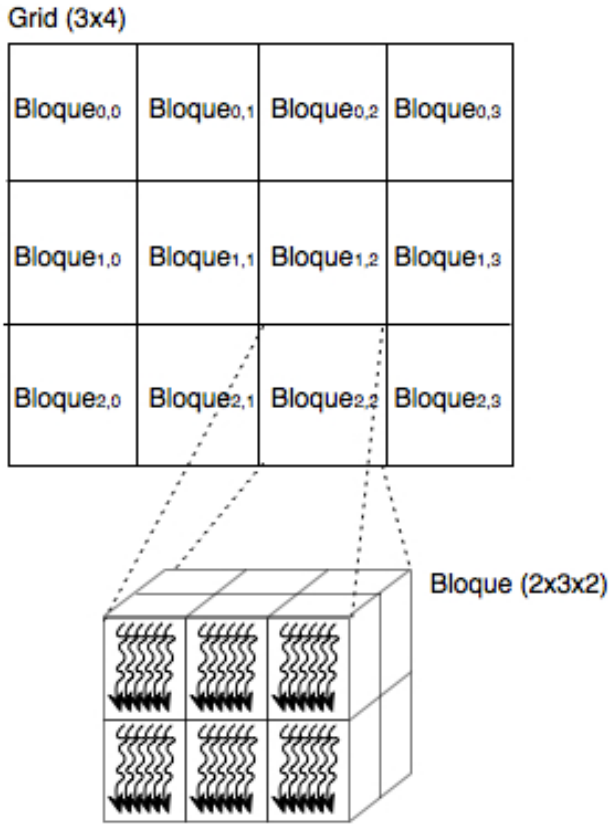


Figura 29. Organización de los Bloques en un grid.

3.3.4.3. Modelo de ejecución

La arquitectura del sistema CPU-GPU se muestra en la Figura 30 y se puede observar el flujo de procesamiento de un kernel en la GPU, pudiendo verse las distintas acciones a realizarse antes, durante y después de su ejecución. En orden cronológico, ellas son:

1. Copia de datos de la memoria de la CPU a la memoria global de la GPU.
2. La CPU ordena la ejecución del kernel en la GPU.
3. En cada uno de los cores (SP) se ejecutan los threads indicados en la invocación del kernel. Los threads acceden a la memoria global del dispositivo para leer o escribir datos.
4. El resultado final se copia desde la memoria del dispositivo a la memoria de la CPU.

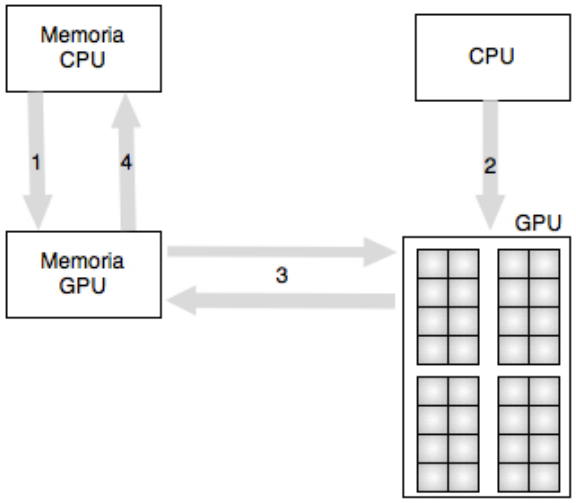


Figura 30. Flujo de procesamiento de un kernel en la GPU

El modelo de ejecución de los programas CUDA está íntimamente ligado a la arquitectura de la GPU. Los threads se asignan a los recursos de ejecución en base a los bloques. Al estar los recursos de ejecución organizados por SM, por ejemplo una de las tarjetas utilizadas en este proyecto la GTX 560ti (de la serie GT500) tiene 8 SM, cada SM cuenta con 48 procesadores escalares. Esto significa que en dicha arquitectura pueden ejecutar en paralelo hasta 384 threads. Como un SM necesita recursos para administrar los threads, los identificadores de los bloques y el estado de las ejecuciones,

existe una cantidad de threads máxima por cada bloque, dicha limitación es impuesta por el hardware. En la GTX 560 el límite de los threads a administrar por SM es 1024, lo cual significa 8192 threads ejecutándose simultáneamente.

Cuando un programa CUDA invoca un kernel con una configuración determinada, los bloques del grid son enumerados y distribuidos a los SM disponibles. Todos los threads de un bloque se ejecutan concurrentemente en un único SM. Cuando un bloque finaliza, un nuevo bloque es ejecutado en su lugar. La ejecución de los bloques es paralela/concurrente, en paralelo se ejecutan tantos bloques como SM tenga el dispositivo y concurrente porque si existen más bloques que SM, estos se ejecutan concurrentemente entre sí. En otras palabras en paralelo se ejecutan grupos de tantos bloques como SM tenga la GPU.

Cada SM crea, gestiona y ejecuta los threads concurrentemente en hardware, sin sobrecarga de planificación. Para poder administrar cientos de threads en ejecución, el SM emplea una arquitectura denominada SIMT (Simple Instrucción-Múltiples Threads)[25]. La arquitectura SIMT es similar a la arquitectura vectorial SIMD (Simple Instrucción-Múltiples Datos) en el sentido en que una instrucción controla varios elementos de procesado. Sin embargo, la diferencia clave es la organización de los vectores SIMD, en ella se exponen el ancho del vector al software, mientras que desde el punto de vista SIMT, las instrucciones especifican la ejecución y comportamiento de un único thread. A diferencia de las máquinas SIMD, SIMT permite al programador describir paralelismo a nivel de thread para threads independientes, así como paralelismo a nivel de datos para threads coordinados. El programador puede ignorar el comportamiento SIMT para el correcto funcionamiento de su código, sin embargo es un detalle muy importante para lograr un buen rendimiento.

El SM asigna un thread a cada SP, ejecutando cada uno el código de forma independiente basándose en sus registros de estado. El multiprocesador SIMT crea, administra y organiza la ejecución. Ejecuta los threads en grupos de 32 threads paralelos llamados warps. Los threads que componen un warp empiezan a ejecutar el código de forma conjunta en la misma dirección de programa, siendo libres para ejecutar sentencias condicionales.

La pregunta ahora es: ¿Cómo es dividido un bloque en warps? Los threads son divididos en grupos según sus identificadores y en orden creciente, el thread 0 pertenece al primer warps. En la Figura 31 se muestra la división de N bloques en warps.

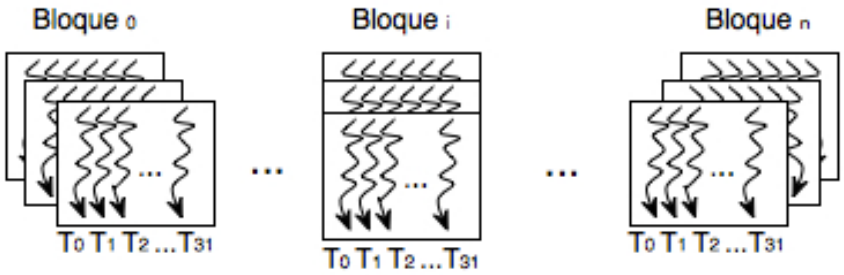


Figura 31. División de N bloques en warps

La unidad SIMT selecciona un warp listo para ejecutar y prepara la siguiente instrucción para los threads activos del warp. Un warp ejecuta una instrucción por vez, de modo que la mayor eficiencia se obtiene cuando los 32 threads del warp coinciden en el mismo camino de ejecución, es decir no hay sentencias condicionales por la cual se produce una bifurcación en la ejecución. Si la ejecución de los threads toma diferentes caminos, el warp ejecuta en forma serializada cada una de las ramas, deshabilitando los threads no pertenecientes a dicha rama; cuando todas las ramas finalizan, los threads se reúnen en el camino común. La divergencia de saltos ocurre únicamente entre los threads de un warp; los warps diferentes ejecutan código de forma independiente, sin importar qué camino están ejecutando los otros.

Ahora explicare la administración (scheduling) de los threads. Cuando un bloque se asigna a un SM se lo divide en grupos de 32 threads, warps (la cantidad de threads por warps depende del hardware, no es una especificación de CUDA). Un warp es la unidad de administración de threads. En la figura 24 se muestra la división de los threads en warps de 32 threads cada uno. Como se mencionó antes, los identificadores de los threads son consecutivos, es así que los threads 0 al 31 forman el primer warp, los threads 32 al 63 al segundo warps y así sucesivamente. El número de warps a tratar por cada SM es establecido según la cantidad de bloques a resolver y la cantidad de threads por bloque, aunque existen máximos definidos para cada arquitectura.

La administración de los threads en warps permite realizar diferentes optimizaciones, entre las más importantes se encuentran:

- Ejecutar eficientemente operaciones de gran latencia como son los accesos a la memoria global del dispositivo. Si una instrucción debe esperar por otra iniciada antes y con mayor latencia, el warp no es seleccionado para su ejecución, otro sin estas características será el elegido. Si existen varios warps listos para ejecutar, se selecciona uno según una prioridad establecida.
- Llevar a cabo ejecuciones más eficientes de operaciones de alta latencia como son las operaciones aritméticas de punto flotante y las instrucciones de branch (salto). Si hay muchos warps, es posible la existencia de alguno listo para ejecutar mientras se resuelven las operaciones más lentas.

Todas estas características son posibles porque la selección de un warps listo no implica coste, esto es conocido como scheduling de threads con overhead cero.

Esta optimización de la ejecución de threads, evitando las demoras de unos ejecutando otros es lo que en la bibliografía se conoce como ocultamiento de la latencia (latency hiding)[26]. Esta propiedad es una de las principales razones por las que la GPU no dedican áreas del chip a memoria caché y mecanismos de predicción de salto como lo hace la CPU, dichas áreas pueden ser utilizados como recursos para la ejecución de operaciones de punto flotante por ejemplo.

Conocer todas estas características de ejecución permite, a la hora de resolver un problema en la GPU, determinar el valor óptimo de bloques y de threads por bloques para resolver el problema en consideración, pudiendo establecer la subutilización del dispositivo por ejemplo. No siempre la máxima cantidad de threads por bloques es la cantidad óptima.

3.3.4.4 Modelo de Memoria de GPU

Como se mencionó anteriormente, la tarjeta gráfica posee distintos niveles de memoria, algunos de ellos *on-chip* (dentro de la GPU) y otros *off-chip* (fuera de la GPU). Desde el punto de vista de la ejecución, las instrucciones a memoria se tratan de forma diferente según a qué nivel se esté accediendo. La latencia de una instrucción de memoria varía dependiendo del espacio de memoria al que se accede e incluso del patrón de acceso. Recuerde que los cambios de contexto en la ejecución de un *threads* se producen cuando se ejecuta una instrucción de memoria, evitando así que el SM permanezca inactivo.

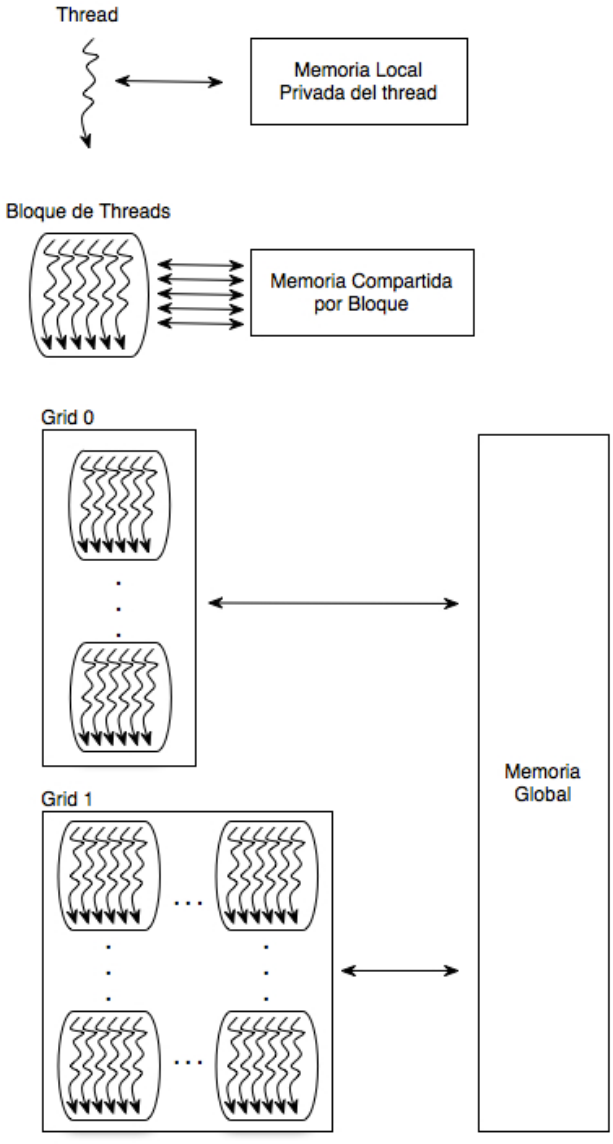


Figura 32. Jerarquía de memoria en CUDA.

En la Figura 32 y Figura 33 se muestra la jerarquía completa y quienes pueden acceder a cada una de ellas, las flechas indican el tipo de acceso, sólo lectura (Unidad de procesamiento←Memoria), sólo escritura (Unidad de procesamiento→Memoria) y lectura/escritura (Unidad de procesamiento ↔Memoria).

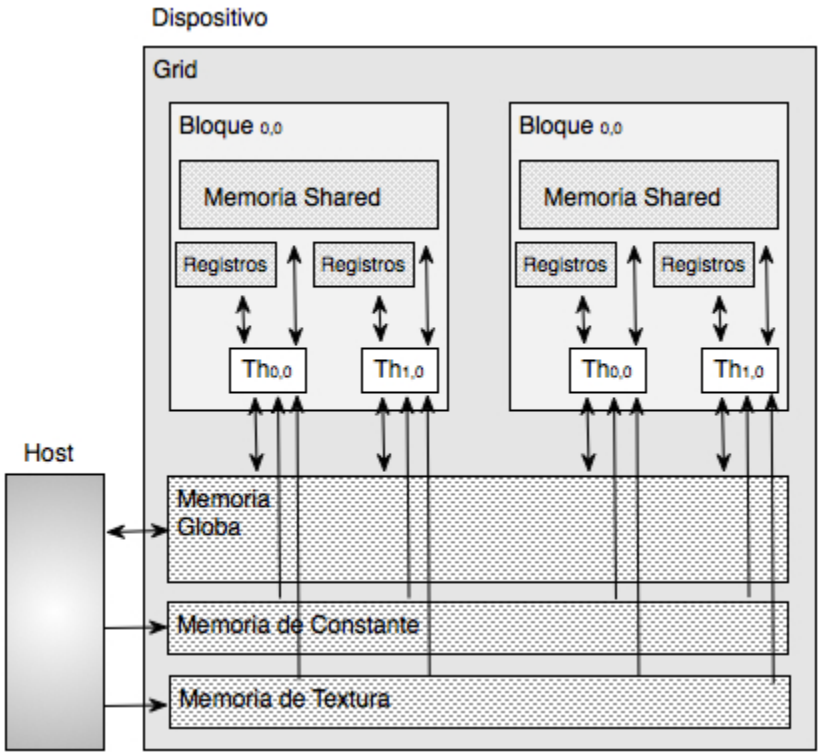


Figura 33. Accesos a la jerarquía de memoria.

Los distintos tipos de memorias se diferencian entre ellos, no sólo por las restricciones de acceso y los tamaños, sino también por las velocidades de acceso, cada una de las memorias tiene su propio ancho de banda. Esto es un punto a tener en cuenta para lograr una buena eficiencia de la aplicación desarrollada.

Es el programador quien decide las características de las variables a utilizar en sus soluciones, estableciendo en qué memoria las aloja. Con ello, no sólo determina la velocidad y visibilidad de la variable a los participantes de la solución, sino también su alcance y tiempo de vida. Cada uno de estos conceptos se refieren a:

- El alcance: Establece qué threads acceden a la variable. Un único thread, todos los threads de un bloque, todos los threads de un grid o todos los threads de una aplicación. Dadas las características del modelo de programación SPMD, si una variable es declarada en un kernel como variable privada, una instancia privada se creará para cada thread ejecutando el kernel, pudiendo acceder únicamente a su versión. Por ejemplo, si un kernel ejecutado por un grid con 1000 bloques de 256 threads cada uno, declara 4 variables privadas, 256000 instancias de cada una de las variables privadas conviven al mismo tiempo en el dispositivo.
- El ciclo de vida: Indica la duración de la variable en la ejecución del programa, este puede ser durante la ejecución de un kernel o durante toda la aplicación, es decir permanece y es accesible por todos los kernel involucrados en la aplicación. En el primer caso, están incluidas todas aquellas variables declaradas dentro de un kernel, su valor se mantiene durante la ejecución del mismo, diferentes ejecuciones de un kernel no implica mantener el mismo valor en dichas variables. En el segundo caso se encontrarán todas aquellas variables declaradas en el ámbito de la aplicación, no de un kernel, para ello los valores de estas variables se mantienen a través de los distintos kernel de la aplicación.

Los distintos tipos de memoria de la jerarquía son:

- Memoria global: Es una memoria de lectura/escritura, se localiza en la tarjeta de la GPU y la acceden todos los threads de una aplicación.
- Memoria compartida o shared: Es una memoria de lectura/escritura para los threads de un bloque, el acceso es rápido porque se encuentra dentro del circuito integrado de la GPU.
- Memoria de registros: Es la memoria más rápida, de lectura/escritura para los threads. Está dentro del circuito integrado de la GPU y sólo accesible por cada thread.
- Memoria local: Es una memoria de lectura/escritura privada de cada thread. Es off-chip y se ubica en la memoria global del dispositivo.
- Memoria de constante: Es una memoria off-chip, pero como tiene una caché onchip el acceso es rápido. La acceden todos los threads de una aplicación sólo para realizar lecturas.

- Memoria de texturas: Al igual que la memoria de constante, es una memoria offchip con caché. Los threads de toda la aplicación la acceden para leer datos escritos por el host.

3.3.4.4.1 Memoria Global

La memoria global es una memoria off-chip, por lo tanto su acceso es lento. Esta memoria se divide en tres espacios de memoria separada: la Memoria Global, la Memoria de Textura y la Memoria de Constantes. El primer espacio es de acceso tanto de lectura como de escritura, no así la memoria de constante y de textura, a la cual todos los threads de un grid pueden acceder sólo para lecturas. Tanto la memoria de textura y constante poseen caché on-chip.

Para declarar una variable en memoria global se antepone, opcionalmente, a la declaración de la misma la palabra clave __device__. Una variable en memoria global es declarada en el host. Todos los threads de todos los bloques del grid pueden acceder a la variable global, ésta puede verse como una variable compartida por todos los threads del grid. El ciclo de vida de una variable global es todo la aplicación, esto significa que una vez finalizado un kernel los valores de las variables globales modificadas persisten y pueden ser recuperados y accedidos por los siguientes kernels. Por lo tanto, las variables globales pueden utilizarse como medio de comunicación de threads de distintos bloques. Generalmente se utilizan para transmitir información entre distintos kernels.

El acceso a una variable global es más lento y como todos los threads de un grid pueden acceder a ella, son necesarios mecanismos de sincronización para asegurar la consistencia de los datos, una de ellas son las funciones atómicas.

CUDA tiene limitaciones respecto al uso de variables punteros a la memoria del dispositivo. En general, se usan punteros a datos de la memoria global. Los punteros se pueden usar de dos maneras, una cuando un objeto es alojado en la memoria del dispositivo desde el host a través de la función cudaMalloc() y pasado como parámetro

en la invocación del kernel. El segundo uso es para asignarle la dirección de una variable global para trabajar sobre ella

Como se mencionó antes, el acceso a la memoria global es más lento, dependiendo del patrón de acceso puede variar la velocidad. Los accesos a la memoria global se resuelven en medio warp. Cuando un acceso a memoria global es ejecutada por un warp, se realizan dos peticiones: una para la primera mitad del warp y otra para la segunda mitad. Para aumentar la eficiencia de la memoria global, el hardware puede unificar las transacciones dependiendo del patrón de acceso. Las restricciones para la unificación dependen de la arquitectura, en las más modernas GPU basta con que todos los threads de medio warp accedan al mismo segmento de memoria. El patrón de acceso dentro del segmento no importa, varios threads pueden acceder a un dato, puede haber permutaciones, etc. Sin embargo, si los threads acceden a n segmentos distintos de memoria, entonces se producen n transacciones. El tamaño del segmento puede ser:

- 32 bytes si todos los threads acceden a palabras de 1 byte,
- 64 bytes si todos los threads acceden a palabras de 2 bytes,
- 128 bytes si todos los threads acceden a palabras de 4 bytes.

Si los threads no acceden a todos los datos del segmento, entonces se leen datos que no serán usados y se desperdicia ancho de banda. Por ello, el hardware facilita un sistema para acotar la cantidad de datos a traer dentro del segmento, pudiendo traer subsegmentos de 32 bytes o 64 bytes. La Figura 34 muestra algunos ejemplos de acceso a la memoria global.

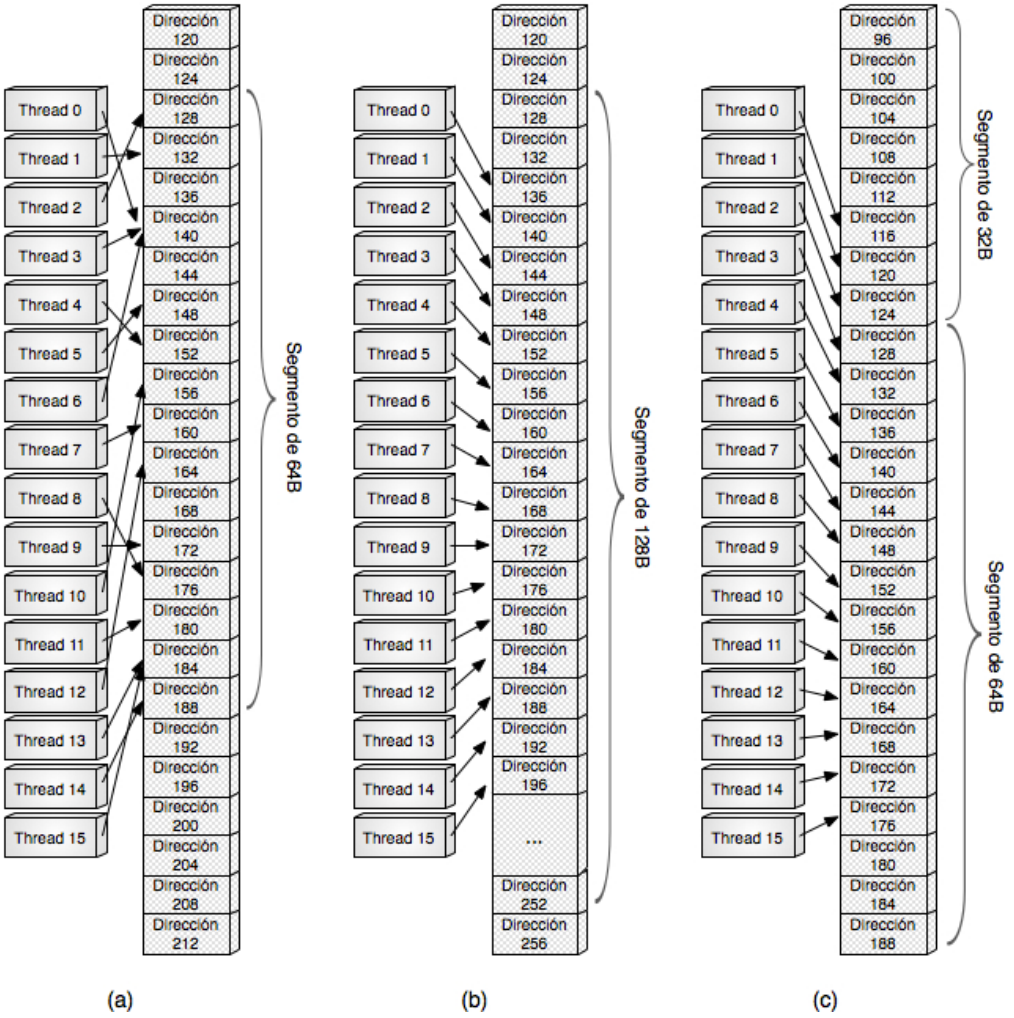


Figura 34. Patrones de acceso a la memoria global.

En los tres casos los threads acceden a palabras de 4Bytes, esto implica un tamaño de segmento de 128Bytes. En el caso de la figura 32.(a), los threads acceden a 16 posiciones consecutivas alineadas con el segmento de 128B, por lo que el hardware reduce el tamaño a 64B para evitar leer datos inútiles. De esta forma, realiza una única transacción de 64B. En la figura 32.(b) ocurre todo lo contrario, también se accede a 16 posiciones, pero al no estar alineadas, el hardware no puede reducir la cantidad de datos a leer y genera una transacción de 128B. Por último, en la figura 32.(c), los threads acceden a palabras alojadas en distintos segmentos de 128B implicando la generación de dos transacciones, las cuales son una de 32B y la otra de 64B.

La memoria global es una de las memorias más utilizadas, es el medio de comunicación entre el host y la GPU. Además es la memoria de mayor tamaño, su buen

uso implica tener en cuenta todas las consideraciones aquí mencionadas para obtener un buen rendimiento.

3.3.4.4.2 Memoria Compartida

La memoria compartida o memoria *shared* recibe su nombre porque es una memoria común a todos los *threads* de un *bloque*, siendo el medio de comunicación entre ellos.

Es una memoria *on-chip*, su acceso es rápido, llegando a ser en algunos casos igual de rápido como acceder a la memoria de registro o a la memoria constante.

La declaración de una variable compartida está precedida por la palabra clave __share__. Una variable compartida se declara dentro del ámbito de una función *kernel* o de una función del dispositivo. El alcance de una variable compartida está circunscripto a un *bloque*, esto significa que todos los *threads* del *bloque* ven la misma instancia de la variable compartida. Cada *bloque* tiene una instancia propia de dicha variable, lo cual hace que pueda verse como una variable privada a nivel de *bloque*.

El tiempo de vida de una variable compartida es la función donde es definida, es decir la función *kernel* o la función de dispositivo. Al finalizar la ejecución de la función que declaró la variable compartida, ésta dejará de existir.

Las variables compartidas son un medio eficiente para la comunicación de los *threads* dentro de un *bloque*, mediante ellas un *thread* puede colaborar con los demás. Como es una memoria más rápida que la memoria global, los programadores suelen utilizarla para almacenar datos de la memoria global muy utilizados por el *kernel*.

Para tener una velocidad de acceso a la memoria compartida similar a la de la memoria de registro se debe asegurar la no existencia de conflictos de accesos. El conflicto de acceso se da a nivel de *bancos*, un *banco* de memoria compartida es la unidad en la que está organizada, todos tienen el mismo tamaño (1KB) y pueden ser accedidos simultáneamente. Los *bancos* están organizados de forma que palabras

sucesivas de 32 bits pertenecen a *bancos* sucesivos (pudiendo ser distintos). El ancho de banda de cada *banco* es de 32 bits cada 3 ciclos de reloj.

La organización de la memoria compartida en *bancos* se realiza para obtener un mayor ancho de banda, al estar dividida en *bancos*, la memoria puede atender con éxito una lectura o una escritura en *n* direcciones pertenecientes a *n bancos* distintos. De esta manera se puede conseguir un ancho de banda *n* veces mayor a si existe un único *banco*.

Un conflicto de memoria implica dos accesos simultáneos de *threads* distintos al mismo *banco* de memoria. La resolución se hace a través de la serialización de los accesos a la memoria, siendo responsabilidad del hardware la serialización. Es este quien divide el acceso a memoria en tantos accesos libres de conflicto como sean necesarios, llevando a una pérdida del ancho de banda en un factor igual al número de peticiones libres de conflicto.

Para conseguir el máximo rendimiento es importante evitar los conflictos de *bancos*, esto implica conocer la organización de los *bancos* de memoria y comprender la forma en la cual se acomodan los datos en ellas. De esta manera se podrán programar los accesos a memoria compartida a fin de minimizar los conflictos.

Para evitar esto, cuando un *warp* ejecuta una instrucción sobre la memoria compartida, la petición se separa en dos peticiones: una para la primera mitad del *warp* y otra para la segunda mitad. De este modo cada uno de los 16 *threads* del medio *warp* puede acceder a un *banco* y obtener el máximo rendimiento. En la figura 28 se muestran algunos ejemplos de patrones de acceso sin conflictos en la memoria compartida.

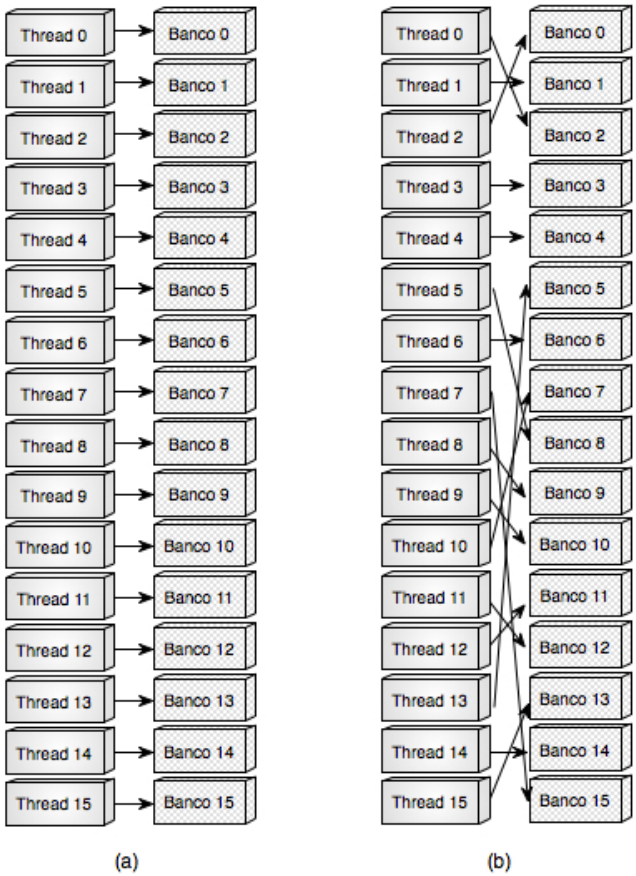


Figura 35. Accesos sin conflictos a la memoria compartida.

En la Figura 35, cualquiera de los dos patrones de acceso a los *bancos* de memoria no producen conflictos, cada uno accede a un *banco* diferente. La diferencia entre ambos esquemas de acceso es la forma en la que lo hacen. En (a) cada *thread* accede al *banco* de memoria en forma lineal, en cambio en (b) el acceso se hace en forma aleatoria, no existe ningún patrón de acceso.

Los accesos al mismo *banco* sin conflicto son permitidos. Esto es posible porque la memoria compartida implementa un mecanismo de distribución a través del cual una palabra puede ser leída por varios *threads* simultáneamente en la misma acción de lectura sin producir conflicto. Esto reduce el número de conflictos sobre un *banco*, siempre y cuando todos los *threads* accedan a leer la misma posición de memoria. La Figura 36 muestra dos casos en los cuales no existe conflicto porque los *threads* leen del mismo *banco* la misma posición.

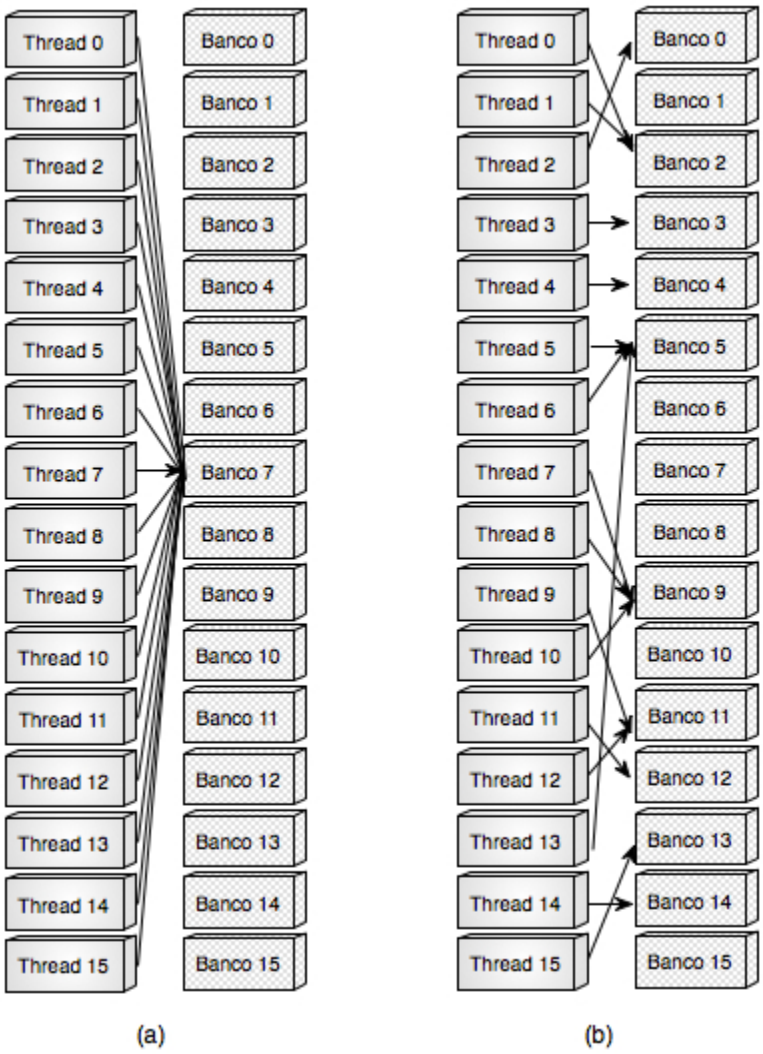


Figura 36. Accesos sin conflictos a la memoria compartida

En el caso (a) todos acceden a la misma posición del mismo *banco*, no así en el caso (b) donde se accede a distintos *bancos* y en el caso de existir coincidencia es a la misma posición. Si los *threads* acceden al mismo *banco* de memoria pero a diferentes posiciones, entonces existe conflicto de *banco*. Como se mencionó antes, la solución será serializar los accesos.

3.3.4.4.3 Memoria de Registros

La memoria de registro reside *on-chip*, permite accesos a muy alta velocidad y es altamente paralela. El acceso es considerado de costo cero. Los registros se asignan a los *threads* individuales, teniendo cada uno la privacidad sobre ellos. Su principal función es almacenar en ella aquellos datos, los cuales se necesitan frecuentemente y son privados a cada *thread*.

Todas aquellas variables escalares, no matrices o arreglos, declaradas dentro de un *kernel* son ubicadas automáticamente en la memoria de registros. Su alcance es dentro de cada *thread* individual y el ciclo de vida está limitado al *kernel* donde se declaró. Al invocarse el *kernel*, una copia privada de cada una de las variables automáticas es creada para cada *thread.*

3.3.4.4.4 Memoria Local

El espacio de memoria local se encuentra dentro del espacio de memoria global y por lo tanto, es tan costoso acceder a ella como lo es acceder a la memoria global. La memoria local se utiliza de forma automática por el compilador para alojar variables que no tienen lugar en los registros o demandan un gran número de registros a ser usados por cada *thread*. El exceso de registros se ubica en esta memoria.

3.3.4.4.5 Memoria de Constantes

La memoria constante es una memoria *off-chip*, aunque soporta baja latencia y gran ancho de banda porque tiene una memoria caché asociada *on-chip*. Los *threads* de un *grid* acceden a la memoria constante sólo para lectura, es el *host* el responsable de escribir en ella.

La declaración de una variable en la memoria constante de la GPU es hecha en el *host* y debe estar precedida por la palabra clave __constant__, opcionalmente se le

puede agregar la palabra clave __device__ antes de __constant__ y se logra el mismo efecto.

El ámbito de una variable de memoria de constante es la aplicación, todos *threads* de cada uno de los *grids* de la aplicación ven la misma variable. El ciclo de vida de estas variables es toda la aplicación.

El uso más frecuente de las variables de memoria de constantes es el de proporcionar valores de entrada a las funciones de un *kernel*. Las variables constantes se almacenan en la memoria global, pero se cargan en la caché para un acceso eficiente. Con los patrones de acceso adecuado, el acceso a la memoria de constante es rápido y paralelo.

El coste de acceso a la memoria constante puede ser diferente, esto obedece al hecho de que como es un espacio con caché, el costo de acceso sería:

- Semejante al acceso a la memoria de registro, si la referencia está en la caché y dos o más *threads* del medio *warp* acceden a la misma posición de caché.
- Linealmente superior al anterior si las direcciones están en la caché y son distintas.
- Igual de costoso a una lectura en memoria global cuando se produce fallo de caché.

La memoria de constante es muy útil cuando la aplicación utiliza datos provistos por el *host* y no se modifican durante toda la aplicación, por ejemplo límites preestablecidos o datos calculados en CPU y necesarios en la GPU.

3.3.4.4.6 Memoria de Texturas

El espacio de memoria de textura es una memoria *off-chip* de sólo lectura para los *threads*. Una variable en la memoria de texturas tiene un ciclo de vida igual al de la aplicación y es accedida por todos los *threads* de los diferentes *grid*.

CUDA soporta un subconjunto del hardware de textura que usan las GPU. El acceso a la memoria de textura desde el *kernel* se hace mediante la invocación de funciones denominadas *texture fetches*[27].

Una referencia a textura especifica qué parte de la memoria de textura es traída. Pueden existir varias referencias, las cuales pueden ser a la misma o a texturas solapadas. La referencia a textura tiene varios atributos, ellos son: la dimensionalidad (dependiendo de cómo va a ser referenciada: 1, 2 o 3 dimensiones), el tipo de los datos de entrada y salida de la textura, y la forma en que las coordenadas son interpretadas y hecho el procesamiento.

A igual que la memoria de constante, tiene una memoria caché *on-chip*. Esto significa accesos lentos cuando se producen fallos de caché. Además de la caché, el costo de acceso es distinto a las otras memorias, la memoria de texturas está optimizada para aprovechar la localidad espacial de dos dimensiones, desde el punto de vista de las imágenes es muy probable que si se accede a una parte de ésta, se lea la imagen completa. Esta propiedad permite obtener mejor rendimiento cuando los *threads* de un mismo *warp* acceden a direcciones cercanas de la memoria.

Realizar las lecturas a través de la memoria de texturas puede tener algunos beneficios que la convierte en una mejor alternativa a la memoria global o a la memoria constante, ellos son:

- Si las lecturas no se ajustan a los patrones de acceso específicos para la memoria global o la memoria de constantes, es posible obtener mayor ancho de banda explotando las ventajas de localidad en la memoria de texturas.
- La latencia producida por el cálculo de direcciones se oculta mejor, pudiendo mejorar el rendimiento de las aplicaciones con acceso aleatorio a los datos.
- Los datos pueden ser distribuidos a variables separadas en una única instrucción.
- Los enteros de 8 bits y 16 bits pueden ser convertidos a punto flotante de 32 bits (en rangos [0.0, 1.0] o [-1.0, 1.0]).

Pero no todo son ventajas, para trabajar con la memoria de texturas se deben dar varias condiciones, uno de ellas es trabajar con una estructura de datos específica con direccionamiento no lineal y de cálculo no trivial. Además para realizar cualquier tipo de operaciones en ella es necesario que todos los datos están presentes en la caché correspondiente. Estos dos factores hacen que se deba analizar muy bien el uso de la memoria de textura en la solución de cualquier aplicación.

4. Aceleración de los algoritmos hiperespectrales

Una vez explicados todos los conceptos teóricos, tanto de los algoritmos, PCA CE 6CP y PCA 1D15CE 8CP, como de la alternativa a utilizar, se procederá a descripción de los pasos llevados en cada lenguaje de programación para la aceleración de los distintos algoritmos. Por lo tanto, este capítulo estará formado por tres apartados. En el primero se explicará los pasos realizados para desarrollar los algoritmos en *MatLab*, en el segundo se desarrollará los algoritmos en C/C++ y para finalizar se explicará cómo se ha implementado en la alternativa elegida, es decir, en la GPU.

Para desarrollar estos algoritmos se ha basado en los archivos suministrados por AINIA que se componen de tres packs:

1. Modelo PCA CE 6CP: aquí se incluye un archivo .txt que contiene los espectros de los distintos cuerpos extraños llamado matriz, luego un archivo con los coeficientes de PCA, que en este caso se trata de 6 componentes principales, un Excel donde se encuentran los scores y las clases, que se utilizarán para la comprobación de pasos intermedios del algoritmo y un Word con los coeficientes y funciones del Análisis Discriminante.

2. Modelo PCA 1D15CE 8CP: como ocurre con el modelo anterior hay un archivo .txt que contiene los espectros de los distintos cuerpos extraños llamado matriz, luego un archivo que contiene los coeficientes de la primera derivada de Saviztky-Golay con 15 puntos de ventana, un archivo con los coeficientes de la matriz PCA, que en este caso se trata de 8 componentes principales, un Excel donde se encuentran los scores y las clases, que se utilizarán para la comprobación de pasos intermedios del algoritmo y un Word con los coeficientes y funciones del Análisis Discriminante.

3. Tomos de imágenes: aquí se encuentran tres tipos de ejemplares de muestra cárnica que dependiendo de la longitud de la muestra y su forma tendrá más o

menos imágenes. En la primera hay 887 imágenes, en la segunda 786 y en la última 876.

4.1 Implementación en Matlab

A continuación, se utilizará este lenguaje de programación por lo descrito en el apartado 1.3.1 Herramientas Utilizadas.

En este apartado se explicará lo realizado para los dos modelos en *MatLab*. En un primer lugar, se empezará por el modelo PCA CE 6CP y a continuación se seguirá con modelo PCA 1D15CE 8CP.

4.1.1 Modelo PCA CE 6CP

Este algoritmo estará formado por tres partes. La primera, lo formará el preprocesado que consistirá en un centrado y escalado. La segunda consistirá en aplicar PCA, análisis de principales componentes, con 6 componentes principales y la tercera será aplicar análisis discriminante.

Tras explicar las partes que contiene el algoritmo, el siguiente paso será aplicarlo sobre la matriz que contiene los espectros de los distintos cuerpos extraños caracterizados en vez de directamente sobre el tomo de imágenes. Esto se realizará para comprobar que los *scores* obtenidos tras aplicar PCA y la clasificación efectuada por el análisis discriminante hayan sido satisfactorios. Para ello, se comparará los resultados obtenidos con los valores de los archivos suministrados por AINIA.

Para aplicar el algoritmo PCA CE 6CP a la matriz de espectros, lo primero que se hará será cargar los archivos necesarios para su utilización que son: Matriz.txt; aquí se encuentran los espectros de los cuerpos extraños, PCA.txt; son los coeficientes para aplicar el análisis de principales componentes, coeficientes.txt y constantes.txt; son para aplicar la clasificación, es decir, el análisis discriminante. Para esto, se utilizará la función *load* de *MatLab*.

```
%% Carga espectros de los distintos cuerpos extraños
Matcamara =load('Matriz.txt','unicode');

%% PCA Análisis de las principales componentes
pca =load('PCA.txt');

%Coeficientes
coeficientes=load('coeficientes.txt');

%Constantes
constantes=load('constantes.txt');
```

A continuación, se calculará la media y varianza de los espectros de los distintos cuerpos extraños para cada longitud de onda, es decir, si la variable matcamara es la que recoge la información leída del fichero con los distintos espectros de los cuerpos extraños, matriz de 891 x 256, se calculará su media y varianza por columnas obteniendo al final dos vectores de 256 componentes.

```
media = mean(Matcamara);          %Obtengo la media para cada columna

varianza = var(Matcamara,1);    %Obtengo la varianza para cada columna
```

Con estas dos variables obtenidas lo que se hará es guárdalas en dos archivo de texto distintos y en un fichero .mat. Luego, se utilizará estas dos variables para aplicar el centrado y escalado a la matriz matcamara en primer lugar, espectros de los distintos cuerpos extraños caracterizados, y luego sobre el tomo de imágenes.

Para aplicar el centrado y escalado en la matriz matcamara se adaptará la fórmula 2.2 a las características de Matalb para implementarlo con el menor coste posible. Para ello, se recorrerá sus filas de una en una, 891, y se accederá a todas las columnas de una misma fila, 256 valores como se puede apreciar en el código siguiente.

```
for i=1:891
MatrizNueva(i,:)=(Matcamara(i,:)-mediaModelo)./sqrt(varianzaModelo);
end
```

Para acceder en una sola iteración a todo el contenido de la fila i, que contendrá los 256 espectros para un mismo pixel u objeto caracterizado, se utilizará el comando ":". Una vez obtenida la fila se le restará la media y se dividirá por la desviación estándar. Este conjunto de operaciones se le denomina centrado y escalado. Gracias a la

forma de trabajar de *MatLab* solo habrá que utilizar un bucle para realizar esta tarea que será recorrer las 891 filas.

Una vez aplicada esta técnica se tendrá la matriz centrada y escalda y lista para poder utilizar cualquier técnica estadística multivariante. Por lo tanto, el siguiente paso será aplicar análisis de principales componentes. En este caso será de seis principales componentes, que consistirá en multiplicar la matriz centrada y escalada por la matriz cargada denominada PCA y que fue suministrada por AINIA.

La matriz pca está compuesta por 256 filas y 6 columnas correspondientes al número de componentes principales utilizados y que contendrá los coeficientes para aplicar esta técnica.

A continuación, se puede ver como se ha aplicado en *MatLab* este método.

```
%% Multiplicación por PCA (Análisis de las principales componentes)
scores = MatrizNueva*pca;
```

Tras aplicar este método se obtendrá una matriz de 891 x 6, 891 corresponderá con los distintos puntos de los cuerpos extraños caracterizados y el seis con el número de componentes principales utilizados. Al resultado de esta operación se le llamará *scores*. Los valores de estos *scores* serán comparados con los *scores* suministrados por AINIA en el archivo excel llamado scores y clases para comprobar que todo el proceso llevado hasta este método hayan sido realizados correctamente. Si no es así habrá que revisar cuidadosamente todos los pasos llevados a cabo para que den los mismos valores.

Una vez obtenidos los valores correctos se procederá a aplicar el análisis discriminante para clasificar cada punto del cuerpo extraño caracterizado en una clase. Para ello, se multiplicará los valores obtenidos, *scores*, por la matriz de coeficientes que fueron cargados anteriormente y suministradas por Ainia en el documento Analisis_discriminante. La matriz de coeficientes estará formada por 6 filas y 10 columnas. El seis corresponde con cuantas componentes principales se estará utilizando en el algoritmo y el diez a la cantidad de clases que hay para clasificar los distintos

cuerpos extraños. Por lo tanto, la multiplicación de los scores con la matriz coeficientes hará que para cada clase, columnas, se le aplique una determinada función que viene determinada por el valor de los coeficientes para clasificar cada clase de forma diferentes como se puede ver en la Figura 9. Ejemplo de la función usada para clasificar el primer grupo. y que dará como resultado a una matriz de 891 x 10.
A continuación, se puede ver como se ha implementado esta multiplicación en *MatLab*.

```
%%  Análisis Discriminatorio (Clasificación por grupos)

analisis=scores*coeficientes;
```

Para completar la función del análisis discriminante para que clasifica se debe sumar unas constantes. Estas constantes han sido cargadas anteriormente del fichero constante y suministrado por Ainia en el documento Analisis_discriminante. Este fichero está formado por 10 valores, uno para cada clase. Por lo tanto, a cada fila, 891, que estará compuesto por 10 columnas se le sumará los valores del vector constantes cargado anteriormente como se puede comprobar en el siguiente código.

```
for i=1:891
    analisis(i,:)= analisis(i,:)+ constantes;
end
```

Ahora lo único que falta es clasificar cada fila en la clase correspondiente. Para ello, se buscará el máximo de cada fila, 891, y a la columna a la cual pertenezca ese máximo es a la clase que pertenecerá ese pixel de un cuerpo extraño. Para encontrar el máximo y su posición se utilizará el comando max de *MatLab* como se puede apreciar en el siguiente código.

```
for i=1:891
    [Valor,posicion]=max(analisis(i,:));
    clasificacion(i)=posicion;
    valores(i)=Valor;
end
```

Una vez realizado todo este proceso y comprobado que dan los mismos grupos clasificados para cada pixel se puede decir que el algoritmo ha sido implementado de forma satisfactoria y por lo tanto, se puede pasar a probar los distintos tomos de imágenes para comprobar su representación.

El siguiente paso será comprobar los disintos tomos de imágenes. Para ello, habrá que cargarlos previamente en *MatLab*. Cada tomo de imágenes posee 256 imágenes, una imagen para cada longitud de onda que captura la cámara.

```
%% Carga las imágenes
imagenes=cell(1,256);

for i=1:10
imagen=sprintf('K:\\20 febrero -
copia(Jesús)\\MatLab\\Lomo_Todo_1\\Img_00%d.tif',i-1);
Matcamara=imread(imagen);
Matcamara=double(Matcamara);
%[m, n]=size(Matcamara);
imagenes{i}=Matcamara;
end

for i=11:100
imagen=sprintf('K:\\20 febrero -
copia(Jesús)\\MatLab\\Lomo_Todo_1\\Img_0%d.tif',i-1);
Matcamara=imread(imagen);
Matcamara=double(Matcamara);
%[m, n]=size(Matcamara);
imagenes{i}=Matcamara;
end

for i=101:256
imagen=sprintf('K:\\20 febrero -
copia(Jesús)\\MatLab\\Lomo_Todo_1\\Img_%d.tif',i-1);
Matcamara=imread(imagen);
Matcamara=double(Matcamara);
%[m, n]=size(Matcamara);
imagenes{i}=Matcamara;
end
```

En el código anterior muestra cómo serán cargadas las distintas imágenes. Como se puede ver se realizará en tres bucles. Esto se deberá a la cantidad de ceros a la izquierda que tenga el nombre de la imagen, es decir, dos ceros; 001 hasta 009, un cero; 010 hasta 099, o ninguno cero; 100 hasta 256. Una vez cargadas las imágenes se tendrá un matriz tridimensional, como se puede ver en la Figura 37. Matriz Tridimensional., donde el ancho corresponderá con el ancho de las imágenes, en este caso 320, el largo vendrá dado por el alto de la imagen; 887 en el tomo 1, 786 en el tomo 2 y 876 en el tomo 3 y la profundidad corresponderá con el numero de imágenes cargadas que a su vez corresponderá con las distintas longitudes de ondas adquiridas, 256.

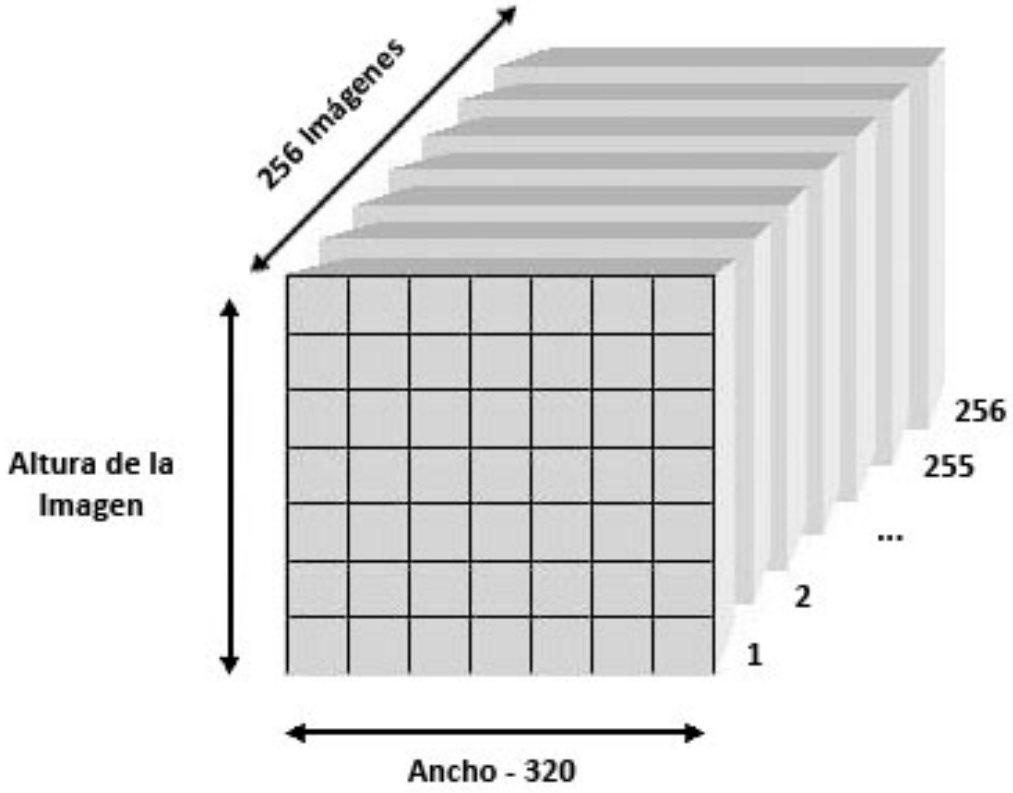


Figura 37. Matriz Tridimensional.

Cargadas las imágenes en forma de matriz tridimensional se procederá a cargar los archivos necesarios para aplicar el algoritmo PCA CE 6CP que serán: PCA, coeficientes, constantes, la media y la varianza que han sido calculadas y guardadas en el proceso anterior de comprobación.

A continuación, se procederá a aplicar el algoritmo PCA CE 6CP que consistirá en los siguientes pasos:

1. Se compondrá una fila de trabajo compuesto por 256 elementos. Esta fila se formará accediendo al mismo pixel para las distintas imágenes y guardando los distintos valores de los pixeles en un vector.
2. Con la fila obtenida en el paso 1 se le aplicará el centrado y escalado según la fórmula 2.2. En este paso se aprovechará las características de *MatLab* para realizar la resta por la media y la división por la desviación estándar de la forma más eficiente.
3. El resultado tras aplicar centrado y escalado se multiplicará con la matriz PCA. Con dicha multiplicación se estará aplicando el análisis de las principales

componentes, en este caso de seis. El resultado de esta multiplicación será un vector de seis elementos y al que se le denominará "scores".

4. Una vez obtenidos los "scores" se le aplicará análisis discriminante. Esto se realizará multiplicando los "scores" con la matriz coeficientes con lo que se obtendrá un vector de diez elementos. Cada valor de este vector corresponderá con el peso calculado por el análisis discrimínate para cada clase. A este resultado habrá que sumarles el archivo constantes que estará formado por un vector de diez componentes.
5. Tras calcular el análisis discriminante se buscará en qué posición se encuentra el máximo del vector obtenido en el paso anterior. Este máximo será el valor con el que se clasifique esta fila de trabajo, es decir, si se ha encontrado el máximo en la posición seis, ese pixel se clasificará con el valor seis.

Estos pasos se realizará para recorrer todos los pixeles de una imagen por lo que se efectuarán en dos bucles iterativos; uno para recorrer el ancho y otro para el largo de la imagen. Por lo tanto, como resultado final dará una matriz bidimensional de ancho como la imagen, 320, y de largo como la imagen tratada, en unos casos será 887, en otros 786 y otras veces 876.

A continuación se puede ver el código realizado y un esquema, Figura 38. Pasos seguidos en el algoritmo PCA CE 6CP, donde se ve la evolución seguida.

```
for i=1:887
    for j=1:320

           fila(h)=imagenes{h}(i,j); Obtengo fila de trabajo
        end
      %Aplico Centrado y escalado
        fila=(fila-mediaModelo)./sqrt(varianzaModelo);

      %Aplico PCA y obtengo una fila de scores
         scores=fila*pca;

      %Aplico AD
        analisis=scores*coeficientes;
        analisis=analisis+ constantes;

      %Analizo a que grupo pertenece
        [Valor,posicion]=max(analisis);

      %Obtengo una matriz con las distintas clasificaciones
        ImagenNueva(i,j)=posicion;

      end
end
```

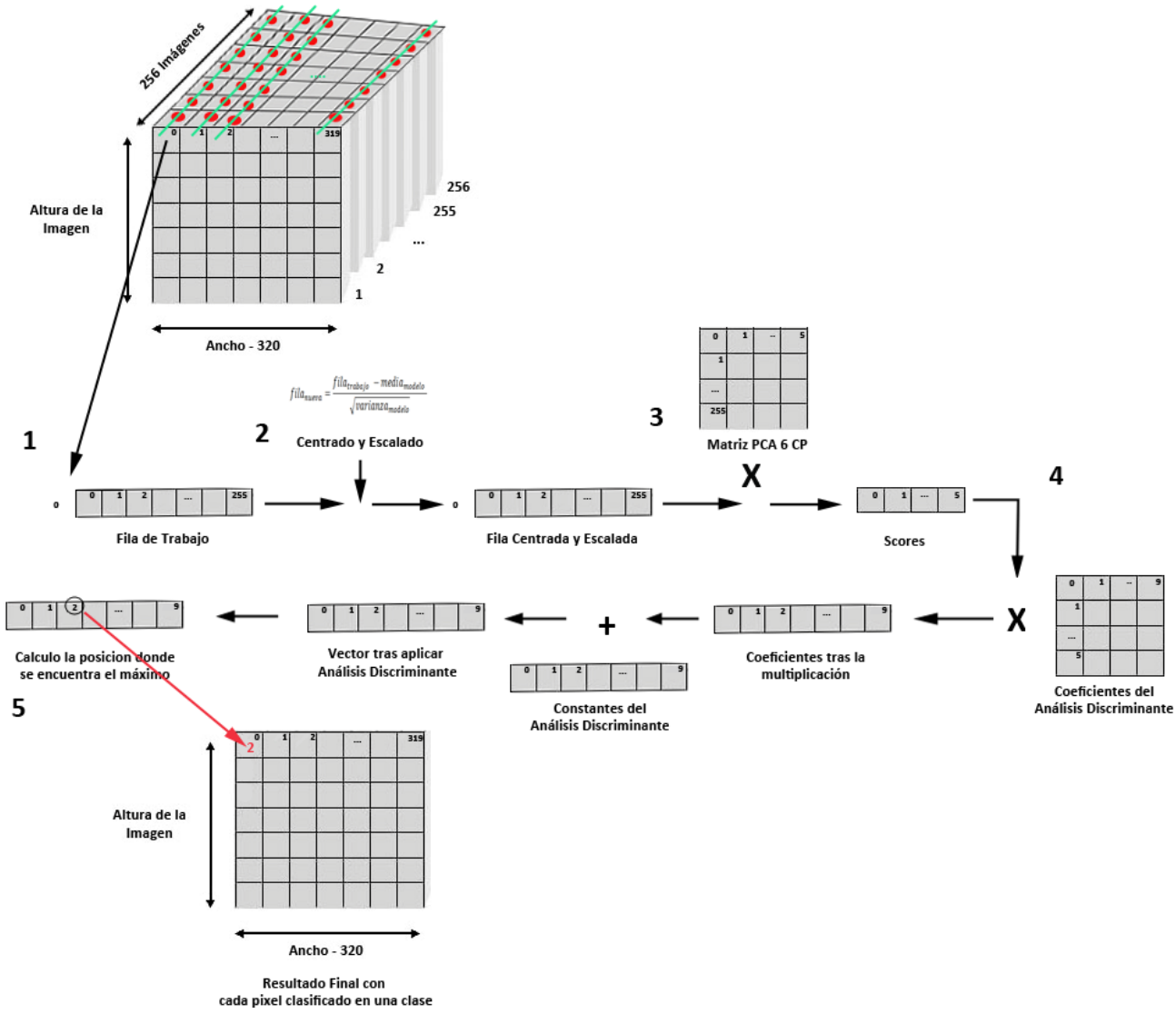


Figura 38. Pasos seguidos en el algoritmo PCA CE 6CP.

Una vez aplicado el algoritmo PCA CE 6 CP, se obtendrá como se ha comentado una matriz bidimensional o imagen donde cada valor de la matriz representará a que clase pertenece esa posición o ese pixel, se procederá a la visualización del resultado. Para ello, se creará una matriz tridimensional donde cada dimensión corresponderá con una de las matrices de color: rojo, verde y azul. Esta matriz rgb se conseguirá en *MaLab* utilizando el comando *zeros* que además permitirá inicializar la matriz a todo ceros. A continuación, se recorrerá la matriz bidimensional resultante del algoritmo PCA CE 6CP y en función de la clase a la que pertenezca ese pixel se le asignará un color u otro, es decir, si en la posición 4, que corresponde con la fila 1 y columna 4, la clase a la que pertenece ese pixel es el 3 y a la clase 3 se le ha asignado el color rojo, en la matriz rgb se escribirá (1,0,0) que corresponderá con dicho color. Luego, para recorrer la matriz

bidimensional se utilizará dos bucles y con un *switch* se escribirá el valor del color correspondiente en la matriz rgb como se puede comprobar en el código siguiente.

```
rgb=zeros(m,n,3);

for i=1:m
    for j=1:n
       switch ImagenNueva(i,j)
            case 1
                rgb(i,j,1)=0;
                rgb(i,j,2)=1;
                rgb(i,j,3)=0;

            case 2
                rgb(i,j,1)=1;
                rgb(i,j,2)=1;
                rgb(i,j,3)=0;

            case 3
                rgb(i,j,1)=0.55;
                rgb(i,j,2)=0.35;
                rgb(i,j,3)=0;

            case 4
                rgb(i,j,1)=0.5;
                rgb(i,j,2)=0.5;
                rgb(i,j,3)=0.5;

            case 5
                rgb(i,j,1)=0;
                rgb(i,j,2)=0;
                rgb(i,j,3)=1;

            case 6
                rgb(i,j,1)=1;
                rgb(i,j,2)=1;
                rgb(i,j,3)=1;

            case 7
                rgb(i,j,1)=1;
                rgb(i,j,2)=0;
                rgb(i,j,3)=0;

            case 8
                rgb(i,j,1)=0.1;
                rgb(i,j,2)=0.4;
                rgb(i,j,3)=0;

            case 9
                rgb(i,j,1)=0.85;
                rgb(i,j,2)=0.35;
                rgb(i,j,3)=0;

           case 10
                rgb(i,j,1)=0;
                rgb(i,j,2)=0;
                rgb(i,j,3)=0;
       end
```

```
    end
end

imwrite(rgb, 'lomo1.jpg');
```

El siguiente paso será representar la matriz rgb. Así pues, se utilizará el comando *imwrite* de *MatLab* que permitirá convertir la matriz rgb en una archivo de imagen. Este comando utiliza dos parámetros, uno la variable a representar y la otra el nombre del archivo de imagen incluida su extensión.

Después de aplicar este comando se obtendrá la imagen química procesada. Tras aplicarlo en un tomo se volverá a ejecutar el algoritmo pero para los otros dos tomos con la única modificación de sus rutas a la hora de cargar cada una de las imágenes. Por lo tanto, el resultado final de los tres tomos serán tres imágenes que se observan en la Figura 39. Resultado tras aplicar PCA CE 6 CP..

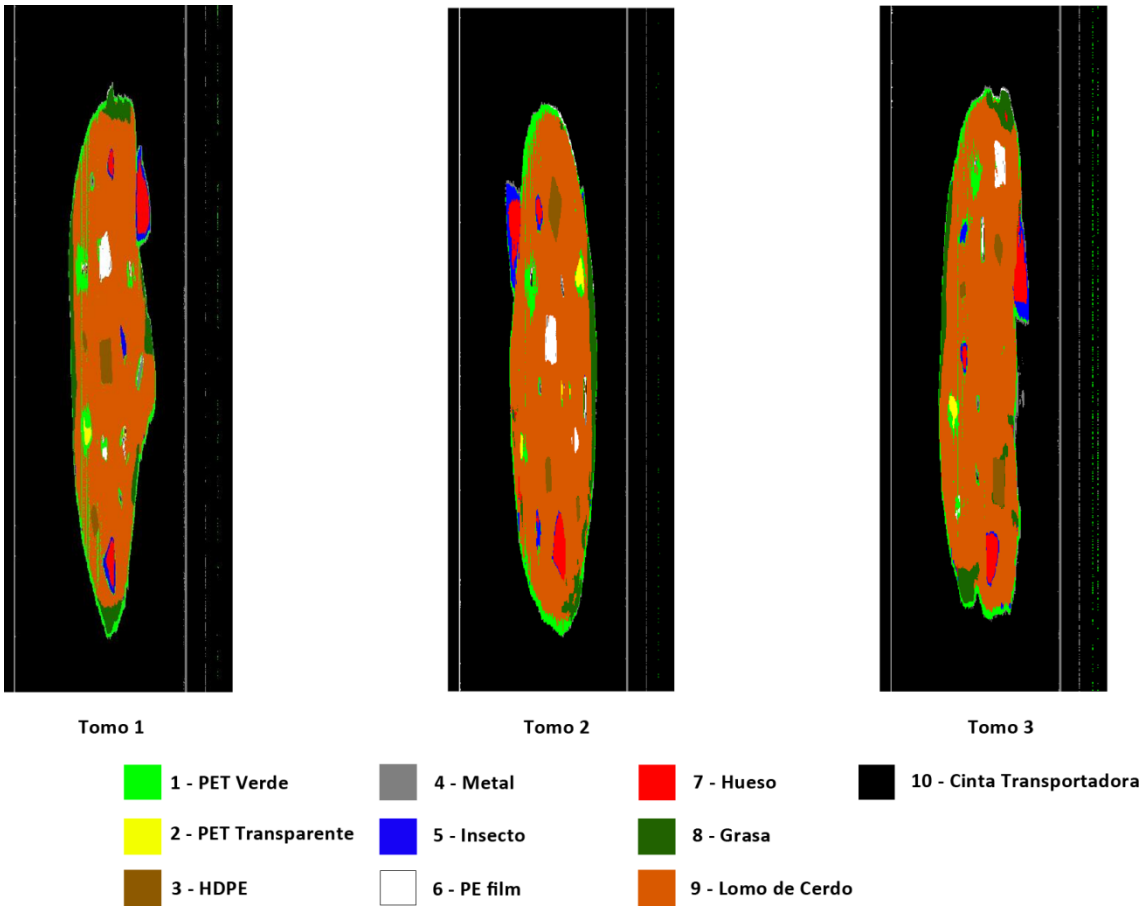


Figura 39. Resultado tras aplicar PCA CE 6 CP.

4.1.2 Modelo PCA 1D15CE 8CP

Este modelo estará formado por tres partes. La primera, lo formará el preprocesado que consistirá en la primera derivada de Saviztky-Golay de orden 2 con una ventana de quince puntos seguido de un centrado y escalado. La segunda consistirá en aplicar PCA, análisis de principales componentes, con 8 componentes principales y la tercera será aplicar análisis discriminante.

Una vez descrito las partes en que está formado este algoritmo se explicará los pasos que se llevarán a cabo. En primer lugar se aplicará el modelo PCA 1D15CE 8CP sobre la matriz que contiene los espectros de los distintos cuerpos extraños. Este paso se realizará para comparar los resultados obtenidos tras aplicar PCA con los valores suministrados por AINIA en el archivo "scores".

Antes de aplicar el algoritmo PCA 1D15CE 8CP se deberán cargar los archivos necesarios para poder aplicar el modelo PCA 1D15 8CP y que serán: Matriz.txt, donde se encuentran los espectros de los cuerpos extraños clasificados, PCA.txt, este archivo contendrá los coeficientes para aplicar el análisis de ocho principales componentes, coeficientes.txt y constantes.txt, tendrán las variables dependientes como independientes con los cuales se aplicará el análisis discriminante y Mat_SG_1D51.txt, que contendrá los coeficientes con los que se aplicará la primera derivada de Saviztky-Golay. Para cargar estos archivos se utilizará el comando load como se puede ver en el código siguiente.

```
%% Carga espectros de los distintos cuerpos extraños
Matcamara =load('Matriz.txt','unicode');

%% PCA Análisis de las principales componentes
pca =load('PCA.txt');

%Coeficientes
coeficientes=load('coeficientes.txt');

%Constantes
constantes=load('constantes.txt');

%% Carga de la matriz Derivada
deriv=load('Mat_SG_1D51.txt');
```

Una vez cargados los archivos necesarios se procederá a aplicar a la variable Matcamara, matriz con los distintos espectros de los diferentes cuerpos extraños tratados, la primera derivada de Saviztky-Golay. Este paso se realizará multiplicando los coeficientes de la primera derivada con los valores que se encuentran en el archivo Matriz.txt, es decir, se multiplicará la variable Matcamara con deriv como se puede apreciar a continuación.

```
% 1 Derivada S.G

Matcamara=Modelo*deriv;
```

Como resultado se tendrá una matriz con los espectros de los distintos cuerpos filtrados y suavizados de 891 filas por 256 columnas. Cada fila corresponderá con los distintos puntos de los diferentes cuerpos extraños seleccionados y las columnas representarán una longitud de onda distinta.

Una vez se haya aplicado la primera derivada de Saviztky-Golay el siguiente paso será calcular su media y varianza por columnas, es decir, si en la variable Matcamara está el resultado de aplicar la primera derivada de Saviztky-Golay, matriz de 891 x 256, se calculará tanto la media como la varianza para cada longitud de onda. Este cálculo se realizará utilizando el comando mean y var de *MatLab* como se pude comprobar en el siguiente código.

```
media1=mean(Matcamara);          %Obtengo la media para cada columna
varianza1= var(Matcamara,1);  %Obtengo la varianza para cada columna
```

Estos dos resultados serán dos vectores de 256 elementos que se guardarán en dos variables y en dos archivos .txt para su posterior utilización.

El siguiente paso será aplicar estas dos variables calculadas en el resultado de la primera derivada de Saviztky-Golay mediante un centrado y escalado, es decir, al resultado de la derivada se le restara elemento a elemento la media y se dividirá por la desviación estándar como muestra el código siguiente.

```
for i=1:891
MatrizNueva(i,:)=(Matcamara(i,:)-mediaModelo)./sqrt(varianzaModelo);
end
```

Una vez aplicada esta técnica se tendrá la matriz centrada y escalda y lista para poder utilizar cualquier técnica estadística multivariante. Luego el siguiente paso será

aplicar análisis de principales componentes. En este caso será de ocho principales componentes, que consistirá en multiplicar el resultado de aplicar centrado y escalado por la matriz cargada denominada PCA y que estará compuesta por 256 filas, una para cada longitud de onda y por 8 columnas que corresponderán con el número de componentes principales utilizados.

A continuación, se puede ver como se ha aplicado en *MatLab* esta técnica.

```
%% Multiplicación por PCA (Análisis de las principales componentes)
scores = MatrizNueva*pca;
```

Tras aplicar este método se obtendrá una matriz de 891 x 8, 891 corresponderá con los distintos puntos de los cuerpos extraños caracterizados y el ocho con el número de componentes principales utilizados. Al resultado de esta operación se le llamará *scores*. Los valores de estos *scores* serán comparados con los *scores* suministrados por AINIA en el archivo excel llamado scores y clases para comprobar que todo el proceso llevado hasta este método hayan sido realizados correctamente. Si no es así habrá que revisar cuidadosamente todos los pasos llevados a cabo para que den los mismos valores.

Una vez obtenidos los valores correctos se procederá a aplicar el análisis discriminante para clasificar cada punto del cuerpo extraño caracterizado en una clase. Para ello, se multiplicará los valores obtenidos, *scores*, por la matriz de coeficientes que fueron cargados anteriormente y suministradas por Ainia en el documento Analisis_discriminante. La matriz de coeficientes estará formada por 8 filas y 10 columnas. El ocho corresponde con cuantas componentes principales se estará utilizando en el algoritmo y el diez a la cantidad de clases que hay para clasificar los distintos cuerpos extraños. Por lo tanto, la multiplicación de los scores con la matriz coeficientes hará que para cada clase, columnas, se le aplique una determinada función que viene determinada por el valor de los coeficientes para clasificar cada clase de forma diferentes como se puede ver en la Figura 9. Ejemplo de la función usada para clasificar el primer grupo. y que dará como resultado a una matriz de 891 x 10.

A continuación, se puede ver como se ha implementado esta multiplicación en *MatLab*.

```
%%  Análisis Discriminatorio (Clasificación por grupos)

analisis=scores*coeficientes;
```

Para finalizar el análisis discriminante se deberán sumar unas constantes. Estas constantes han sido cargadas anteriormente del fichero constante y suministrado por Ainia en el documento Analisis_discriminante. Este fichero está formado por 10 valores, uno para cada clase. Por lo tanto, a cada fila, 891, que estará compuesto por 10 columnas se le sumará los valores del vector constantes cargado anteriormente como se puede comprobar en el siguiente código.

```
for i=1:891
    analisis(i,:)= analisis(i,:)+ constantes;
end
```

Ahora lo único que falta es clasificar cada fila en la clase correspondiente. Para ello, se buscará el máximo de cada fila, 891, y a la columna a la cual pertenezca ese máximo es a la clase que pertenecerá ese pixel de un cuerpo extraño. Para encontrar el máximo y su posición se utilizará el comando max de *MatLab* como se puede apreciar en el siguiente código.

```
for i=1:891
    [Valor,posicion]=max(analisis(i,:));
    clasificacion(i)=posicion;
    valores(i)=Valor;
end
```

Una vez realizado todo este proceso y comprobado que dan los mismos grupos clasificados para cada pixel se puede decir que el algoritmo ha sido implementado de forma satisfactoria y por lo tanto, se puede pasar a probar los distintos tomos de imágenes para comprobar su representación.

El siguiente paso será comprobar los disintos tomos de imágenes. Para ello, habrá que cargarlos previamente en *MatLab*. Cada tomo de imágenes posee 256 imágenes, una imagen para cada longitud de onda que captura la cámara.

```
%% Carga las imágenes
imagenes=cell(1,256);

for i=1:10
imagen=sprintf('K:\\20 febrero -
copia(Jesús)\\MatLab\\Lomo_Todo_1\\Img_00%d.tif',i-1);
Matcamara=imread(imagen);
Matcamara=double(Matcamara);
%[m, n]=size(Matcamara);
imagenes{i}=Matcamara;
end

for i=11:100
imagen=sprintf('K:\\20 febrero -
copia(Jesús)\\MatLab\\Lomo_Todo_1\\Img_0%d.tif',i-1);
Matcamara=imread(imagen);
Matcamara=double(Matcamara);
%[m, n]=size(Matcamara);
imagenes{i}=Matcamara;
end

for i=101:256
imagen=sprintf('K:\\20 febrero -
copia(Jesús)\\MatLab\\Lomo_Todo_1\\Img_%d.tif',i-1);
Matcamara=imread(imagen);
Matcamara=double(Matcamara);
%[m, n]=size(Matcamara);
imagenes{i}=Matcamara;
end
```

En el código anterior muestra cómo serán cargadas las distintas imágenes. Como se puede ver se realizará en tres bucles. Esto se deberá a la cantidad de ceros a la izquierda que tenga el nombre de la imagen, es decir, dos ceros; 001 hasta 009, un cero; 010 hasta 099, o ninguno cero; 100 hasta 256. Una vez cargadas las imágenes se tendrá un matriz tridimensional, como se puede ver en la Figura 37. Matriz Tridimensional.40, donde el ancho corresponderá con el ancho de las imágenes, en este caso 320, el largo vendrá dado por el alto de la imagen; 887 en el tomo 1, 786 en el tomo 2 y 876 en el tomo 3 y la profundidad corresponderá con el numero de imágenes cargadas que a su vez corresponderá con las distintas longitudes de ondas adquiridas, 256.

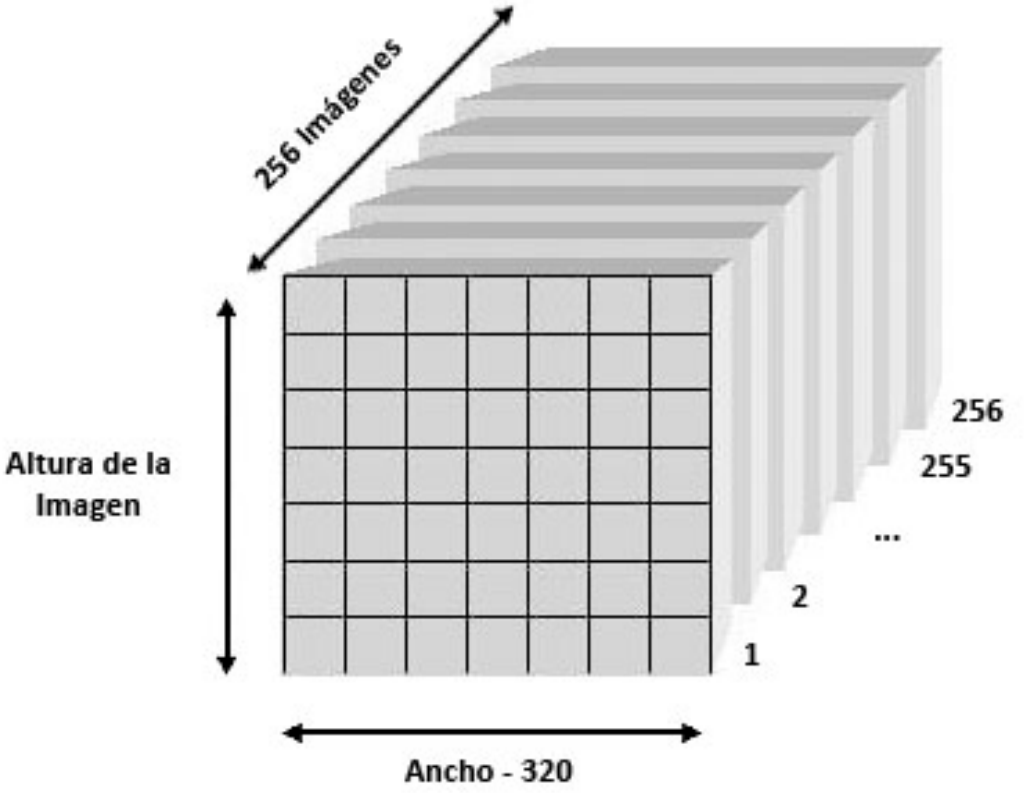


Figura 40.Matriz Tridimensional.

Cargadas las imágenes de un tomo en forma de matriz tridimensional se procederá a cargar los archivos necesarios para aplicar el algoritmo PCA 1D15CE 8CP que serán: PCA, coeficientes, constantes, derivada, la media y la varianza que han sido calculadas y guardadas en el proceso anterior de comprobación.

A continuación, se procederá a aplicar el algoritmo PCA 1D15CE 8CP que consistirá en los siguientes pasos:

1. Se compondrá una fila de trabajo compuesto por 256 elementos. Esta fila se formará accediendo al mismo pixel para las distintas imágenes y guardando los distintos valores de los pixeles en un vector.
2. Con la fila obtenida en el paso 1 se le aplicará la primera derivada de Saviztky-Golay. Esto se realizará multiplicando la fila del paso 1 por la matriz con los coeficientes de la primera derivada de Saviztky-Golay obteniendo un vector de 256 elementos.
3. Al resultado del paso anterior se le aplicará el centrado y escalado según la fórmula 2.2. En este paso se aprovechará las características de *MatLab* para realizar la resta por la media y la división por la desviación estándar de la forma más eficiente.

4. El resultado tras aplicar centrado y escalado se multiplicará con la matriz PCA. Con dicha multiplicación se estará aplicando el análisis de las principales componentes, en este caso de ocho. El resultado de esta multiplicación será un vector de ocho elementos y al que se le denominará "scores".
5. Una vez obtenidos los "scores" se le aplicará análisis discriminante. Esto se realizará multiplicando los "scores" con la matriz coeficientes con lo que se obtendrá un vector de diez elementos. Cada valor de este vector corresponderá con el peso calculado por el análisis discrimínate para cada clase. A este resultado habrá que sumarles el archivo constantes que estará formado por un vector de diez componentes.
6. Tras calcular el análisis discriminante se buscará en qué posición se encuentra el máximo del vector obtenido en el paso anterior. Este máximo será el valor con el que se clasifique esta fila de trabajo, es decir, si se ha encontrado el máximo en la posición seis, ese pixel se clasificará con el valor seis.

Estos pasos se realizará para recorrer todos los pixeles de una imagen por lo que se efectuarán en dos bucles iterativos; uno para recorrer el ancho y otro para el largo de la imagen. Por lo tanto, como resultado final dará una matriz bidimensional de ancho como la imagen, 320, y de largo como la imagen tratada, en unos casos será 887, en otros 786 y otras veces 876.

A continuación se puede ver el código realizado y un esquema, Figura 38. Pasos seguidos en el algoritmo PCA CE 6CP, donde se ve la evolución seguida.

```
for i=1:m
    for j=1:n
        for h=1:t
           fila(h)=imagenes{h}(i,j);%Obtengo cada pixel de cada imagen
        end
%Aplico primera derivada de S.G orden 2 y con una ventana de 15 puntos
        d=fila*deriv;
        fila=d;
%Aplico Centrado y escalado
        fila=(fila-mediaModelo)./sqrt(varianzaModelo);
%Aplico PCA y obtengo una fila de scores
         scores=fila*pca;
%Aplico AD
         analisis=scores*coeficientes;
         analisis=analisis+ constantes;
%Analizo a que grupo pertenece
        [Valor,posicion]=max(analisis);
%Obtengo una matriz con las distintas clasificaciones
        ImagenNueva(i,j)=posicion;
```

```
    end
end
```

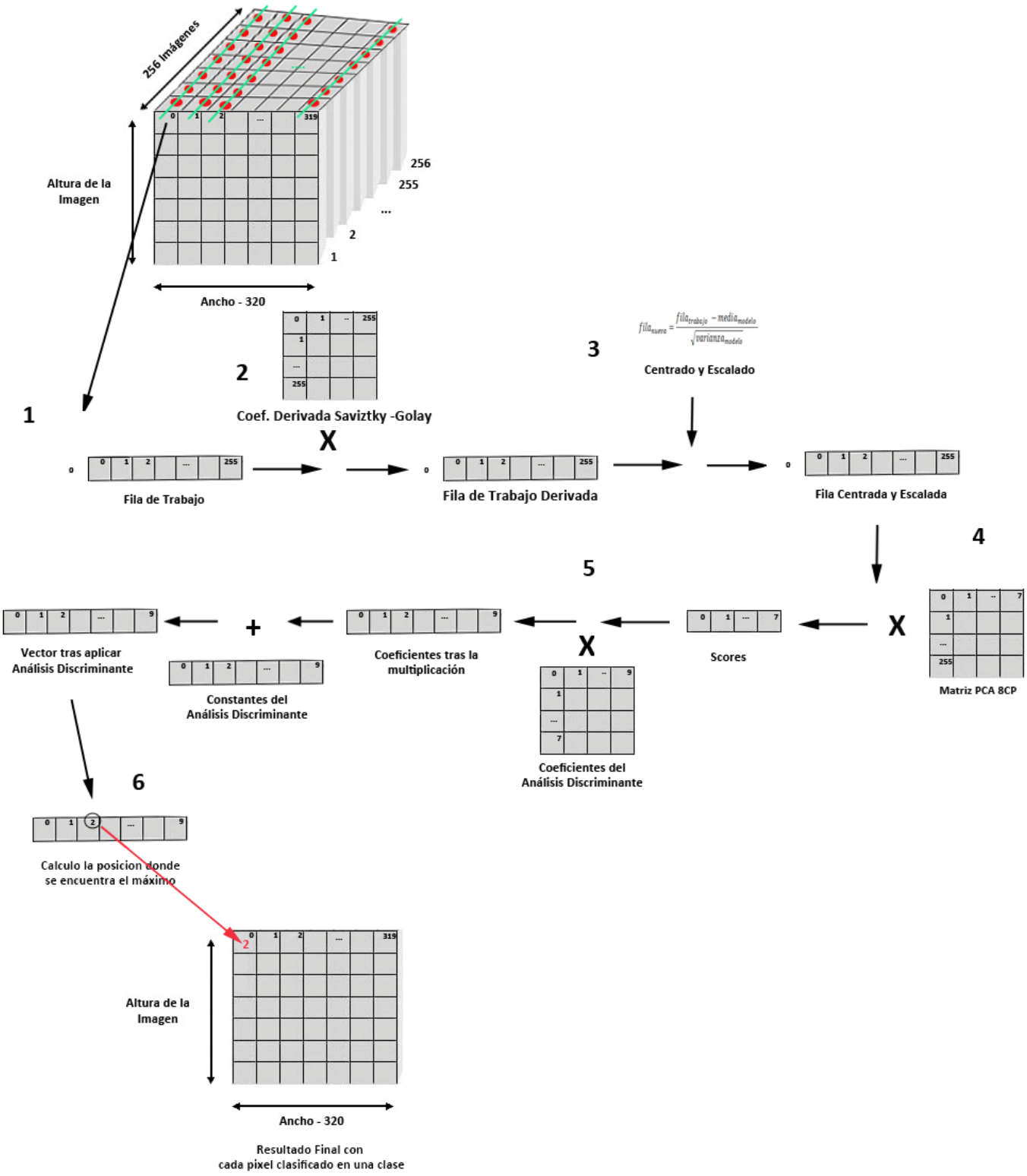


Figura 41. Pasos seguidos en el algoritmo PCA 1D15CE 8CP.

Una vez aplicado el algoritmo PCA 1D15CE 8CP, se obtendrá como se ha comentado una matriz bidimensional o imagen donde cada valor de la matriz representará a que clase pertenece esa posición o ese pixel, se procederá a la visualización del resultado. Para ello, se creará una matriz tridimensional donde cada dimensión corresponderá con una de las matrices de color: rojo, verde y azul. Esta matriz rgb se conseguirá en *MatLab* utilizando el comando *zeros* que además permitirá inicializar la matriz a todo ceros. A continuación, se recorrerá la matriz bidimensional resultante del algoritmo PCA 1D15CE 8CP y en función de la clase a la que pertenezca ese pixel se le asignará un color u otro, es decir, si en la posición 7, que corresponde con la fila 1 y columna 7, la clase a la que pertenece ese pixel es el 5 y a la clase 5 se le ha asignado el color azul, en la matriz rgb se escribirá (0,0,1) que corresponderá con dicho color. Luego, para recorrer la matriz bidimensional se utilizará dos bucles y con un *switch* se escribirá el valor del color correspondiente en la matriz rgb como se puede comprobar en el código siguiente.

```
rgb=zeros(m,n,3);

for i=1:m
    for j=1:n
       switch ImagenNueva(i,j)
            case 1
                rgb(i,j,1)=0;
                rgb(i,j,2)=1;
                rgb(i,j,3)=0;

            case 2
                rgb(i,j,1)=1;
                rgb(i,j,2)=1;
                rgb(i,j,3)=0;

            case 3
                rgb(i,j,1)=0.55;
                rgb(i,j,2)=0.35;
                rgb(i,j,3)=0;

            case 4
                rgb(i,j,1)=0.5;
                rgb(i,j,2)=0.5;
                rgb(i,j,3)=0.5;

            case 5
                rgb(i,j,1)=0;
                rgb(i,j,2)=0;
                rgb(i,j,3)=1;

            case 6
                rgb(i,j,1)=1;
                rgb(i,j,2)=1;
                rgb(i,j,3)=1;
```

```
            case 7
                rgb(i,j,1)=1;
                rgb(i,j,2)=0;
                rgb(i,j,3)=0;

            case 8
                rgb(i,j,1)=0.1;
                rgb(i,j,2)=0.4;
                rgb(i,j,3)=0;

            case 9
                rgb(i,j,1)=0.85;
                rgb(i,j,2)=0.35;
                rgb(i,j,3)=0;

      case 10
                rgb(i,j,1)=0;
                rgb(i,j,2)=0;
                rgb(i,j,3)=0;
       end
    end
end

imwrite(rgb, 'lomo1SG.jpg');
```

El siguiente paso será representar la matriz rgb. Así pues, se utilizará el comando *imwrite* de *MatLab* que permitirá convertir la matriz rgb en una archivo de imagen. Este comando utiliza dos parámetros, uno la variable a representar y la otra el nombre del archivo de imagen incluida su extensión.

Después de aplicar este comando se obtendrá la imagen química procesada. Tras aplicarlo en un tomo se volverá a ejecutar el algoritmo pero para los otros dos tomos con la única modificación de sus rutas a la hora de cargar cada una de las imágenes. Por lo tanto, el resultado final de los tres tomos serán tres imágenes que se observan en la Figura 42. Resultado tras aplicar PCA 1D15CE 8 CP en MatLab.Figura 39. Resultado tras aplicar PCA CE 6 CP.

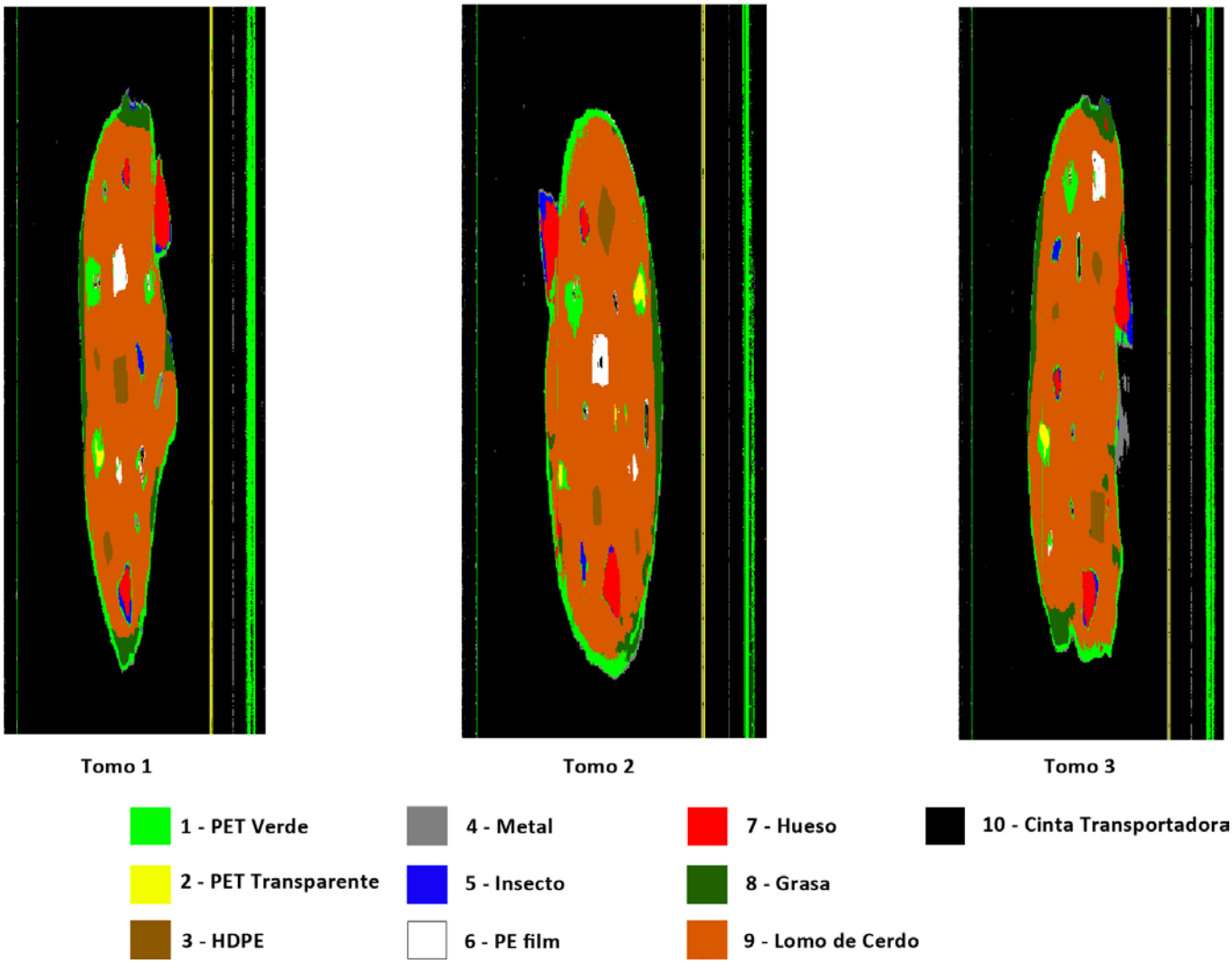


Figura 42. Resultado tras aplicar PCA 1D15CE 8 CP en MatLab.

4.2 Implementación en C++

Una vez realizado y entendidos los algoritmos en *MatLab* se realizará en C++. Ya que este lenguaje es uno de los más comunes y utilizado para realizar software de sistemas y aplicaciones en tiempo real. Además combina la flexibilidad y el acceso de bajo nivel de C con las características de la programación orientada a objetos como abstracción, encapsulación y ocultación, permitiéndome realizar un código eficiente. Asimismo el empleo de C++ permite la integración de bibliotecas ya existentes, al encontrarse muchas de ellas implementadas en dicho lenguaje de programación, con lo que ayudará en buena medida. Como puede ser el caso de la biblioteca OpenCV que realizará la parte de lectura y representación de las imágenes.

El hecho de realizarlo también en este leguaje que es secuencial, en el cual se realizará todo el proceso en la CPU, permitirá más tarde una comparativa de tiempos entre los distintos lenguajes y sobre todo con la alternativa elegida, la GPU, que realizará una parte en la CPU y otra en la GPU en un lenguaje paralelo denominado CUDA.

Luego en este apartado se explicará lo desarrollado para los dos modelos en el lenguaje de programación C++. En un primer lugar se empezará por el modelo PCA CE 6CP y a continuación con el modelo PCA 1D15Ce 8CP.

4.2.1 Modelo PCA CE 6 CP

Este algoritmo estará formado por tres partes. La primera, lo formará el preprocesado que consistirá en un centrado y escalado. La segunda consistirá en aplicar PCA, análisis de principales componentes, con 6 componentes principales y la tercera será aplicar análisis discriminante.

Una vez descrito las partes en que está formado este algoritmo se explicará los pasos que se llevarán a cabo en este lenguaje de programación, C++. A diferencia del apartado 4.1.1 Modelo PCA CE 6CP no se aplicará el modelo PCA CE 6 CP sobre la matriz que contiene los espectros de los distintos cuerpos extraños. Este paso no se

realizará debido a que en el lenguaje anterior, apartado 4.1 Implementación en Matlab, se entendido y desarrollado correctamente obteniéndose resultados iguales al archivo "scores" y los archivos necesarios para el desarrollo del modelo PCA CE 6 CP, media y varianza. Por lo tanto, este paso en este lenguaje se ha considerado innecesarios y el modelo PCA CE 6CP se aplicará directamente sobre uno de los tomos de imágenes.

El desarrollo de este modelo en C++ estará formado por tres archivos. Uno el programa principal, el segundo archivo denominado imágenes donde estará todas las funciones relacionados con las imágenes, como son lectura de imagen, columnas que posee una imagen, filas que posee una imagen y visualización de una imagen. Y el tercer archivo denominado proceso donde estará todas las funciones relacionadas con el algoritmo, como son lecturas de los archivos necesarios, visualización por pantalla de los contenidos de los archivos, reserva de memorias, liberación de memoria y el modelo PCA CE 6CP. A continuación, se describirá con más detalles todos los pasos llevados para el desarrollo del modelo PCA CE 6CP.

En primer lugar, se definirán las rutas y las variables para leer los distintos archivos necesarios para la aplicación del modelo PCA CE 6CP. Para leer los distintos archivos dependerá si el archivo leído es un vector o una matriz. Los archivos media, varianza y constantes son vectores por lo que se utilizará la función leervector mientras que los archivos PCA y coeficientes son matrices que se leerán con la función leermatriz del archivo procesos. Como se sabe de qué tamaño son los vectores y las matrices lo que se hace en cada una de las funciones, dependiendo de si es vector o matriz, es ir leyendo por espacios hasta completar el tamaño deseado del vector o matriz. Esto se realizará utilizando la biblioteca iostream de C++. Ver anexo.

Una vez se tenga cargado los archivos que se necesitarán para el algoritmo PCA CE 6CP el siguiente paso será cargar las imágenes de unos de los tomos que hay. Para este cometido se utilizará la librería OpenCV (1.3.1 Herramientas Utilizadas). Luego, para la lectura de una imagen se utilizará la función Cargaimagenes dentro del archivo imágenes y dentro de esta se utilizara el comando *cvLoadImage* de OpenCV cuyo objetivo es cargar una imagen desde un archivo especificado y devolver el puntero a la imagen cargada. Como parámetros de entrada tendrá un *filename* que será el nombre de la carpeta cargada e *iscolor* que especifica el color dado a la imagen cargada siendo mayor de 0 para que la imagen cargada se vea en 3 canales de imagen RGB, 0 para que

la imagen cargada se vea en escala de grises y menor de 0 para cargar la imagen tal y como esta. Siendo este ultimo de nuestro interés para analizar las imágenes hiperespectrales tal como se puede apreciar en el código siguiente.

```
IplImage* imagen=NULL;
imagen=cvLoadImage(ruta,-1);
```

Los archivos compatibles con esta función son: Windows mapas de bits (BMP,DIB), JPEG (JPEG,JPG,JPE), Gráficos de red portátiles (PNG), Formato de imagen Portable (PBM,PGM,PPM), Sun rasters (SR, RAS) y TIFF (TIFF, TIF).

Una vez leída la imagen el siguiente paso será recorrer pixel a pixel para obtener sus valores y guardarlo en una matriz con el que se pueda trabajar de forma más cómoda y rápida. Para ello, se utilizará el comando *cvGet2D* de la librería OpenCV, que devuelve un elemento concreto de la imagen especificada. Como parámetros de entrada tendrá la imagen cargada mediante *cvLoadImage* y luego tendrá dos variables enteras, la primera corresponde con las filas y la segunda con las columnas, que definirán a que pixel en concreto se accede como se puede comprobar a continuación.

```
s=cvGet2D(imagen,fila,columna);
```

Luego se guardará el valor devuelto por el comando *cvGet2D* en una matriz bidiminesional convirtiendo el valor *double* devuelto por el comando *cvGet2D* a *float.* Este paso se hará simplemente igualando el valor devuelto al elemento correspondiente en la matriz como se puede observar a continuación.

```
imag[fila][columna]=(float)s.val[0];
```

Estas dos acciones anteriores se realizarán para todos los pixeles de la imagen. Por lo tanto, se recorrerá la imagen completa utilizando dos bucles uno para recorrer las filas y otro las columnas como se puede comprobar en la siguiente figura.

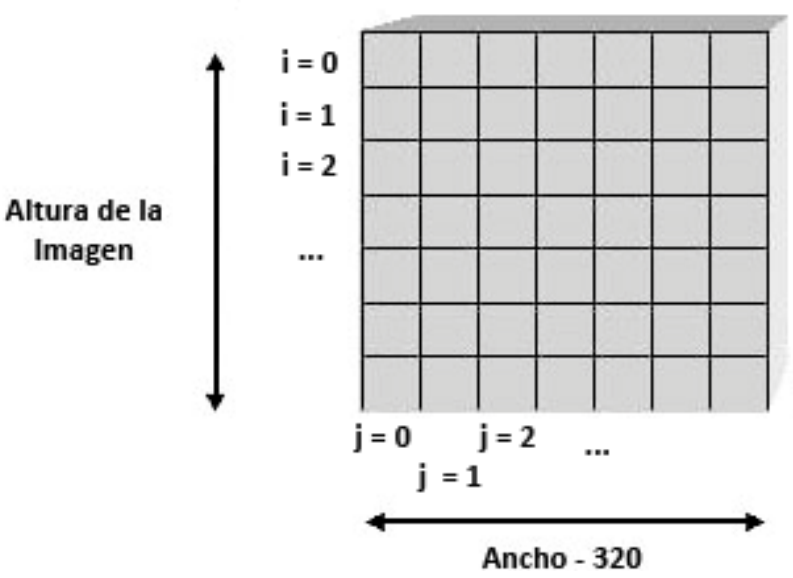


Figura 43. Recorre cada pixel de una imagen hiperespectral mediante cvGet2D.

Una vez explicado cómo se leerá una imagen y se guardará en una matriz se procederá explicar cómo se leerán y guardarán las 256 imágenes hiperespectrales. Esto se realizará leyendo una imagen como se ha comentado anteriormente y guardándola en la matriz bidimensional. A continuación, se copiará los valores guardados de esa imagen en una matriz tridimensional creada para guardar las 256 imágenes hiprespectrales. Todo este proceso se realizará en un bucle para cada imagen. Por lo tanto, habrá tres bucles uno para recorrer las filas de una imagen, el segundo para recorrer las columnas de una imagen y el tercero para recorrer las distintas imágenes.

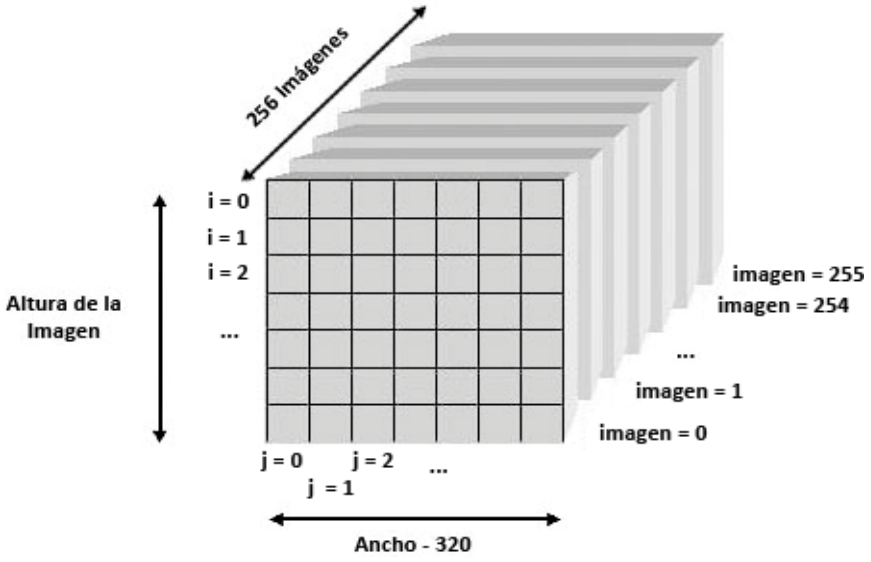


Figura 44. Matriz tridimensional donde se muestran los tres bucles a recorrer.

Una vez cargadas todas las imágenes de uno de los tomos en la matriz tridimensional se procederá a aplicar el algoritmo PCA CE 6CP mediante la función CracionImagen del archivo proceso.

1. En primer lugar se compondrá una fila de trabajo compuesto por 256 elementos. Esta fila se formará accediendo al mismo pixel para las distintas imágenes. Este proceso se realizará en un bucle accediendo adecuadamente a la matriz tridimensional. A la vez que se accede a un elemento de la imagen se le aplicará el centrado y escalado según la fórmula 2.2. Además de aplicarle el centrado y escalado se multiplicará por el correspondiente coeficiente de la matriz PCA, es decir, se multiplicará el pixel accedido en esa instante por el coeficiente que corresponde con la fila igual al número de imagen accedido y la columna será igual al valor del otro bucle anidado para poder recorrer las seis componentes principales o mejor dicho las columnas de la matriz PCA. Mientras se esté en el bucle de la fila de trabajo, es decir, el primer bucle se irán sumando todas las multiplicaciones que se vayan haciendo con PCA. Al final de este bucle gracias a estas sumas se tendrá el *score* que corresponderá con la iteración con la que se esté en el segundo bucle.

   Todo este proceso se hará en dos bucles en vez de tener que repetir bucles con el mismo número de iteraciones y relacionados entre sí. Por ejemplo, no tiene sentido realizar un bucle simplemente para formar la fila de trabajo y a continuación otro bucle con el mismo número de iteraciones que realice el centrado y escalado. Por esta razón todos los bucles que se han podido unir se han realizado quedándose en dos bucles solamente.

   A continuación se muestra una figura y un pseudocódigo que ilustra lo realizado.

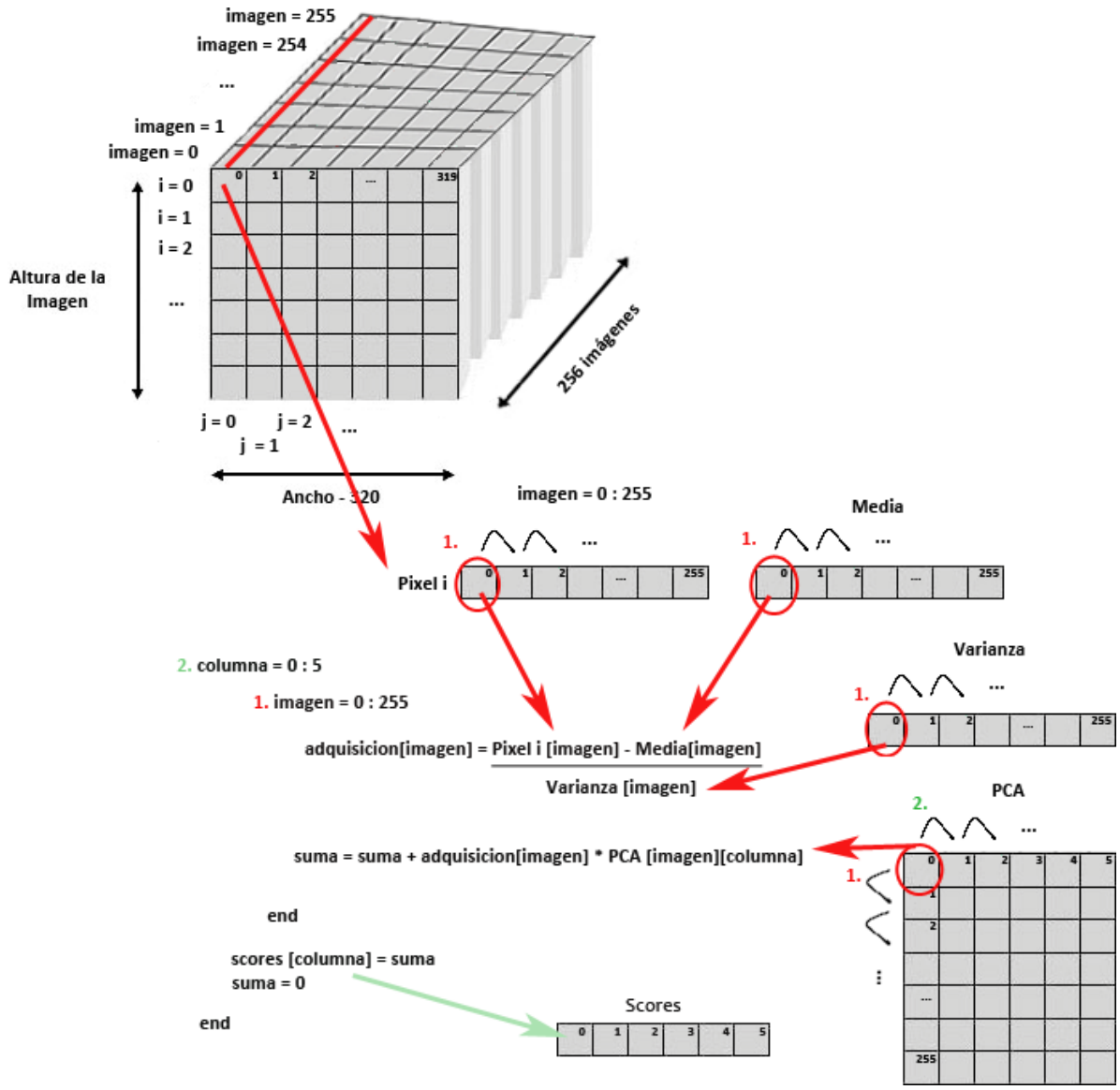


Figura 45. Centrado y Escalado seguido de un PCA de 6CP elemento a elemento (C++).

2. Tras aplicar centrado y escalado y PCA de seis componentes principales dará como resultado un vector de seis elementos y al que se le denominará *scores*. Estos *scores* obtenidos se compararán con los obtenidos mediante *MatLab* para comprobar que han sido alcanzados de forma satisfactoria. Si no es así habrá que revisar los pasos que se han llevado a cabo hasta llegar ahí.

   Una vez obtenidos los *scores* correctamente se aplicará análisis discriminante. Esto se realizará mediante un bucle doble para recorrer la matriz de coeficientes. El primer bucle será para recorrer la matriz *scores* y multiplicarla por la primera columna de la matriz coeficientes elemento a elemento. Con esta multiplicación se clasificará el vector *scores* para la primera clase. El segundo bucle será el encargado de recorrer las columnas de la matriz coeficientes para clasificar el

mismo *score* para las distintas clases o columnas, en este caso diez. Además de recorrer las columnas de la matriz coeficientes también sirve para sumar el elemento adecuado del vector constante tras acabar el primer bucle. Tras esta suma el análisis discriminante para esa clase estará finalizado.

3. Una vez sumado el elemento del vector constante se buscará en qué posición se encuentra el máximo. Esto se realizará de forma iterativa ya que en cada iteración se sumara el elemento constante y se comparará con el máximo actual y en caso de ser superior se actualiza el máximo con este valor. También se actualizará la variable que almacena la posición de ese máximo.

   A continuación se muestra una figura y un pseudocódigo que ilustra lo realizado.

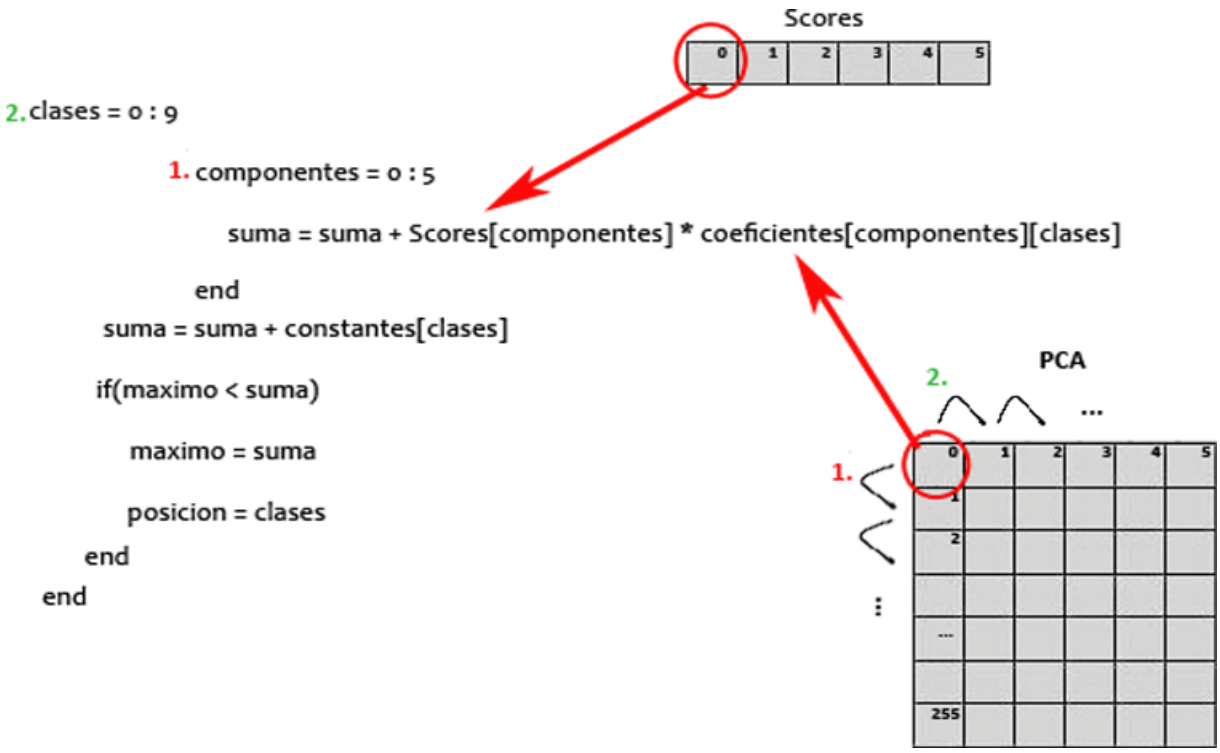


Figura 46. Análisis discriminante y cálculo del máximo del vector del modelo PCA CE 6CP (C++).

4. Una vez obtenido la posición del máximo, este valor será con el que se clasifique esta fila de trabajo, es decir, si se ha encontrado el máximo en la posición seis, ese pixel se clasificará con el valor seis. En la Figura 47 se puede ver un ejemplo de clasificación.

Estos pasos se realizará para recorrer todos los pixeles de una imagen por lo que se efectuarán en dos bucles iterativos; uno para recorrer el ancho y otro para el largo de la imagen. Por lo tanto, como resultado final dará una matriz bidimensional de ancho como la imagen, 320, y de largo como la imagen tratada, en unos casos será 887, en otros 786 y otras veces 876.

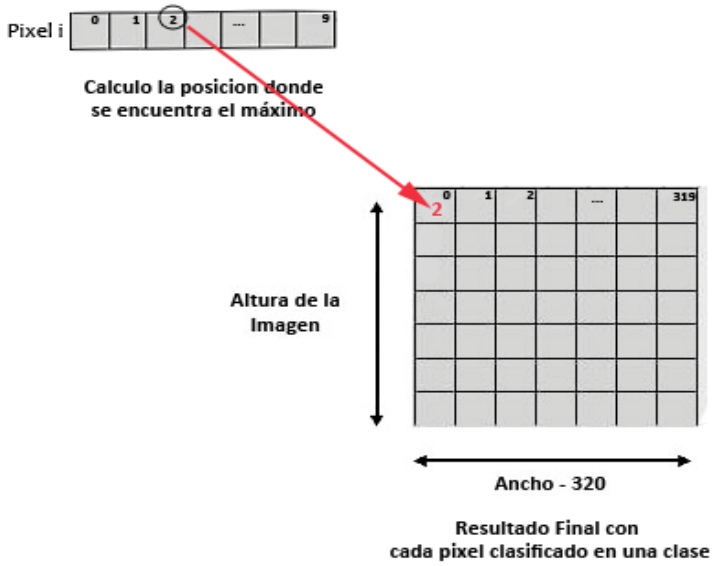


Figura 47.Clasificación tras aplicar PCA CE 6CP (C++).

Una vez aplicado el algoritmo PCA CE 6 CP, se obtendrá una matriz bidimensional donde cada valor de la matriz representará a que clase pertenece esa posición o ese pixel, se procederá a la visualización del resultado mediante la función mostraImagen del archivo imágenes. Para ello, al igual que en *MatLab* se creará una nueva imagen RGB mediante la librería OpenCV que utilizará la función cvCreateImage que tiene la siguiente estructura:

IplImage * cvCreateImage (CvSize tamaño, int profundidad bit, int canales)

El tamaño en este caso dependerá de la altura de las imágenes hiperespectrales, es decir, del tomo que se haya seleccionado mientras que la anchura es constante, 320. La profundidad de bit se refiere a la cantidad de bits de información necesarios para representar el color de un píxel en una imagen digital que en este caso serán enteros sin signo de 8 bits. En cuanto a los canales determinan que tipo de imagen será si monocromática, escala de grises o RGB que en este caso será RGB por lo tanto se escribirá un 3.

Para mostrar la imagen final se tendrá que recorrer la matriz bidimensional resultante del algoritmo PCA CE 6CP y en función de la clase a la que pertenezca ese pixel se le asignará un color u otro, es decir, si en la posición 7, que corresponde con la fila 1 y columna 7, la clase a la que pertenece ese pixel es el 5 y a la clase 5 se le ha asignado el color azul se le asignará ese color a ese pixel de la imagen final.

La forma de asignar un color a la imagen final será atreves de la librería OpenCV y mediante la función *cvSet2D* que tiene la siguiente estructura:

void cvSet2D (IplImage* imagen , int fila, int columna, CvScalar v) ;

La imagen será la imagen final creada con OpenCV; la fila indicará la fila del pixel; la columna indicará la columna del pixel y CvScalar es una estructura de OpenCV con la cual se le asignará el color determinado al pixel seleccionado mediante la fila y la columna de la imagen final. Por ejemplo, el color azul mediante la estructura CvScalar sería así:

cvScalar(255,0,0) --- (B,G,R)

Por lo tanto, se recorrerá la matriz bidimensional mediante dos bucles y con un *switch* se escribirá el valor del color correspondiente en la imagen final dependiendo del valor del pixel, es decir, a la clase que pertenezca. A continuación se puede comprobar parte del código.

```
for(int i=0;i<fila;i++){
      for(int j=0;j<columna;j++){

            switch((int)Imagen[i][j])
            {
            case 1:
                  s=cvScalar(0,255,0);
                  break;
            case 2:
                  s=cvScalar(0,255,255);
                  break;
            case 3:
                  s=cvScalar(0,90,141);
                  break;
            case 4:
                  s=cvScalar(128,128,128);
                  break;
            case 5:
                  s=cvScalar(255,0,0);
                  break;
            case 6:
                  s=cvScalar(255,255,255);
                  break;
```

```
                case 7:
                      s=cvScalar(0,0,255);
                      break;
                case 8:
                      s=cvScalar(0,103,26);
                      break;
                case 9:
                      s=cvScalar(0,90,218);
                      break;
                case 10:
                      s=cvScalar(0,0,0);
                      break;
                default:
                      s=cvScalar(0,0,0);
                      break;
                }

                cvSet2D(ImagenNueva,i,j,s);
          }
}
```

Tras asignarse los colores a la nueva imagen el siguiente paso será representar dicha imagen. Esta acción se realizará aplicando el comando cvShowImage de OpenCV cuyo parámetro de entrada será la imagen final. Una vez aplicado este comando se obtendrá la imagen química procesada. Después de aplicarlo en un tomo se volverá a ejecutar el algoritmo pero para los otros dos tomos con la única modificación de sus rutas a la hora de cargar cada una de las imágenes hiperespectrales. Por lo tanto, el resultado final de los tres tomos serán tres imágenes que se observan en la Figura 48.

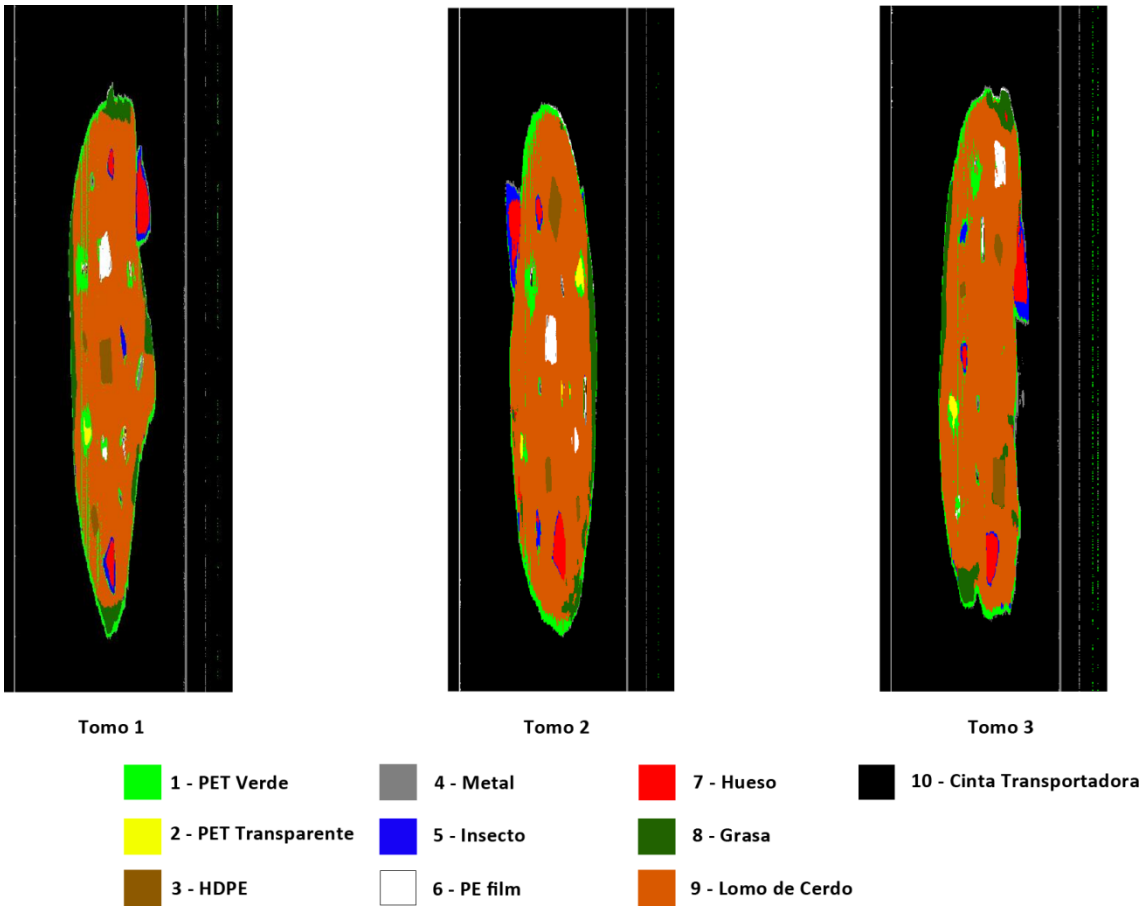


Figura 48. Imágenes Finales tras aplicar el modelo PCA CE 6CP (C++).

4.2.2 Modelo PCA 1D15CE 8 CP

Este algoritmo estará formado por tres partes. La primera, lo formará el preprocesado que consistirá en la primera derivada de Savistky-Golay y un centrado y escalado. La segunda consistirá en aplicar PCA, análisis de principales componentes, con 8 componentes principales y la tercera será aplicar análisis discriminante.

Una vez descrito las partes en que está formado este algoritmo se explicará los pasos que se llevarán a cabo en este lenguaje de programación, C++. A diferencia del apartado 4.1.1 Modelo PCA CE 6CP no se aplicará el modelo PCA 1D15CE 8 CP sobre la matriz que contiene los espectros de los distintos cuerpos extraños. Este paso no se realizará debido a que en el lenguaje anterior, apartado 4.1 Implementación en Matlab, se entendido y desarrollado correctamente obteniéndose resultados iguales al archivo "scores" y los archivos necesarios para el desarrollo del modelo PCA 1D15CE 8 CP, media y varianza. Por lo tanto, este paso en este lenguaje se ha considerado innecesarios y el modelo PCA 1D15CE 8CP se aplicará directamente sobre uno de los tomos de imágenes.

El desarrollo de este modelo en C++ estará formado por tres archivos como en el caso anterior 4.2.1 Modelo PCA CE 6 CP. Uno el programa principal, el segundo archivo denominado imágenes donde estará todas las funciones relacionados con las imágenes, como son lectura de imagen, columnas que posee una imagen, filas que posee una imagen y visualización de una imagen. Y el tercer archivo denominado proceso donde estará todas las funciones relacionadas con el algoritmo, como son lecturas de los archivos necesarios, visualización por pantalla de los contenidos de los archivos, reserva de memorias, liberación de memoria y el modelo PCA 1D15CE 8CP. A continuación, se describirá con más detalles todos los pasos llevados para el desarrollo del modelo PCA 1D15CE 8CP.

En primer lugar, se definirán las rutas y las variables para leer los distintos archivos necesarios para la aplicación del modelo PCA 1DCE 8CP. Para leer los distintos archivos dependerá si el archivo leído es un vector o una matriz. Los archivos media, varianza y constantes son vectores por lo que se utilizará la función leervector mientras que los archivos PCA y coeficientes son matrices que se leerán con la función leermatriz del archivo procesos. Como se sabe de qué tamaño son los vectores y las

matrices lo que se hace en cada una de las funciones, dependiendo de si es vector o matriz, es ir leyendo por espacios hasta completar el tamaño deseado del vector o matriz. Esto se realizará utilizando la biblioteca iostream de C++. Ver anexo.

Una vez se tenga cargado los archivos que se necesitarán para el algoritmo PCA 1D15CE 8CP el siguiente paso será cargar las imágenes de unos de los tomos que hay. Este paso se realizará como en el modelo anterior, 4.2.1 Modelo PCA CE 6 CP.

Una vez cargadas todas las imágenes de uno de los tomos en la matriz tridimensional se procederá a aplicar el algoritmo PCA 1D15CE 8CP mediante la función CracionImagen del archivo proceso.

1. En primer lugar se compondrá una fila de trabajo compuesto por 256 elementos. Esta fila se formará accediendo al mismo pixel para las distintas imágenes. Este proceso se realizará en un bucle accediendo adecuadamente a la matriz tridimensional. A la vez que se accede a un elemento de la imagen se le aplicará la primera derivada de Saviztky-Golay de segundo orden con una ventana de 15 puntos. Esto se realizará mediante un bucle doble para multiplicar el pixel accedido con el correspondiente coeficiente de la matriz derivada. A continuación se muestra una figura y un pseudocódigo que ilustra lo realizado.

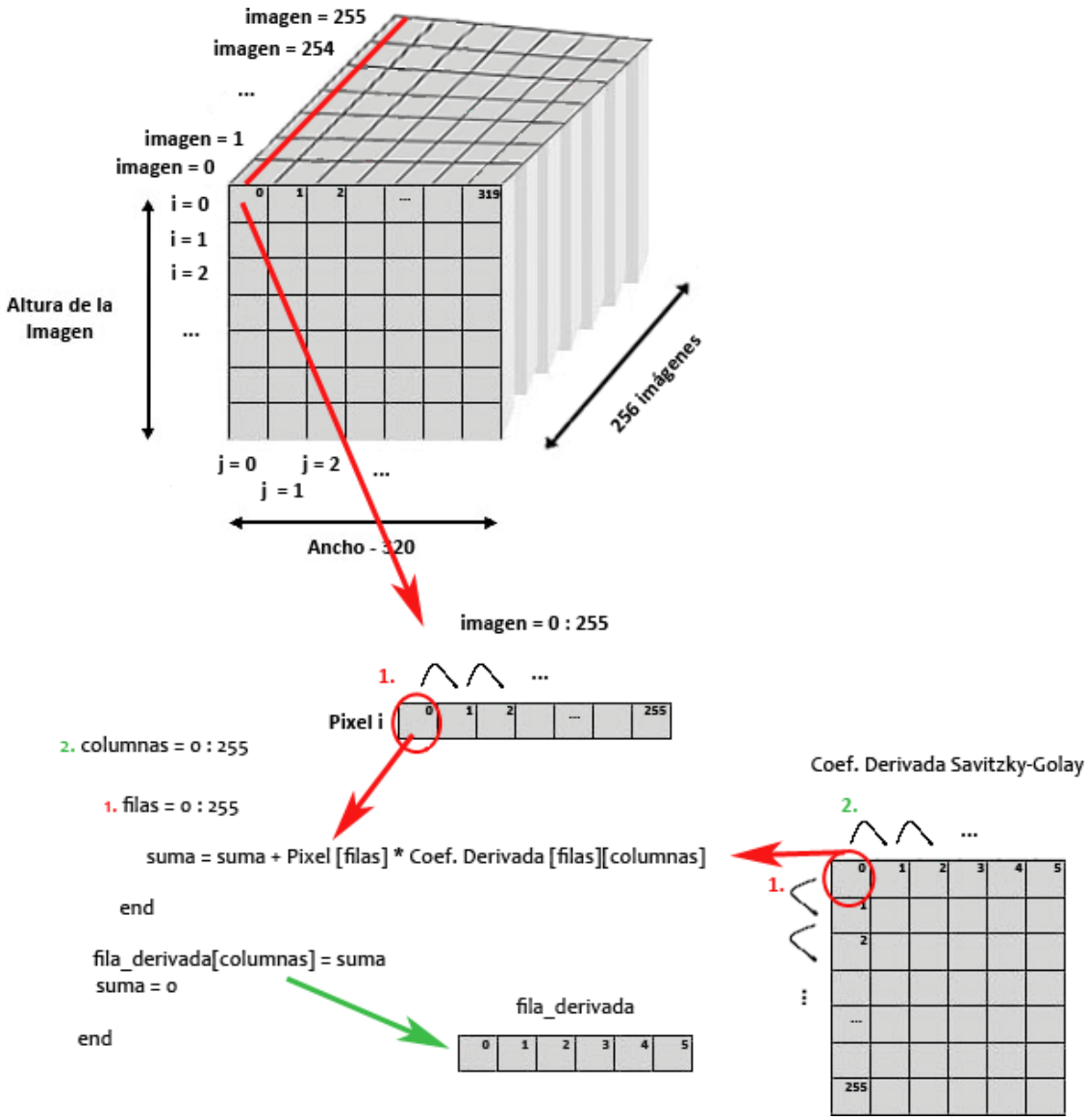


Figura 49. Adquisición de la fila de trabajo seguido de la primera derivada de Saviztky-Golay (C++).

2. Tras tener la fila derivada el siguiente paso será aplicar el centrado y escalado según la fórmula 2.2. Además de aplicarle el centrado y escalado se multiplicará por el correspondiente coeficiente de la matriz PCA, es decir, se multiplicará el elemento de la fila derivada accedido en ese instante por el coeficiente que corresponde con la fila igual a la posición del elemento accedido de la fila de trabajo y la columna será igual a la iteración del otro bucle anidado para poder recorrer las ocho componentes principales o mejor dicho las columnas de la matriz PCA. Mientras se esté en el bucle de la fila derivada de trabajo, es decir, el primer bucle se irán sumando todas las multiplicaciones que se vayan haciendo con PCA. Al final de este bucle gracias a estas sumas se tendrá el *score* que corresponderá con la iteración con la que se esté en el segundo bucle.

A continuación se muestra una figura y un pseudocódigo que ilustra lo realizado.

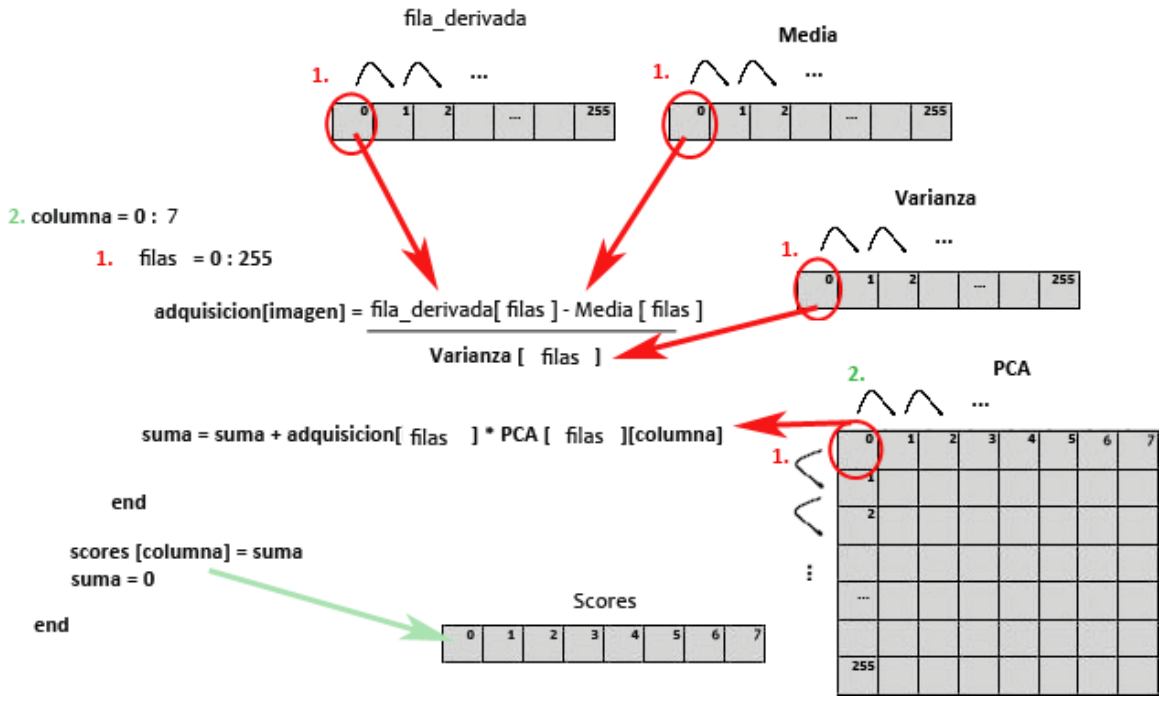


Figura 50. Centrado y Escalado seguido de un PCA de 8 CP elemento a elemento (C++).

3. Tras aplicar centrado y escalado y PCA de ocho componentes principales dará como resultado un vector de ocho elementos y al que se le denominará *scores*. Estos *scores* obtenidos se compararán con los obtenidos mediante *MatLab* para comprobar que han sido alcanzados de forma satisfactoria. Si no es así habrá que revisar los pasos que se han llevado a cabo hasta llegar ahí.

   Una vez obtenidos los *scores* correctamente se aplicará análisis discriminante. Esto se realizará mediante un bucle doble para recorrer la matriz de coeficientes. El primer bucle será para recorrer la matriz *scores* y multiplicarla por la primera columna de la matriz coeficientes elemento a elemento. Con esta multiplicación se clasificará el vector *scores* para la primera clase. El segundo bucle será el encargado de recorrer las columnas de la matriz coeficientes para clasificar el mismo *score* para las distintas clases o columnas, en este caso diez. Además de recorrer las columnas de la matriz coeficientes también sirve para sumar el elemento adecuado del vector constante tras acabar el primer bucle. Tras esta suma el análisis discriminante para esa clase estará finalizado.

4. Una vez sumado el elemento del vector constante se buscará en qué posición se encuentra el máximo. Esto se realizará de forma iterativa ya que en cada iteración se sumara el elemento constante y se comparará con el máximo actual y en caso de ser superior se actualiza el máximo con este valor. También se actualizará la variable que almacena la posición de ese máximo.

   A continuación se muestra una figura y un pseudocódigo que ilustra lo realizado.

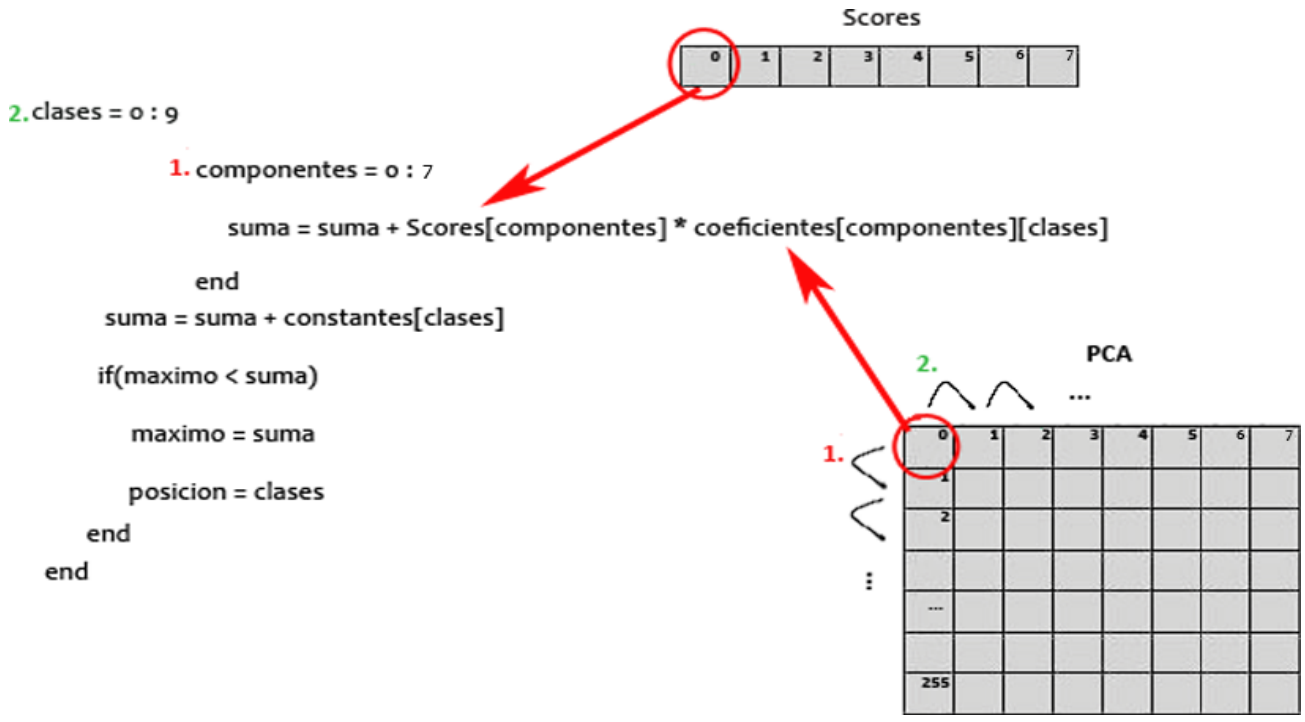


Figura 51. Análisis discriminante y cálculo del máximo del vector del modelo PCA 1D15CE 8CP (C++).

5. Una vez obtenido la posición del máximo, este valor será con el que se clasifique esta fila de trabajo, es decir, si se ha encontrado el máximo en la posición seis, ese pixel se clasificará con el valor seis. En la Figura 52 se puede ver un ejemplo de clasificación.

Estos pasos se realizará para recorrer todos los pixeles de una imagen por lo que se efectuarán en dos bucles iterativos; uno para recorrer el ancho y otro para el largo de la imagen. Por lo tanto, como resultado final dará una matriz bidimensional de ancho como la imagen, 320, y de largo como la imagen tratada, en unos casos será 887, en otros 786 y otras veces 876.

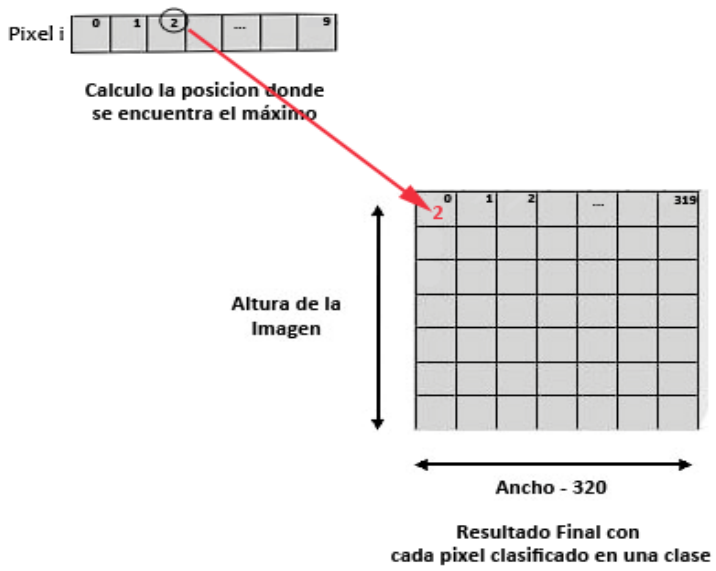


Figura 52. Clasificación tras aplicar PCA 1D15CE 8CP (C++).

Una vez aplicado el algoritmo PCA 1D15CE 8 CP, se obtendrá una matriz bidimensional donde cada valor de la matriz representará a que clase pertenece esa posición o ese pixel, se procederá a la visualización del resultado mediante la función mostraImagen del archivo imágenes. Para ello, al igual que en *MatLab* se creará una nueva imagen RGB mediante la librería OpenCV que utilizará la función cvCreateImage que tiene la siguiente estructura:

IplImage * cvCreateImage (CvSize tamaño, int profundidad bit, int canales)

El tamaño en este caso dependerá de la altura de las imágenes hiperespectrales, es decir, del tomo que se haya seleccionado mientras que la anchura es constante, 320. La profundidad de bit se refiere a la cantidad de bits de información necesarios para representar el color de un píxel en una imagen digital que en este caso serán enteros sin signo de 8 bits. En cuanto a los canales determinan que tipo de imagen será si

monocromática, escala de grises o RGB que en este caso será RGB por lo tanto se escribirá un 3.

Para mostrar la imagen final se tendrá que recorrer la matriz bidimensional resultante del algoritmo PCA 1D15CE 8CP y en función de la clase a la que pertenezca ese pixel se le asignará un color u otro, es decir, si en la posición 7, que corresponde con la fila 1 y columna 7, la clase a la que pertenece ese pixel es el 5 y a la clase 5 se le ha asignado el color azul se le asignará ese color a ese pixel de la imagen final.

La forma de asignar un color a la imagen final será atreves de la librería OpenCV y mediante la función *cvSet2D* que tiene la siguiente estructura:

void cvSet2D (IplImage* imagen , int fila, int columna, CvScalar v) ;

La imagen será la imagen final creada con OpenCV; la fila indicará la fila del pixel; la columna indicará la columna del pixel y CvScalar es una estructura de OpenCV con la cual se le asignará el color determinado al pixel seleccionado mediante la fila y la columna de la imagen final. Por ejemplo, el color azul mediante la estructura CvScalar sería así:

cvScalar(255,0,0) --- (B,G,R)

Por lo tanto, se recorrerá la matriz bidimensional mediante dos bucles y con un *switch* se escribirá el valor del color correspondiente en la imagen final dependiendo del valor del pixel, es decir, a la clase que pertenezca. A continuación se puede comprobar parte del código.

```
for(int i=0;i<fila;i++){
      for(int j=0;j<columna;j++){

            switch((int)Imagen[i][j])
            {
            case 1:
                  s=cvScalar(0,255,0);
                  break;
            case 2:
                  s=cvScalar(0,255,255);
                  break;
            case 3:
                  s=cvScalar(0,90,141);
                  break;
            case 4:
                  s=cvScalar(128,128,128);
```

```
                        break;
                case 5:
                        s=cvScalar(255,0,0);
                        break;
                case 6:
                        s=cvScalar(255,255,255);
                        break;
                case 7:
                        s=cvScalar(0,0,255);
                        break;
                case 8:
                        s=cvScalar(0,103,26);
                        break;
                case 9:
                        s=cvScalar(0,90,218);
                        break;
                case 10:
                        s=cvScalar(0,0,0);
                        break;
                default:
                        s=cvScalar(0,0,0);
                        break;
                }

                cvSet2D(ImagenNueva,i,j,s);
        }
}
```

Tras asignarse los colores a la nueva imagen el siguiente paso será representar dicha imagen. Esta acción se realizará aplicando el comando cvShowImage de OpenCV cuyo parámetro de entrada será la imagen final. Una vez aplicado este comando se obtendrá la imagen química procesada. Después de aplicarlo en un tomo se volverá a ejecutar el algoritmo pero para los otros dos tomos con la única modificación de sus rutas a la hora de cargar cada una de las imágenes hiperespectrales. Por lo tanto, el resultado final de los tres tomos serán tres imágenes que se observan en la Figura 53.

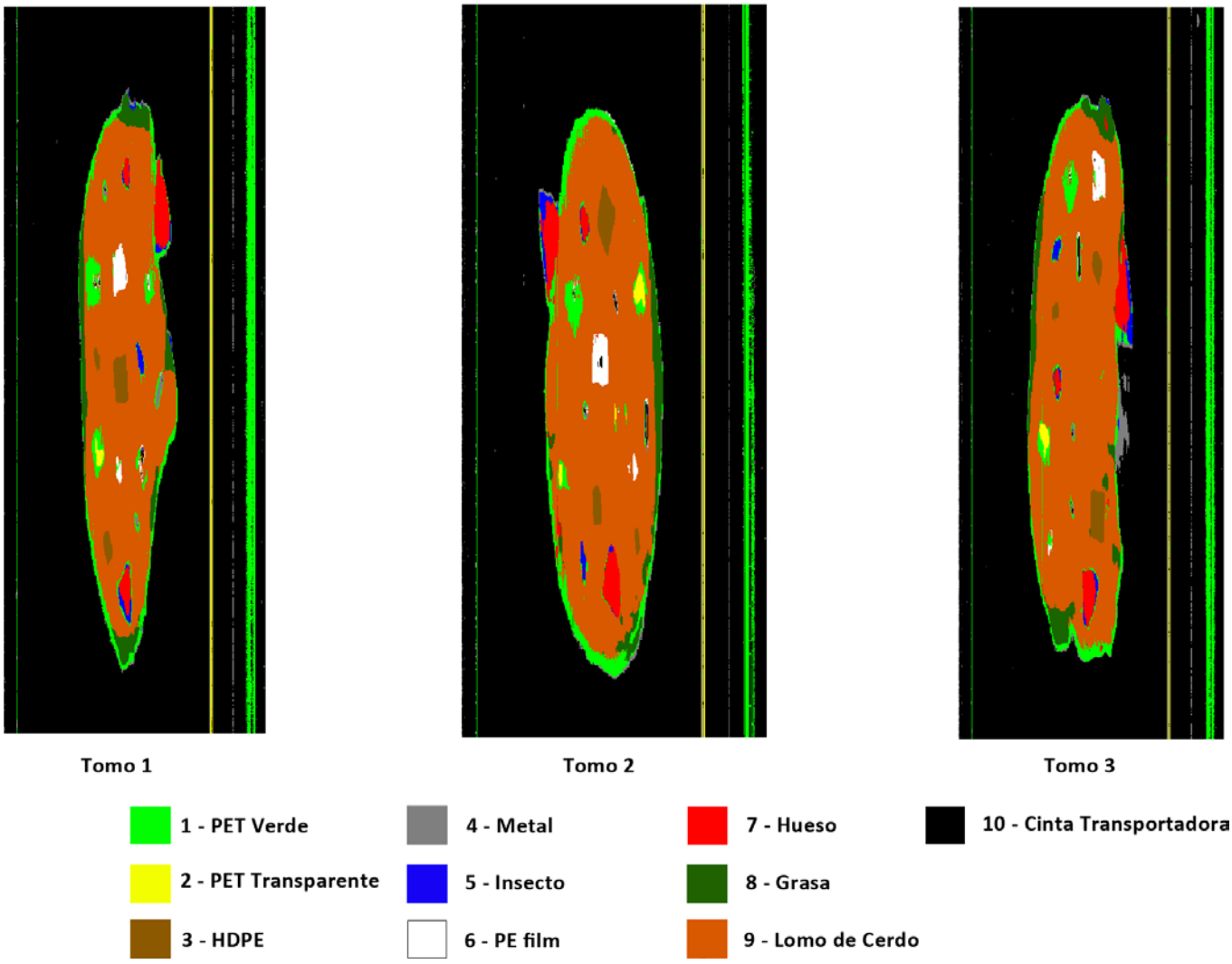


Figura 53.Imágenes Finales tras aplicar el modelo PCA 1D15CE 8CP (C++).

4.3 Implementación en CUDA

Por último se implementara los algoritmos en la alternativa elegida, GPU, mediante CUDA. Para desarrollar los algoritmos se deberá cambiar la estructura de los programas que se han implementado. Esto se debe principalmente a que los lenguajes utilizados hasta ahora eran secuenciales sin embargo la alternativa elegida se trata de un lenguaje paralelo.

Por lo tanto, para esta implementación el programador debe entender completamente los algoritmos y haber comprendido la arquitectura CUDA. Antes de todo, el programador deberá de haber pensado antes los hilos de ejecución que se lanzarán en paralelo en cada bloque y cuantos bloques se ejecutarán en paralelo para maximizar la potencia de cálculo de la GPU.

Otro aspecto importante durante la implementación de los algoritmos es que la GPU se encargará de los cálculos intensivos, debido a su gran cantidad de núcleos, aliviando a la CPU de tener que procesar gran cantidad de datos. Razón por lo cual se espera que haya una considerable reducción de los tiempos de procesado, el cuál es el objetivo principal de este proyecto final de carrera.

Por lo tanto, en el transcurso del presente apartado se describirá la estrategia que se seguirá y el desarrollo seguido para la implementación de los algoritmos en un lenguaje paralelo. En primer lugar el modelo PCA CE 6CP y luego el modelo PCA 1D15CE 8CP.

4.3.1 Modelo PCA CE 6 CP

Este algoritmo estará formado por tres partes. La primera, lo formará el preprocesado que consistirá un centrado y escalado. La segunda consistirá en aplicar PCA, análisis de principales componentes, con 6 componentes principales y la tercera será aplicar análisis discriminante.

Una vez descrito las partes en que está formado este algoritmo se explicará los pasos que se llevarán a cabo en este lenguaje de programación paralela, CUDA. A diferencia del apartado 4.1.1 Modelo PCA CE 6CP no se aplicará el modelo PCA CE 6 CP sobre la matriz que contiene los espectros de los distintos cuerpos extraños. Este paso no se realizará debido a que en el lenguaje anterior, apartado 4.1 Implementación en Matlab, se entendido y desarrollado correctamente obteniéndose resultados iguales al archivo "scores" y los archivos necesarios para el desarrollo del modelo PCA CE 6 CP, media y varianza. Por lo tanto, este paso en este lenguaje se ha considerado innecesarios y el modelo PCA CE 6CP se aplicará directamente sobre uno de los tomos de imágenes.

El desarrollo de este modelo en CUDA[29] estará formado por cinco archivos. Uno el programa principal, que será lomos.cu, el segundo archivo denominado imágenes donde estará todas las funciones relacionados con las imágenes, como son lectura de imagen, columnas que posee una imagen, filas que posee una imagen y visualización de una imagen. El tercer archivo denominado proceso donde estará todas las funciones relacionadas con el algoritmo, como son lecturas de los archivos necesarios, visualización por pantalla de los contenidos de los archivos, reserva de memorias, liberación de memoria y el modelo PCA CE 6CP secuencial. Estos dos archivos últimos son herencia del lenguaje de programación secuencial y que permitirán la lectura de las imágenes, archivos y la visualización de las imágenes tras todo el proceso. El cuarto y el quinto archivo estarán todas las funciones relacionada con la GPU, como son los kernels, reservas de memoria en la GPU...

A continuación, se describirá con más detalles todos los pasos llevados para el desarrollo del modelo PCA CE 6CP.

En primer lugar, se definirán las rutas y las variables para leer los distintos archivos necesarios para la aplicación del modelo PCA CE 6CP. Para leer los distintos

archivos se realizará de la misma manera que en el apartado 4.2.1 Modelo PCA CE 6 CP.

Una vez que se tengan cargados los archivos que se necesitarán para el algoritmo PCA CE 6CP el siguiente paso será cargar las imágenes de unos de los tomos que hay. Para este cometido se utilizará la librería OpenCV (1.3.1 Herramientas Utilizadas). Al igual que los archivos cargados, las imágenes se leerán de la misma manera que en el apartado 4.2.1 Modelo PCA CE 6 CP.

Tras cargar todas las imágenes de uno de los tomos en la matriz tridimensional se procederá a paralelizar el algoritmo PCA CE 6CP. Para ello, primero habrá que enviar todos los datos cargados en la CPU, media, varianza, PCA, coeficientes y constantes del análisis discriminante e imágenes, a la GPU. Antes de transferir la información a la GPU se reservará memoria en ella mediante la instrucción *cublasAlloc* de la librería CUBLAS que presenta la siguiente estructura:

cublasStatus cublasAlloc (int tamaño, int elemSize, void **VariableGPU)

Con tamaño se indicará el número de elementos que se van a transferir. ElemSize dirá el tamaño en bytes de un tipo de dato, por ejemplo sizeof(float) y variableGPU hace referencia a la variable declarada en la GPU a la cual se le reservará ese espacio de memoria.

Una vez reservado memoria el siguiente paso será transferir dicha información a la GPU para esto se utilizará el comando *cudaMemcpy* de la librería CUDA y que presenta la siguiente estructura:

cudaError_t cudaMemcpy(void* destino, const void* origen, size_t tamaño, enum cudaMemcpykind direccion)

Destino hace referencia a las dirección de memoria destino, es decir, donde se copiarán los datos que en este caso se refiere a la variable definida en la GPU, origen hace referencia a la dirección de memoria origen, es decir, donde se encuentra lo que se va a enviar en la GPU en este caso será una variable definida en la CPU. Tamaño indica el número de bytes que se van a transferir y dirección indica en qué sentido se realiza la transferencia si CPU - GPU o GPU - CPU que en este caso será CPU - GPU.

Tras reservar memoria en la GPU y la transferencia de todos los archivos necesarios incluido las imágenes en la GPU. Los siguiente paso a realizar serán el pretratamiento, el procesado y la clasificación de las imágenes hiperespectrales en la GPU.

Lo primero será realizar el pretratamiento en la GPU. Antes de aplicar el kernel correspondiente, CentradoyEscalado, se definirán los hilos y los bloques que se van a ejecutar en paralelo. Para ver cuál es el número optimo de hilos y bloques se realizarán distintas pruebas hasta llegar al óptimo, que en este caso serán 256 hilos y 320 bloques en el eje x y en el eje y será la longitud de la imagen hiperespectral que variará según el tomo de imagen cargado. Para el primer tomo será 887, para el segundo 786 y para el tercero 876. A continuación se invocará al kernel que se encargará de dividir en varios bloques, tanto en x como en y, la matriz tridimensional donde en cada bloque se ejecutarán 256 hilos. Cada hilo accederá a una parte de la matriz tridimensional en función de que bloque x y bloque y sea. Luego para cada acceso de un hilo de cada bloque se realizará el centrado y escalado, es decir, a cada acceso se le restará la media y se dividirá por la desviación típica y el resultado se guardará en la misma posición accedida. En la Figura 54 se puede ver un diagrama de lo explicado.

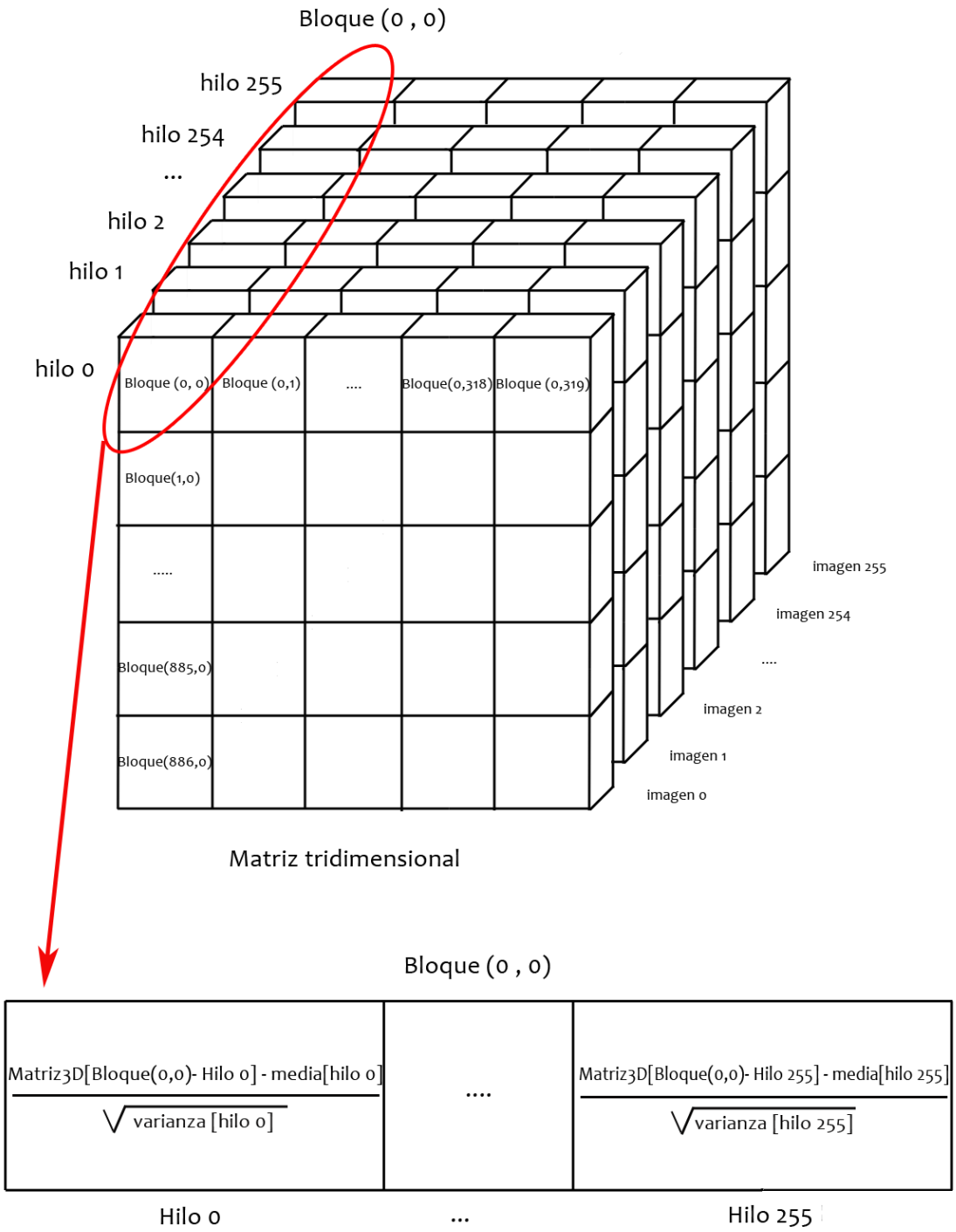


Figura 54. Kernel Centrado y Escalado aplicado a la matriz tridimensional.

Una vez se tenga la matriz tridimensional centrada y escalada, el siguiente paso será aplicarle análisis de principales componentes. Este paso se realizará multiplicando la matriz tridimensional por la matriz PCA. Esta multiplicación se realizará en CUDA mediante la biblioteca CUBLAS de una forma sencilla y eficiente.

La biblioteca CUBLAS [30] es una implementación de BLAS (Subprogramas básicos de álgebra lineal) en la parte superior del tiempo de ejecución de la GPU. Permite acceder a los recursos computacionales de la unidad de procesamiento gráfico, GPU, pero no permite auto-paralelizar a través de múltiples GPUs. En este caso solo habrá una GPU por lo tanto no hay ningún problema.

Para utilizar la biblioteca CUBLAS, se le deberán pasar las matrices y los vectores adecuados en el espacio de memoria de la GPU. A continuación se llamará a la función de CUBLAS para que realice la multiplicación en el espacio de memoria de la GPU. La función utilizada es *cublasSgemm* y que presenta la siguiente estructura:

$$\mathbf{C} = \boldsymbol{\alpha} * \mathbf{op}(\mathbf{A}) * \mathbf{op}(\mathbf{B}) + \boldsymbol{\beta} * \mathbf{C}$$

Donde alfa y beta son dos valores constantes y A, B y C son matrices almacenadas en formato de columna principal. Este formato consiste en almacenar contiguamente en memoria las matrices en el orden de las columnas.

A continuación se muestra un ejemplo de cómo almacenan los datos teniendo la siguiente matriz:

5 8 2
4 7 9
3 6 0

Los datos se almacenarán contiguamente en memoria de la siguiente manera:

5 4 3 8 7 6 2 9 0

La estructura de la función es la siguiente:

cublasSgemm(cublasHandle_t handle,cublasOperation_t transa, cublasOperation_t transb,int m, int n, int k,const float *alpha,const float *A, int lda, const float *B, int ldb,const float *beta,float *C, int ldc)

Por lo tanto, con esta función se multiplicará la matriz tridimensional centrada y escalad, almacenada de forma que cada fila de trabajo forma una columna como se puede comprobar en la Figura 55, con la matriz PCA de una sola operación. En las implementaciones de *MatLab* y C++ se tenían que hacer una multiplicación por cada

fila de trabajo pretratada ahora todas esas multiplicaciones se reducen a una con lo que ello significa, como se verá en el apartado de tiempos.

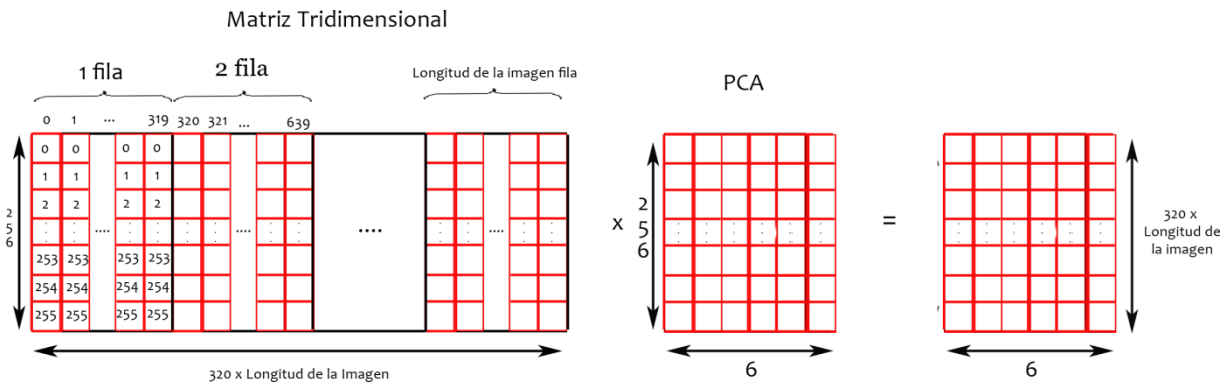


Figura 55. Multiplicación de la matriz 3D pretratada por la matriz PCA mediante CUBLAS en el algoritmo PCA CE 6CP (CUDA).

Tras aplicar PCA de seis componentes principales dará como resultado una matriz de seis columnas y de 320 x longitud de las imágenes al que se le denominará *scores*. Estos *scores* obtenidos se compararán con los obtenidos mediante *MatLab* para comprobar que han sido alcanzados de forma satisfactoria. Si no es así habrá que revisar los pasos que se han llevado a cabo hasta llegar ahí.

Una vez obtenidos los *scores* correctamente se aplicará análisis discriminante. Esto se realizará multiplicando la matriz obtenida, scores, por la matriz de coeficientes con la ayuda también de la función *cublasSgemm* como se puede observar en la Figura 56.De este modo con una sola multiplicación se multiplicaran ambas matrices, suprimiendo repetidas multiplicaciones lo que significa reducir considerablemente el tiempo al igual que en la multiplicación anterior.

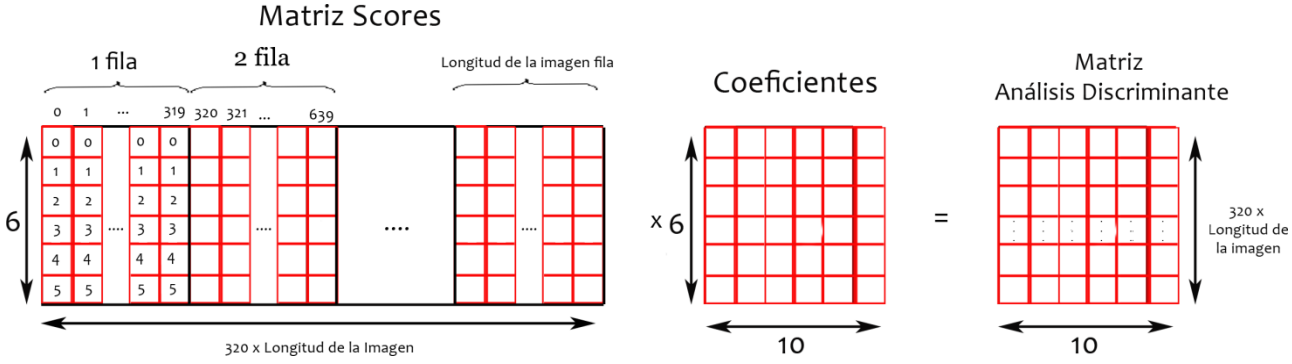


Figura 56. Multiplicación de la matriz *Scores* por la matriz Coeficientes mediante CUBLAS en el algoritmo PCA CE 6CP (CUDA).

Ahora solo falta sumarle a la matriz resultante el vector constante para acabar de aplicar el análisis discriminante a toda la matriz. Mediante el segundo *kernel* implementado llamado Fin se le sumará el vector constante y a la vez se calculará el máximo para cada fila de trabajo con lo cual se clasificará cada fila en una categoría.

·En este último *kernel* se lanzarán 256 hilos y los bloques será el resultado de 320 por la longitud de la imagen divido entre 256. El número de bloques se realizará de esta manera ya que la parte de encontrar el máximo se realizará de manera secuencial recorriendo la fila accedida. Por lo tanto, cada hilo accederá a una fila y el número de bloques corresponderá con cuantas filas haya divido entre el número de hilos lanzados. Una vez encontrado la posición del máximo para una fila accedida este valor será con el cual se clasifique esta fila. La Figura 57 muestra un ejemplo de lo realizado en este último *kernel*.

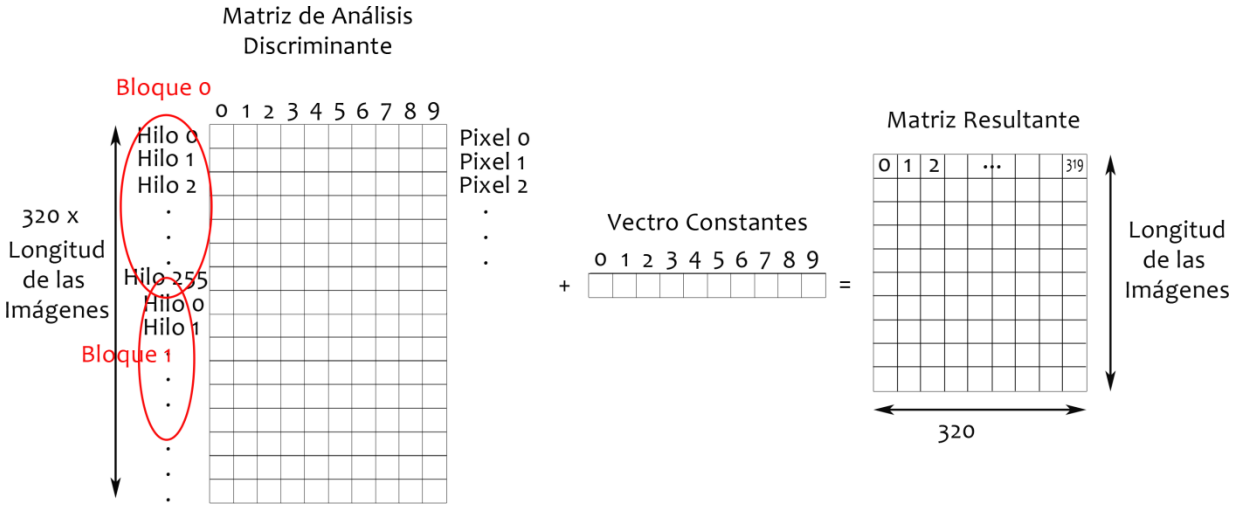


Figura 57. Suma del vector constantes más cálculo de la posición del máximo para la matriz resultante en el algoritmo PCA CE 6 CE (CUDA).

Una vez aplicado el algoritmo PCA CE 6 CP, se obtendrá una matriz bidimensional donde cada valor de la matriz resultante representará a que clase pertenece esa posición o ese pixel, se procederá a la visualización del resultado. Para ello, antes de todo habrá que enviar la matriz resultante que se encuentra alojada en memoria de la GPU a la CPU mediante el mecanismo inverso al utilizado al principio. Una vez se tenga la matriz resultante en la CPU el siguiente paso es visualizar la imagen

mediante la librería de OpenCV al igual que se hizo en el apartado 4.2 Implementación en C++

Después de aplicarlo en un tomo se volverá a ejecutar el algoritmo pero para los otros dos tomos con la única modificación de sus rutas a la hora de cargar cada una de las imágenes hiperespectrales. Por lo tanto, el resultado final de los tres tomos serán tres imágenes que se observan en la Figura 58.

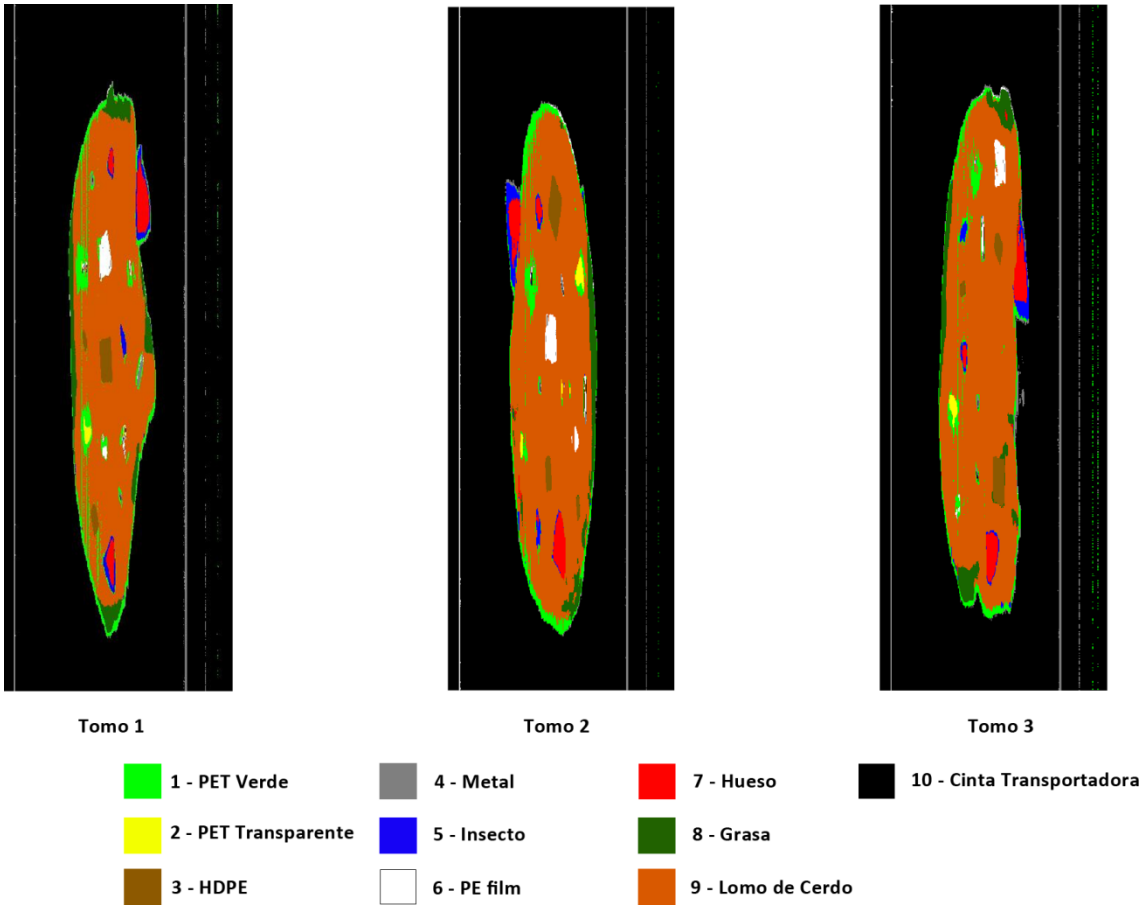


Figura 58. Imágenes Finales tras aplicar el modelo PCA CE 6CP (CUDA).

4.3.2 Modelo PCA 1D15CE 8 CP

Este algoritmo estará formado por tres partes. La primera, lo formará el preprocesado que consistirá en la primera derivada de Savistky-Golay y un centrado y escalado. La segunda consistirá en aplicar PCA, análisis de principales componentes, con 8 componentes principales y la tercera será aplicar análisis discriminante.

Una vez descrito las partes en que está formado este algoritmo se explicará los pasos que se llevarán a cabo en este lenguaje de programación paralela, CUDA. A diferencia del apartado 4.1.1 Modelo PCA CE 6CP no se aplicará el modelo PCA 1D15CE 8CP sobre la matriz que contiene los espectros de los distintos cuerpos extraños. Este paso no se realizará debido a que en el lenguaje anterior, apartado 4.1 Implementación en Matlab, se entendido y desarrollado correctamente obteniéndose resultados iguales al archivo "scores" y los archivos necesarios para el desarrollo del modelo PCA 1D15CE 8CP, media y varianza. Por lo tanto, este paso en este lenguaje se ha considerado innecesarios y el modelo PCA 1D15CE 8CP se aplicará directamente sobre uno de los tomos de imágenes.

El desarrollo de este modelo en CUDA estará formado por cinco archivos. Uno el programa principal, que será lomos.cu, el segundo archivo denominado imágenes donde estará todas las funciones relacionados con las imágenes, como son lectura de imagen, columnas que posee una imagen, filas que posee una imagen y visualización de una imagen. El tercer archivo denominado proceso donde estará todas las funciones relacionadas con el algoritmo, como son lecturas de los archivos necesarios, visualización por pantalla de los contenidos de los archivos, reserva de memorias, liberación de memoria y el modelo PCA 1D15CE 8CP secuencial. Estos dos archivos últimos son herencia del lenguaje de programación secuencial y que permitirán la lectura de las imágenes, archivos y la visualización de las imágenes tras todo el proceso. El cuarto y el quinto archivo estarán todas las funciones relacionada con la GPU, como son los kernels, reservas de memoria en la GPU...

A continuación, se describirá con más detalles todos los pasos llevados para el desarrollo del modelo PCA 1D15CE 8CP.

En primer lugar, se definirán las rutas y las variables para leer los distintos archivos necesarios para la aplicación del modelo PCA 1D15CE 8CP. Para leer los distintos archivos se realizará de la misma manera que en el apartado 4.2.1 Modelo PCA CE 6 CP.

Una vez que se tengan cargados los archivos que se necesitarán para el algoritmo PCA 1D15CE 8CP el siguiente paso será cargar las imágenes de unos de los tomos que hay. Para este cometido se utilizará la librería OpenCV (1.3.1 Herramientas Utilizadas). Al igual que los archivos cargados, las imágenes se leerán de la misma manera que en el apartado 4.2.1 Modelo PCA CE 6 CP.

Tras cargar todas las imágenes de uno de los tomos en la matriz tridimensional se procederá a paralelizar el algoritmo PCA 1D15CE 8CP. Para ello, primero habrá que enviar todos los datos cargados en la CPU, derivada, media, varianza, PCA, coeficientes y constantes del análisis discriminante e imágenes, a la GPU. Antes de transferir la información a la GPU se reservará memoria en ella mediante la instrucción *cublasAlloc* de la librería CUBLAS tal como se hizo en el apartado 4.3.1 Modelo PCA CE 6 CP

Una vez reservado memoria el siguiente paso será transferir dicha información a la GPU para esto se utilizará el comando *cudaMemcpy* de la librería CUDA de la misma manera que en el apartado 4.3.1 Modelo PCA CE 6 CP

Tras reservar memoria en la GPU y la transferencia de todos los archivos necesarios incluido las imágenes en la GPU, los siguientes pasos a realizar serán el pretratamiento, el procesado y la clasificación de las imágenes hiperespectrales en la GPU.

Lo primero será realizar el pretratamiento en la GPU. Para empezar se aplicará la primera derivada de Saviztky-Golay a la matriz tridimensional. Este paso consistirá en multiplicar con la función *cublasSgemm* de la librería CUBLAS, utilizada en el apartado 4.3.1, la matriz tridimensional con la matriz que contiene los coeficientes de la primera derivada de Saviztky-Golay. Una vez multiplicado se tendrá una matriz tridimensional derivada y suavizada. El siguiente paso del preprocesado será aplicar el kernel denominado centrado y escalado. Antes de aplicar el kernel, CentradoyEscalado, se definirán los hilos y los bloques que se van a ejecutar en paralelo. Para ver cuál es el número optimo de hilos y bloques se realizarán distintas pruebas hasta llegar al óptimo,

que en este caso serán 256 hilos y 320 bloques en el eje x y en el eje y será la longitud de la imagen hiperespectral que variará según el tomo de imagen cargado. Para el primer tomo será 887, para el segundo 786 y para el tercero 876. A continuación se invocará al kernel que se encargará de dividir en varios bloques, tanto en x como en y, la matriz tridimensional donde en cada bloque se ejecutarán 256 hilos. Cada hilo accederá a una parte de la matriz tridimensional en función de que bloque x y bloque y sea. Luego para cada acceso de un hilo de cada bloque se realizará el centrado y escalado, es decir, a cada acceso se le restará la media y se dividirá por la desviación típica y el resultado se guardará en la misma posición accedida. En Figura 59 se puede ver un diagrama de lo explicado.

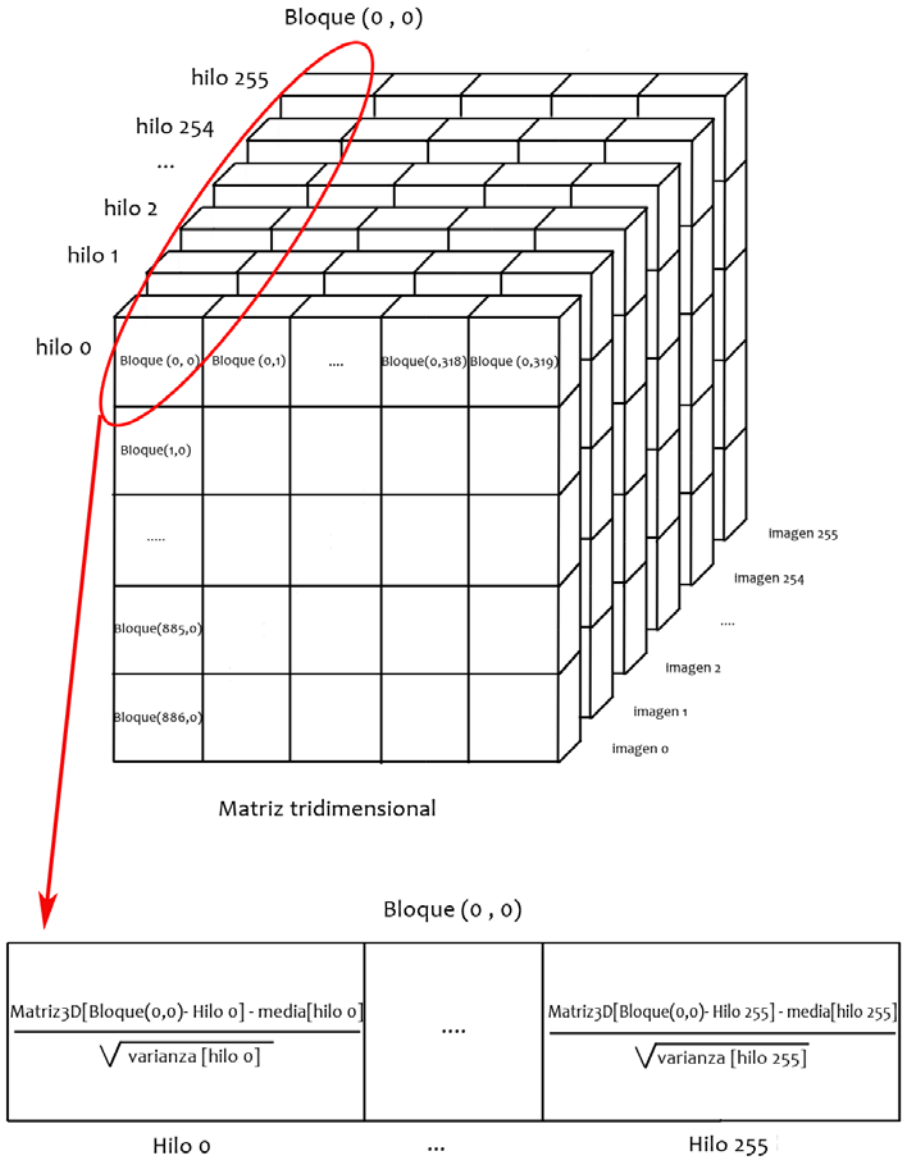


Figura 59. Kernel Centrado y Escalado aplicado a la matriz tridimensional.

Una vez se tenga la matriz tridimensional centrada y escalada, el siguiente paso será aplicarle análisis de principales componentes. Este paso se realizará multiplicando la matriz tridimensional por la matriz PCA. Esta multiplicación se realizará en CUDA mediante la función cublasSgemm de la biblioteca CUBLAS, como en el apartado 4.3.1, de una forma sencilla y eficiente.

Por lo tanto, con esta función se multiplicará la matriz tridimensional centrada y escalad, almacenada de forma que cada fila de trabajo forma una columna como se puede comprobar en la Figura 60, con la matriz PCA de una sola operación. En las implementaciones de *MatLab* y C++ se tenían que hacer una multiplicación por cada fila de trabajo pretratada ahora todas esas multiplicaciones se reducen a una con lo que ello significa, como se verá en el apartado de tiempos.

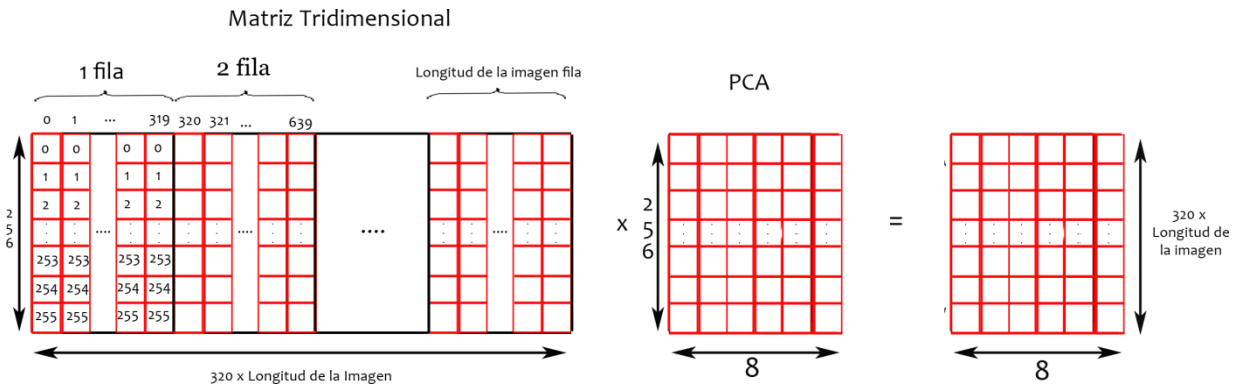


Figura 60. Multiplicación de la matriz 3D pretratada por la matriz PCA mediante CUBLAS en el algoritmo PCA 1D15CE 8CP (CUDA).

Tras aplicar PCA de ocho componentes principales dará como resultado una matriz de ocho columnas y de 320 x longitud de las imágenes al que se le denominará *scores*. Estos *scores* obtenidos se compararán con los obtenidos mediante *MatLab* para comprobar que han sido alcanzados de forma satisfactoria. Si no es así habrá que revisar los pasos que se han llevado a cabo hasta llegar ahí.

Una vez obtenidos los *scores* correctamente se aplicará análisis discriminante. Esto se realizará multiplicando la matriz obtenida, scores, por la matriz de coeficientes con la ayuda también de la función *cublasSgemm* como se puede observar en la Figura 61. De este modo con una sola multiplicación se multiplicaran ambas matrices, suprimiendo

repetidas multiplicaciones lo que significa reducir considerablemente el tiempo al igual que en la multiplicación anterior.

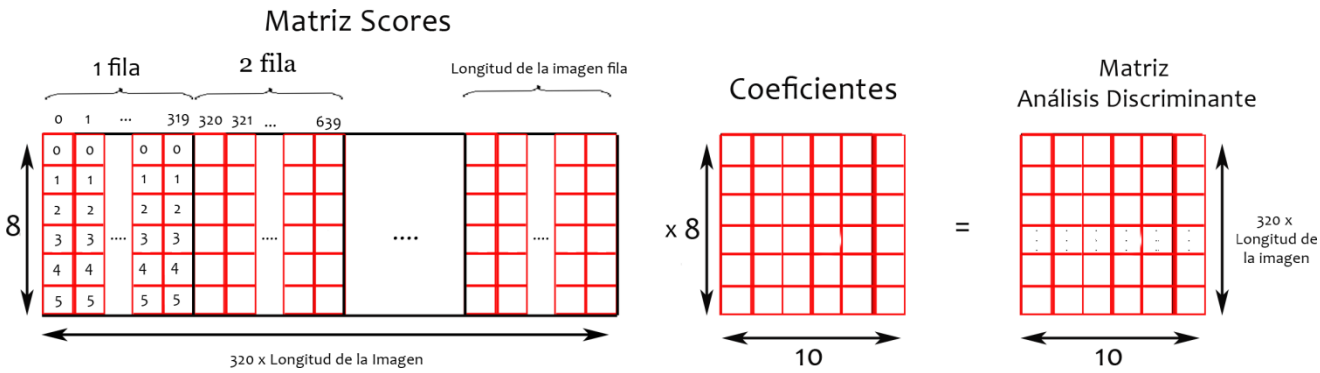


Figura 61. Multiplicación de la matriz *Scores* por la matriz Coeficientes mediante CUBLAS en el algoritmo PCA 1D15CE 8CP (CUDA).

Ahora solo falta sumarle a la matriz resultante el vector constante para acabar de aplicar el análisis discriminante a toda la matriz. Mediante el segundo *kernel* implementado llamado Fin se le sumará el vector constante y a la vez se calculará el máximo para cada fila de trabajo con lo cual se clasificará cada fila en una categoría.

En este último *kernel* se lanzarán 256 hilos y los bloques será el resultado de 320 por la longitud de la imagen dividido entre 256. El número de bloques se realizará de esta manera ya que la parte de encontrar el máximo se realizará de manera secuencial recorriendo la fila accedida. Por lo tanto, cada hilo accederá a una fila y el número de bloques corresponderá con cuantas filas haya divido entre el número de hilos lanzados. Una vez encontrado la posición del máximo para una fila accedida este valor será con que el cual se clasifique esta fila. La Figura 62 muestra un ejemplo de lo realizado en este último *kernel*.

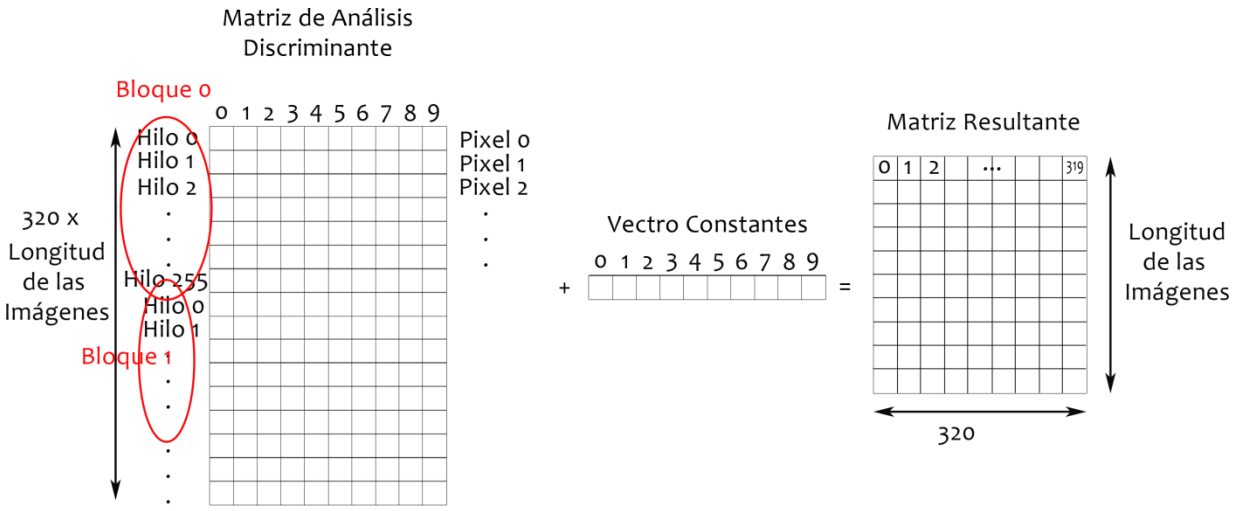


Figura 62. Suma del vector constantes más cálculo de la posición del máximo para la matriz resultante en el algoritmo PCA 1D15CE 8CP (CUDA).

Una vez aplicado el algoritmo PCA 1D15CE 8CP, se obtendrá una matriz bidimensional donde cada valor de la matriz resultante representará a que clase pertenece esa posición o ese pixel, se procederá a la visualización del resultado. Para ello, antes de todo habrá que enviar la matriz resultante que se encuentra alojada en memoria de la GPU a la CPU mediante el mecanismo inverso al utilizado al principio. Una vez se tenga la matriz resultante en la CPU el siguiente paso es visualizar la imagen mediante la librería de OpenCV al igual que se hizo en el apartado 4.2 Implementación en C++

Después de aplicarlo en un tomo se volverá a ejecutar el algoritmo pero para los otros dos tomos con la única modificación de sus rutas a la hora de cargar cada una de las imágenes hiperespectrales. Por lo tanto, el resultado final de los tres tomos serán tres imágenes que se observan en la Figura 63.

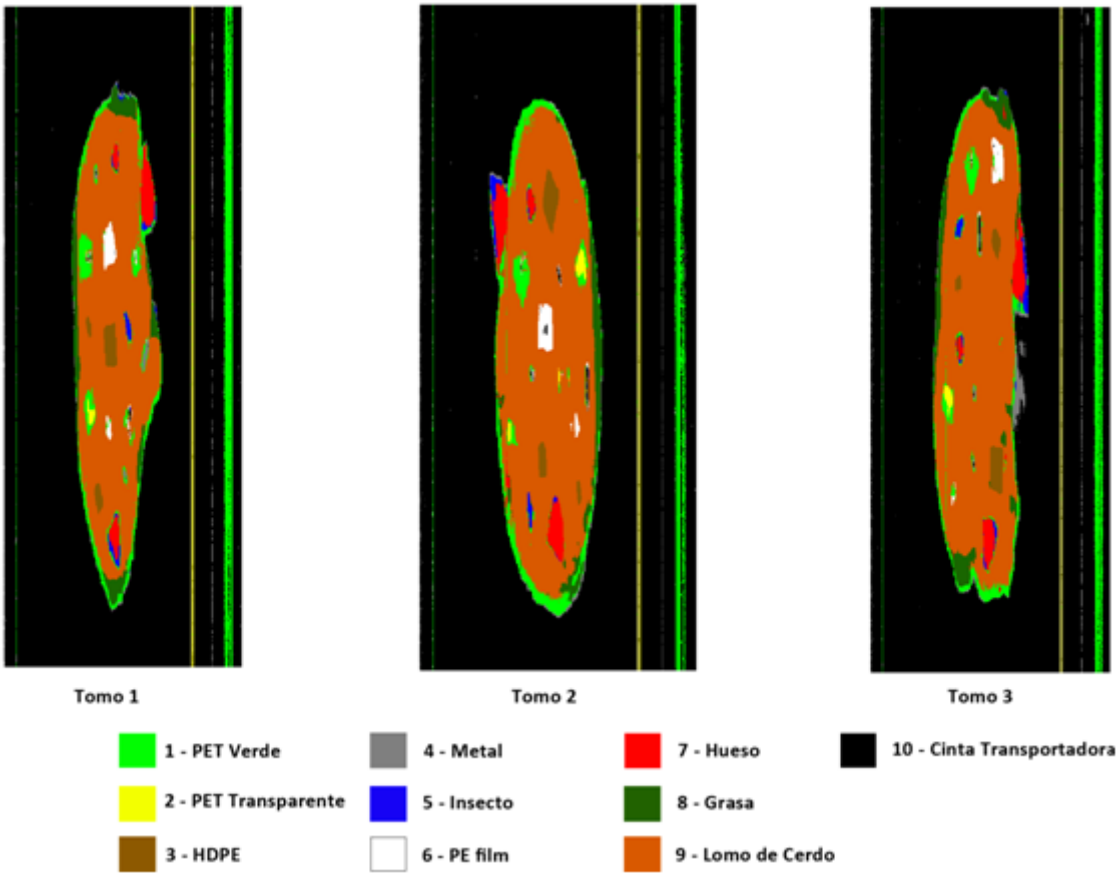


Figura 63. Imágenes Finales tras aplicar el modelo PCA 1D15CE 8CP (CUDA).

5. Resultados

Los algoritmos de clasificación de imágenes hiperespectrales se ejecutaron en dos ordenadores distintos para comprobar la diferencia de tiempos de ejecución. El propósito principal de utilizar dos ordenadores con configuraciones diferentes, tanto CPUs como memoria RAM, fue para comprobar la diferencia de tiempos entre distintas CPUs. Además de utilizar dos ordenadores también se utilizaron dos GPUs distintas para la comparación de tiempos.

El primer ordenador utilizado presentará las siguientes características principales:

- Intel Pentium i5 2500k a 3.3 GHz.
- 8 Gigabyte de RAM.
- Sistema Operativo de 64 bits.
- GeForce GTX 560 Ti [31], cuyas especificaciones se pueden ver en la Figura 64. Especificaciones de la GeForce GTX 560 Ti.

Arquitectura	Fermi
Transistores	1950M
Frecuencia GPU	822 Mhz
Frecuencia del Procesador	1645 Mhz
Núcleos CUDA	384
SMs	8
Frecuencia de la memoria	4008 Mhz
Memorita Total	1024 Mbytes
Bus de datos (bits)	256 bits
Ancho de banda de la memoria (GB/s)	128,3 GB/s

Figura 64. Especificaciones de la GeForce GTX 560 Ti.

El segundo ordenador utilizado presentará las siguientes características principales:

- Pentium Core Duo 2 a 2.2 GHz.
- 2 Gigabyte de RAM.
- Sistema operativo de 32 bits.
- GeForce GTX 260 [32], cuyas especificaciones se pueden ver en la Figura 65. Especificaciones de la GeForce GTX 260.

Arquitectura	**Fermi**
Transistores	**1400M**
Frecuencia GPU	**576 Mhz**
Frecuencia del Procesador	**1242 Mhz**
Núcleos CUDA	**192**
SMs	**8**
Frecuencia de la memoria	**1998 Mhz**
Memorita Total	**896 Mbytes**
Bus de datos (bits)	**448 bits**
Ancho de banda de la memoria (GB/s)	**119,9 GB/s**

Figura 65. Especificaciones de la GeForce GTX 260.

Los tiempos fueron medidos en todos los lenguajes en el que se implementaron los algoritmos incluido en Matlab para así poder comparar tiempos con lenguajes donde no se trabaja a tan bajo nivel.

Luego en este capítulo se mostrarán los tiempos de ejecución para cada algoritmo en cada lenguaje, es decir, estará formado por tres capítulos uno para cada lenguaje, *MatLab*, C++ y CUDA. Donde en cada apartado se mostrará los tiempos para los algoritmos PCA CE 6CP y PCA 1D15 8CP.

5.1 Tiempos en MatLab

Los tiempos mostrados a continuación solo miden el tiempo que se tarda en procesar cada uno de los algoritmos, es decir, no están incluidos ni los tiempos de lectura de archivos ni de imágenes ni la representación de la imagen final.

Ordenador 1	Tomo 1	Tomo 2	Tomo 3
PCA CE 6CP	**28,74 s**	**23,33 s**	**24,85 s**
PCA 1D15 8CP	**32,98 s**	**29,28 s**	**32,66 s**

Tabla 1. Tiempos en Matlab con la configuración del primer ordenador.

Ordenador 2	Tomo 1	Tomo 2	Tomo 3
PCA CE 6CP	**50,63 s**	**56,14 s**	**62,60 s**
PCA 1D15 8CP	**72,26 s**	**65,65 s**	**69,75 s**

Tabla 2. Tiempos en Matlab con la configuración del segundo ordenador.

5.2 Tiempos en C++

Los tiempos mostrados a continuación solo miden el tiempo que se tarda en procesar cada uno de los algoritmos, es decir, no están incluidos ni los tiempos de lectura de archivos ni de imágenes ni la representación de la imagen final.

Ordenador 1	Tomo 1	Tomo 2	Tomo 3
PCA CE 6CP	**8,66 s**	**7,03 s**	**7,82 s**
PCA 1D15 8CP	**214,88 s**	**132,43 s**	**159,86 s**

Tabla 3. Tiempos en C++ con la configuración del primer ordenador.

Ordenador 2	Tomo 1	Tomo 2	Tomo 3
PCA CE 6CP	**23,63 s**	**20,79 s**	**22,65 s**
PCA 1D15 8CP	**423,65 s**	**420,77 s**	**422,54 s**

Tabla 4. Tiempos en C++ con la configuración del segundo ordenador.

5.3 Tiempos en CUDA

Los tiempos mostrados a continuación solo miden el tiempo que se tarda en procesar cada uno de los algoritmos, es decir, no están incluidos ni los tiempos de lectura de archivos ni de imágenes ni la representación de la imagen final.

Ordenador 1	Tomo 1	Tomo 2	Tomo 3
PCA CE 6CP	**0,023 s**	**0,019 s**	**0,022 s**
PCA 1D15 8CP	**0,085 s**	**0,077 s**	**0,082s**

Tabla 5. Tiempos en CUDA con la configuración del primer ordenador.

Ordenador 2	Tomo 1	Tomo 2	Tomo 3
PCA CE 6CP	**0,064 s**	**0,056 s**	**0,064 s**
PCA 1D15 8CP	**0,215 s**	**0,0191 s**	**0,213 s**

Tabla 6. Tiempos en CUDA con la configuración del segundo ordenador.

5.4 Comparativa de Tiempos entre C++ y CUDA

En este apartado se mostrarán unas gráficas donde se compararán los tiempos del lenguaje C++ y CUDA, es decir, se comparará para el mismo algoritmo y bajos las mismas condiciones, primer ordenador o segundo ordenador, los dos lenguajes, el secuencial y el paralelo.

5.4.1 Primera Configuración (Pentium i5 2500k & GTX 560Ti)

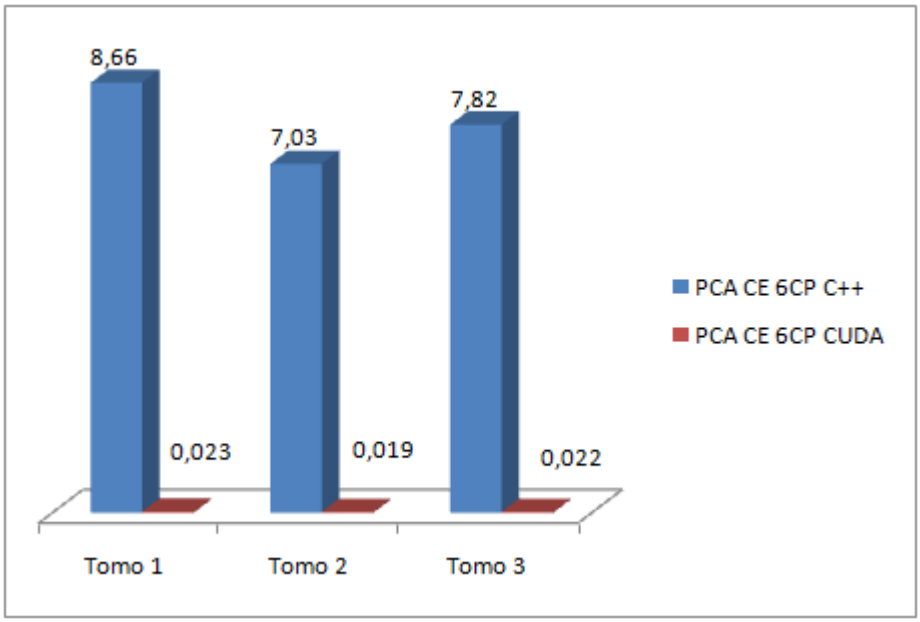


Figura 66. Comparación de tiempos entre el lenguaje C++ y CUDA para el algoritmo PCA CE 6CP con la configuración del primer ordenador.

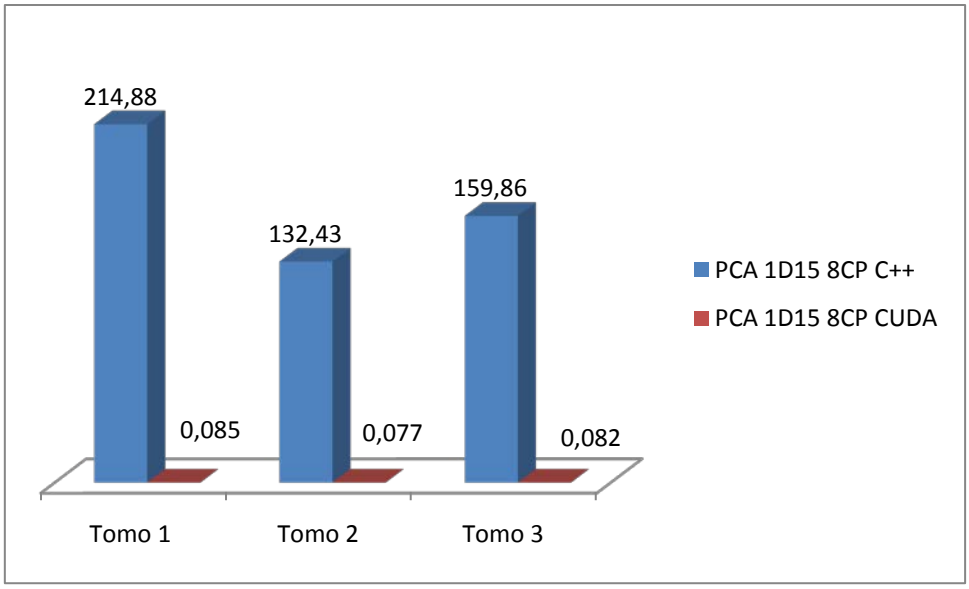


Figura 67. Comparación de tiempos entre el lenguaje C++ y CUDA para el algoritmo PCA 1D15 CE 8CP con la configuración del primer ordenador.

5.4.2 Segunda Configuración (Pentium Core 2 Duo & GTX 260)

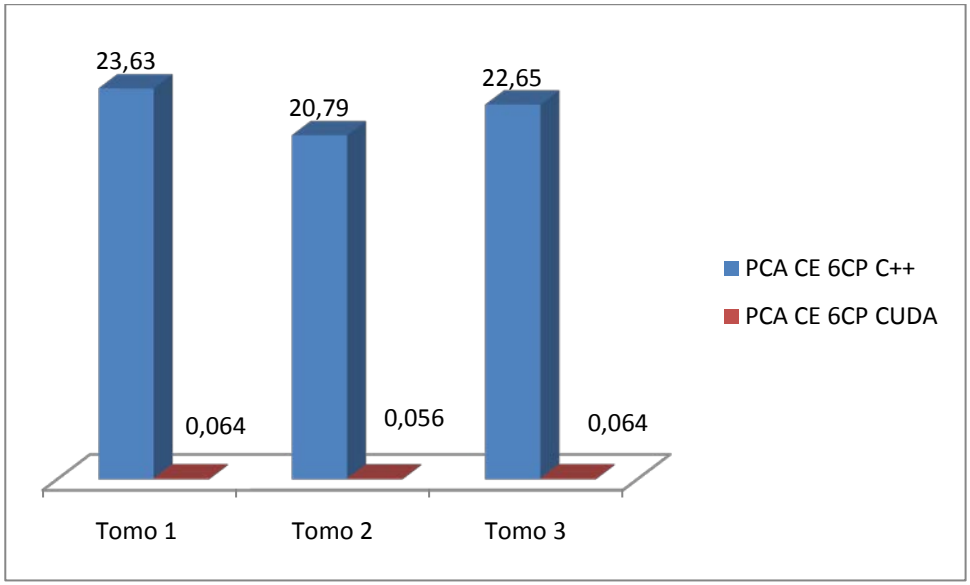


Figura 68.Comparación de tiempos entre el lenguaje C++ y CUDA para el algoritmo PCA CE 6CP con la configuración del segundo ordenador.

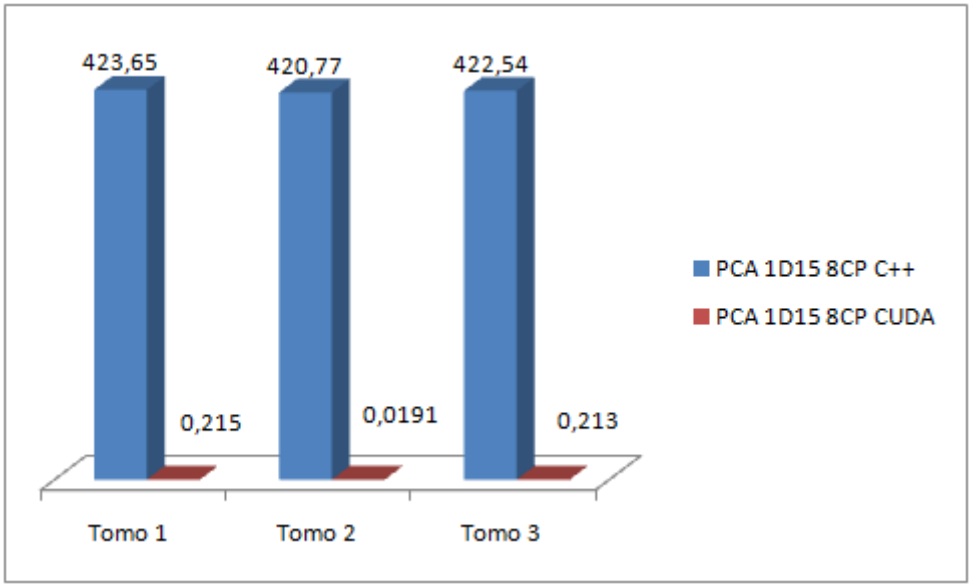


Figura 69. Comparación de tiempos entre el lenguaje C++ y CUDA para el algoritmo PCA 1D15 CE 8CP con la configuración del primer ordenador.

6. Conclusiones y líneas futuras

En el presente capítulo se presentarán las conclusiones extraídas tras la finalización del proyecto, estudiando y analizando los resultados obtenidos del apartado 5. Resultados. Posteriormente se expondrá los problemas que han ido surgiendo a lo largo de su desarrollo hasta poder lograr unos resultados válidos, con la finalidad de tener una aplicación práctica. Y por último el capítulo termina con la exposición de las posibles líneas futuras que se han identificado.

6.1 Conclusiones

En el presente proyecto se ha presentado el estudio de una alternativa para la aceleración de unos algoritmos de clasificación de imágenes hiperespectrales. No solo se implementaron en la alternativa elegida, CUDA, sino que se implementaron en diferente lenguajes, *MatLab* y C++, con la que se pudieron comparar tiempos entre lenguajes secuenciales, *MatLab* y C++, y el paralelo, CUDA.

Debido a que se pudieron implementar los algoritmos de clasificación de imágenes hiperespectrales mediante la alternativa elegida, CUDA, se pudo demostrar que estos algoritmos son totalmente paralelizables con la posible reducción de tiempos en su ejecución.

En el apartado 5. Resultados se puede comprobar los distintos tiempos que se han utilizado para cada algoritmo y bajo qué condiciones. Además se puede apreciar la considerable mejora en los tiempos de ejecución de CUDA frente a C++. Para medir esta mejora se ha utilizado el factor de rendimiento, *speed up*, que relaciona el tiempo de ejecución del algoritmo secuencial entre el paralelo. En la Figura 70 y Figura 71 se puede comprobar los ratios que han dado.

Ordenador 1	Tomo 1	Tomo 2	Tomo 3
Speed Up PCA CE 6CP	**376,52174**	**370**	**355,45455**
Speed Up PCA 1D15CE 8CP	**2528**	**1719,8701**	**1949,5122**

Figura 70. Speed Up del algoritmo PCA CE 6CP en el primer ordenador.

Ordenador 2	Tomo 1	Tomo 2	Tomo 3
Speed Up PCA CE 6CP	**369,21875**	**371,25**	**353,90625**
Speed Up PCA 1D15CE 8CP	**1970,4651**	**22029,843**	**1983,7559**

Figura 71. Speed Up del algoritmo PCA 1D15CE 8CP en el segundo ordenador.

Tal fue el éxito que se propuso por parte de la empresa el participar en un artículo para The International Association for Spectral Imaging (IASIM), con motivo de la cuarta conferencia de imagen espectral celebrada por IASIM. El objetivo de IASIM es proporcionar un foro unificado abierto e interactivo para el intercambio de información e ideas dentro de la comunidad de imagen espectral, sin importar el rango de la imagen espectral (ultravioleta, visible, cerca del medio o de espectrometría de infrarrojo lejano, o espectrometría de masas) o rango espacial (sensores microscópicos, macroscópicos y remotos).

En cuanto a los objetivos se puede decir que el principal de este proyecto final de carrera se ha cumplido y con creces, debido a la drástica reducción de los tiempos de ejecución. Además se ha cumplido otro, que es el introducir el uso de las GPUs en el ámbito de industrial.

6.2 Problemas Surgidos

Los problemas surgidos en este proyecto final de carrera se reducen a entender y comprender, primeramente los algoritmos de clasificación de imágenes hiperespectrales. Seguidamente de elegir una alternativa óptima que acelere los algoritmos. Una vez elegida la alternativa, comprender la arquitectura de la GPU y el lenguaje CUDA. Debido a la estructura de planteamiento para poder entender los algoritmos se implementaron en otros lenguajes de programación surgiendo algunos problemas que se exponen a continuación.

En *MatLab* el problema surgido una vez que se comprendió totalmente el funcionamiento del algoritmo y se implemento el modelo PCA CE 6CP, fue que al comprobar los datos suministrados para el modelo PCA 1D15CE 8CP no eran los correctos y tuvieron que reenviar los archivos adecuados.

En C++ el mayor problema fue como leer las imágenes hiperespectrales y representar la imagen final. Para ello, se tuvo que hacer una búsqueda de bibliotecas que se adaptaran a los requisitos que se necesitaban, es decir, que fueran gratuitas, sencillas, solo se utilizaría para leer y representar, y que se pudieran utilizar con fines empresariales sin ningún problema, ya que no era solo con fines de investigación para la Universidad Politécnica de Valencia, Instituto de Instrumentación para Imagen Molecular, sino que se implementaría en Ainia Centro Tecnológico.

Gracias a la implementación en C++ el cambio a CUDA fue más asequible, ya que únicamente había que cambiar la parte del algoritmo a paralelizar. Otro de los problemas en CUDA fue determinar el número óptimo de hilos y bloques a lanzar para cada uno de los *kernels*. Esto llevo a un proceso de prueba y error para maximizar la eficiencia y el rendimiento de la GPU mediante la herramienta *Visual Profiler*.

Por último, tras acabar la implementación en CUDA y mostrar los resultados a la empresa surgieron algunos imprevistos. Estos improvistos fueron que las imágenes hiperespectrales enviadas por la empresa no correspondían con las imágenes reales que se obtenían de la cámara hiperespectral. Por lo tanto, para simular que las imágenes hiperespectrales enviadas en un primer momento fueran de la cámara, se modificaron las imágenes hiperespectrales de tal manera para que así fueran. Una vez modificadas las imágenes hiperespectrales, se altera levemente los algoritmos para adaptarse a este cambio.

Este cambio consiste en modificar la manera de guardar y de acceder en memoria de las imágenes hiperespectrales.

6.3 Líneas Futuras

A pesar de que el objetivo del proyecto ha sido alcanzado, cabe destacar que todavía se pueden tomar otros pasos para mejorar el presente trabajo, en concreto se ha identificado las siguientes mejoras:

- Este tipo de inspección alimentaria hiperespectral para el control de calidad de alimentos no solo es para los filetes de lomos sino que se puede extender para cualquier alimento de una manera eficaz y rápida.
- Implementación de un sistema en tiempo real con extracción de cuerpos extraños. En él se adquirirá las imágenes hiperespectrales directamente de la cámara en vez de leerlas del disco. Tras adquirir las imágenes hiperespectrales se le aplicará el algoritmo correspondiente para clasificar lo adquirido por la cámara y en el caso de detectar algún cuerpo extraño proceder a su expulsión. El problema de la velocidad de la cinta dependerá de los fps de la cámara y el tiempo que tarde en clasificar las líneas adquiridas, que no será muy alto entorno a milisegundos.
- Realizar un software con interfaz gráfica mediante algún lenguaje orientado a objetos de alto nivel como puede ser C# y acoplar CUDA a este software mediante algún wrapper como puede ser Cudafy.
- Otra línea de trabajo será formar a los trabajadores de Ainia Centro Tecnológico en este tipo de tecnología para que así puedan usarla en otro tipo de proyectos. Con el objetivo de reducir tiempos, ya que a veces los tiempos de procesados son críticos e imposibilitan realizar algún tipo de algoritmo lo que conlleva a cancelar el proyecto.

7. Anexos

7.1 Código desarrollado para el Algoritmo PCA CE 6CP - MatLab

```
%-------------------------------------------------------------
% Universidad Politécnica de Valencia
% Escuela Técnica Superior de Ingenieros de Telecomunicación
%-------------------------------------------------------------
% Proyecto Final de Carrera: Aceleración de algoritmos de
% visión hiperespectral mediante GPU
%-------------------------------------------------------------
% Nombre del archivo: Algoritmo PCA CE 6CP
%-------------------------------------------------------------

%-------------------------------------------------------------
% Matriz con los coeficientes del Análisis de las principales
%componentes PCA   6CP
%-------------------------------------------------------------

 pca =load('PCA.txt');
 [t, r]=size(pca);

%-----------------------------------------------------------
% Matriz con los coeficientes del Análisis Discriminante
%-----------------------------------------------------------
coeficientes=load('coeficientes.txt');

%-----------------------------------------------------------
% Vector con las constantes del Análisis Discriminante
%-----------------------------------------------------------
constantes=load('constantes.txt');

%-----------------------------------------
% Se almacena las imágenes hiperespectrales
%-----------------------------------------
imagenes=cell(1,256);

for i=1:10
ruta_imagen=sprintf('C:\\Lomo_Todo_1\\Img_00%d.tif',i-1);
imagen=imread(ruta_imagen);
imagen =double(imagen);

imagenes{i}= imagen;
end

for i=11:100
ruta_imagen =sprintf('C:\\Lomo_Todo_1\\Img_0%d.tif',i-1);
imagen =imread(ruta_imagen);
imagen =double(imagen);

imagenes{i}= imagen;
end

for i=101:256
ruta_imagen =sprintf('C:\\Lomo_Todo_1\\Img_%d.tif',i-1);
imagen =imread(ruta_imagen);
imagen =double(imagen);

imagenes{i}= imagen;
```

```
end

[m, n]=size(imagen);
[h, r, t]=size(imagenes);

%-------------------------------------------------------------------
% Cálculo de Media y Varianza de la Matriz con los distintos cuerpos
%extraños
%-------------------------------------------------------------------

Modelo =load('Matriz.txt','unicode');
[m, n]=size(Modelo);

media=mean(Modelo);           %Obtengo la media para cada columna
varianza= var(Modelo,1);       %Obtengo la varianza para cada columna

%-------------------------------------------------------------------
% Se aplica el algoritmo PCA CE 6CP a las imágenes hiperespectrales
%-------------------------------------------------------------------

time=clock;

for i=1:m
    for j=1:n
        for h=1:t
            %Accedo a cada pixel de cada imagen para formar mi fila
           fila(h)=imagenes{h}(i,j);
        end

      %Aplico Centrado y escalado
        fila=(fila-media)./sqrt(varianza);
      %Aplico PCA y obtengo una fila de scores
        scores=fila*pca;
      %Aplico AD
        analisis=scores*coeficientes;
        analisis=analisis+ constantes;
      % Obtengo la posición del máximo
        [Valor,posicion]=max(analisis);
      %Obtengo una matriz con las distintas clasificaciones
        ImagenNueva(i,j)=posicion;
    end
end

tiempo1=etime(clock,time);    %Para ver cuánto tarda mi algoritmo

%----------------------------
% Creación de la nueva imagen
%----------------------------

rgb=zeros(m,n,3);

% Grupos de materiales para la clasificación

%1=verde       Green PET          %6=blanco       PE film
%2=amarillo     Transp PET         %7=rojo         Bone
%3=marron      HD PE              %8=verde oscuro  Fat
%4=gris         Metal              %9=naranja      Pork Loin
%5=azul         Insect             %10=negro       Background
```

```
%Escribiendo el color dependiendo a que clasificación pertenece

for i=1:m
    for j=1:n
       switch ImagenNueva(i,j)

            case 1
                rgb(i,j,1)=0;
                rgb(i,j,2)=1;
                rgb(i,j,3)=0;

            case 2
                rgb(i,j,1)=1;
                rgb(i,j,2)=1;
                rgb(i,j,3)=0;

            case 3
                rgb(i,j,1)=0.55;
                rgb(i,j,2)=0.35;
                rgb(i,j,3)=0;

            case 4
                rgb(i,j,1)=0.5;
                rgb(i,j,2)=0.5;
                rgb(i,j,3)=0.5;

            case 5
                rgb(i,j,1)=0;
                rgb(i,j,2)=0;
                rgb(i,j,3)=1;

            case 6
                rgb(i,j,1)=1;
                rgb(i,j,2)=1;
                rgb(i,j,3)=1;

            case 7
                rgb(i,j,1)=1;
                rgb(i,j,2)=0;
                rgb(i,j,3)=0;

            case 8
                rgb(i,j,1)=0.1;
                rgb(i,j,2)=0.4;
                rgb(i,j,3)=0;

            case 9
                rgb(i,j,1)=0.85;
                rgb(i,j,2)=0.35;
                rgb(i,j,3)=0;

            case 10
                rgb(i,j,1)=0;
                rgb(i,j,2)=0;
                rgb(i,j,3)=0;

       end
    end
end
```

```
%-----------------------------------------
% Se muestra la imagen resultante y se guarda
%-----------------------------------------

imshow(rgb);
imwrite(rgb, 'lomo1.jpg');
```

7.2 Código desarrollado para el Algoritmo PCA 1D15CE 8CP - MatLab

```
%-------------------------------------------------------------
% Universidad Politécnica de Valencia
% Escuela Técnica Superior de Ingenieros de Telecomunicación
%-------------------------------------------------------------
% Proyecto Final de Carrera: Aceleración de algoritmos de
% visión hiperespectral mediante GPU
%-------------------------------------------------------------
% Nombre del archivo: Algoritmo PCA 1D15CE 8CP
%-------------------------------------------------------------

%-------------------------------------------------------------
% Matriz con los coeficientes del Análisis de las principales
%componentes PCA   8CP
%-------------------------------------------------------------
pca =load('PCA.txt');
[t, r]=size(pca);

%-------------------------------------------------------
% Matriz con los coeficientes del Análisis Discriminante
%-------------------------------------------------------
coeficientes=load('coeficientes.txt');

%-------------------------------------------------------
% Vector con las constantes del Análisis Discriminante
%-------------------------------------------------------
constantes=load('constantes.txt');

%-----------------------------------------------------------------------
% Matriz con los coeficientes de la primera derivada de Saviztky-Golay
%-----------------------------------------------------------------------
derivada=load('Mat_SG_1D51.txt');


%-----------------------------------------
% Se almacena las imágenes hiperespectrales
%-----------------------------------------
imagenes=cell(1,256);

for i=1:10
ruta_imagen=sprintf('C:\\Lomo_Todo_1\\Img_00%d.tif',i-1);
imagen=imread(ruta_imagen);
imagen =double(imagen);

imagenes{i}= imagen;
end
```

```
for i=11:100
ruta_imagen =sprintf('C:\\Lomo_Todo_1\\Img_0%d.tif',i-1);
imagen =imread(ruta_imagen);
imagen =double(imagen);

imagenes{i}= imagen;
end

for i=101:256
ruta_imagen =sprintf('C:\\Lomo_Todo_1\\Img_%d.tif',i-1);
imagen =imread(ruta_imagen);
imagen =double(imagen);

imagenes{i}= imagen;
end

[m, n]=size(imagen);
[h, r, t]=size(imagenes);

%-------------------------------------------------------------
% Cálculo de Media y Varianza de la Matriz con los distintos cuerpos
%extraños
%-------------------------------------------------------------

Modelo =load('Matriz.txt','unicode');
[m, n]=size(Modelo);

ModeloDerivado = Modelo*derivada;

media=mean(ModeloDerivado);     %Obtengo la media para cada columna
varianza= var(ModeloDerivado,1);%Obtengo la varianza para cada columna


%-------------------------------------------------------
% Se aplica el algoritmo PCA 1D15CE 8CP a las imágenes
%hiperespectrales
%-------------------------------------------------------
time=clock;

for i=1:m
    for j=1:n
        for h=1:t
            %Obtengo cada pixel de cada imagen
           fila(h)=imagenes{h}(i,j);
        end

%Aplico primera derivada de S.G orden 2 y con una ventana de 15 puntos
        fila_derivada=fila*derivada;
%Aplico Centrado y escalado
        fila_derivada =( fila_derivada -media)./sqrt(varianza);
%Aplico PCA y obtengo una fila de scores
        scores=fila*pca;
%Aplico AD
        analisis=scores*coeficientes;
        analisis=analisis+ constantes;
% Obtengo la posición del máximo
        [Valor,posicion]=max(analisis);
%Obtengo una matriz con las distintas clasificaciones
        ImagenNueva(i,j)=posicion;
```

```
    end
end

tiempo1=etime(clock,time);    %Para ver cuánto tarda mi algoritmo

%----------------------------
% Creación de la nueva imagen
%----------------------------

rgb=zeros(m,n,3);

% Grupos de materiales para la clasificación

%1=verde       Green PET           %6=blanco       PE film
%2=amarillo     Transp PET          %7=rojo          Bone
%3=marron      HD PE               %8=verde oscuro  Fat
%4=gris         Metal               %9=naranja       Pork Loin
%5=azul         Insect              %10=negro        Background

%Escribiendo el color dependiendo a que clasificación pertenece

for i=1:m
    for j=1:n
       switch ImagenNueva(i,j)

            case 1
                rgb(i,j,1)=0;
                rgb(i,j,2)=1;
                rgb(i,j,3)=0;

            case 2
                rgb(i,j,1)=1;
                rgb(i,j,2)=1;
                rgb(i,j,3)=0;

            case 3
                rgb(i,j,1)=0.55;
                rgb(i,j,2)=0.35;
                rgb(i,j,3)=0;

            case 4
                rgb(i,j,1)=0.5;
                rgb(i,j,2)=0.5;
                rgb(i,j,3)=0.5;

            case 5
                rgb(i,j,1)=0;
                rgb(i,j,2)=0;
                rgb(i,j,3)=1;

            case 6
                rgb(i,j,1)=1;
                rgb(i,j,2)=1;
                rgb(i,j,3)=1;

            case 7
                rgb(i,j,1)=1;
                rgb(i,j,2)=0;
                rgb(i,j,3)=0;
```

```
            case 8
                rgb(i,j,1)=0.1;
                rgb(i,j,2)=0.4;
                rgb(i,j,3)=0;

            case 9
                rgb(i,j,1)=0.85;
                rgb(i,j,2)=0.35;
                rgb(i,j,3)=0;

            case 10
                rgb(i,j,1)=0;
                rgb(i,j,2)=0;
                rgb(i,j,3)=0;

        end
    end
end
%-------------------------------------------
% Se muestra la imagen resultante y se guarda
%-------------------------------------------

imshow(rgb);
imwrite(rgb, 'lomo1SG.jpg');
```

7.3 Código desarrollado para el Algoritmo PCA CE 6CP - C++

7.3.1 Programa Principal

```
%-------------------------------------------------------------
% Universidad Politécnica de Valencia
% Escuela Técnica Superior de Ingenieros de Telecomunicación
%-------------------------------------------------------------
% Proyecto Final de Carrera: Aceleración de algoritmos de
% visión hiperespectral mediante GPU
%-------------------------------------------------------------
% Nombre del archivo: Algoritmo PCA CE 6CP
%-------------------------------------------------------------

#define _CRT_SECURE_NO_WARNINGS   // Para que el compilador no haga
caso de los warnings de los sprintf y strcpy

//-------------------------------------
//includes para las distintas librerias
//-------------------------------------
#include <stdio.h>
#include <math.h>
//operaciones de entrada/salida en el lenguaje de programación C++
#include <iostream>
/manejadores de los archivos de entrada y salida C++
#include <iomanip>
using namespace std;
//libreria para escritura y lectura de archivos
#include <fstream>
//libreria para manejo de cadenas
#include <string.h>
//para poder acceder al tiempo del sistema
#include <time.h>
```

```
//----------------------------
//Variables Globales definidas
//----------------------------
const int Filas=256;
const int Clases=10;
const int Componentes=6;

//--------------------------------------
//Declaración de las funciones a utilizar
//--------------------------------------
extern "C"
void leerArchivo(float *vector, const char *ruta,const int filas);
extern "C"
void leerMatriz(float **vector,const char *ruta, const int filas,const
int columnas);
extern "C"
void imprimir(float *vector, int const tamaño);
extern "C"
void imprimirMatriz(float **vector, int const fila,const int columna);
extern "C"
void CreacionMatriz(float **matriz,const int filas,const int
columnas);
extern "C"
void libero(float **vector,const int filas);
extern "C"
void Cargaimagenes(const char *ruta,float **imag);
extern "C"
int columnaImagen(char *ruta);
extern "C"
int filaImagen(char *ruta);
extern "C"
void Creacion3D(float ***matriz,const int profundiad,const int
filas,const int columnas);
extern "C"
void lectura3D(float ***matriz,int profundidad,int fila,int
columna,char *buffer,char *ruta1,char *ruta);
extern "C"
void mostrarImagen(float **Imagen,int fila,int columna);
extern "C"
void CreacionImagen(float ***matriz,float **imagen,float *media,float
*varianza,float *constantes,float **coeficientes,float **PCA,int
filas,int componentes,int clases,int fila_imagen,int columna_imagen);
extern "C"
void libero3D(float ***matriz, const int filas, const int columnas);

//------------------
//Programa Principal
//------------------

int main(int argc, char **argv)
{
//-----------------------------------------------------------------
//Declaracion de variables donde guardare los datos de los archivos
cargados con su correspondiente reserva de memoria.
//-----------------------------------------------------------------

      float *MediaModelo=new float[Filas];
      float *VarianzaModelo=new float[Filas];
      float *ConstantesModelo=new float[Clases];
```

```
float **CoeficientesModelo=new float *[Componentes];
//reservo memoria para la matriz bidimnesional, diciendole
(matriz,fila,columna)
CreacionMatriz(CoeficientesModelo,Componentes,Clases);
float **PCAModelo=new float *[Filas];
//reservo memoria para la matriz[][] bidimnesional, diciendole
(matriz,fila,columna)
CreacionMatriz(PCAModelo,Filas,Componentes);

//---------------------------------------------------------------------
//Rutas donde se encuentran los archivos a cargar!! Podria meterlas ya
directamente en la funcion leer
//---------------------------------------------------------------------

      const char *rutamedia="C:\\Users\\-\\Desktop\\c++\\Modelo PCA CE
6CP_Lomo1\\media.txt";
      const int lonM=strlen(rutamedia);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *media=new char[lonM+1];
 //copio rutamedia a una nueva area de memoria apuntada por media
      strcpy(media,rutamedia);

      const char *rutavarianza="C:\\Users\\-\\Desktop\\c++\\Modelo PCA
CE 6CP_Lomo1\\varianza.txt";
      const int lonV=strlen(rutavarianza);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *varianza=new char[lonV+1];
   //copio rutavarianza a una nueva area de memoria apuntada por
varianza
      strcpy(varianza,rutavarianza);

      const char *rutaconstantes="C:\\Users\\-\\Desktop\\c++\\Modelo
PCA CE 6CP_Lomo1\\constantes.txt";
      const int lonC=strlen(rutaconstantes);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *constantes=new char[lonC+1];
//copio rutaconstantes a una nueva area de memoria apuntada por
constantes
      strcpy(constantes,rutaconstantes);

      const char *rutacoeficientes="C:\\Users\\-\\Desktop\\c++\\Modelo
PCA CE 6CP_Lomo1\\coeficientes.txt";
      const int lonCo=strlen(rutacoeficientes);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *coeficientes=new char[lonCo+1];
    //copio rutacoeficientes a una nueva area de memoria apuntada por
coeficientes
      strcpy(coeficientes,rutacoeficientes);

      const char *rutaPCA="C:\\Users\\-\\Desktop\\c++\\Modelo PCA CE
6CP_Lomo1\\PCA.txt";
      const int lonP=strlen(rutaPCA);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *PCA=new char[lonP+1];
//copio rutaPCA a una nueva area de memoria apuntada por PCA
strcpy(PCA,rutaPCA);
```

```
//-------------------------------------------------
//Cargar los archivos necesarios para el algoritmo
//-------------------------------------------------

//Tengo en MediaModelo mis datos,(vector donde se almacenan los datos,
ruta,tamaño de la fila a leer)
      leerArchivo(MediaModelo,media,Filas);
//Libero memoria media,ruta.
      delete media;

//Tengo en VarianzaModelo mis datos,(vecotr donde se almacenan los
datos, ruta,tamaño de la fila a leer)
      leerArchivo(VarianzaModelo,varianza,Filas);
//Libero memoria varianza,ruta.
      delete varianza;

//Tengo en ConstantesModelo mis datos,(vector donde se almacenan los
datos, ruta,tamaño de la fila a leer)
      leerArchivo(ConstantesModelo,constantes,Clases);
      //Libero memoria constantes,ruta.
      delete constantes;

//Tengo en CoeficienteModelo mis datos,(matriz donde se almacenan los
datos, ruta,tamaño de la fila a leer,tamaño de la columna a leer)
      leerMatriz(CoeficientesModelo,coeficientes,Componentes,Clases);
//Libero memoria coeficientes,ruta.
      delete coeficientes;

//Tengo en PCAModelo mis datos,(matriz donde se almacenan los datos,
ruta,tamaño de la fila a leer,tamaño de la columna a leer)
      leerMatriz(PCAModelo,PCA,Filas,Componentes);
//Libero memoria PCA,ruta.
      delete PCA;

//-------------------------------------------------
//Rutas donde se encuentran las IMAGENES a cargar!!
//-------------------------------------------------

      const char *R1="C:\\Users\\-\\Desktop\\c++\\Modelo PCA CE
6CP_Lomo1\\Lomo_Todo_1\\Img_00";
      const int lonR1=strlen(R1);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *ruta1=new char[lonR1+1];
//copio R1 a una nueva area de memoria apuntada por ruta1
      strcpy(ruta1,R1);

      const char *R=".tif";
      const int lonR=strlen(R);
      char *ruta=new char[lonR+1];
//copio R a una nueva area de memoria apuntada por ruta
      strcpy(ruta,R);
//asigno memoria,en buffer formare la ruta completa. Es 1+1 para
guardar el caracter '/0' y el 0 ---->Img_00'0'.tif;
      char *buffer=new char[lonR1+1+1+lonR];
```

```
//---------------------------
//Creo    mi cubo de imagenes
//---------------------------

//Leo la primera imagen para saber su tamaño y asi crear el cubo de
imagenes, sabiendo que el resto tiene las mismas dimensiones

      float ***Matriz=new float **[Filas];
      sprintf(buffer,"%s%d%s",ruta1,0,ruta);
      int fila_imagen=filaImagen(buffer);
      int columna_imagen=columnaImagen(buffer);
//reservo memoria para la matriz[][][] tridimnesional, diciendole
(matriz3D,Profundidad,fila,columna)
      Creacion3D(Matriz,Filas,fila_imagen,columna_imagen);

//---------------------------
//Carga del cubo de imágenes
//---------------------------
      time_t leer=clock();
//Guardo en Matriz todas las imagenes, (matriz3D donde guarda las
imagenes,profundidad,filas de la imagen,columna de la imagen
      lectura3D(Matriz,Filas,fila_imagen,columna_imagen,buffer,ruta1,r
uta);

//,buffer donde formo ruta completa,trozo de la ruta primera ruta1,
ruta la segunda parte .tif!!
      printf("Lectura 3D %f\n",(float)(clock()-leer)/CLOCKS_PER_SEC);
//libero memoria de la ruta
      delete ruta;
//libero memoria del buffer
      delete buffer;
//libero memoria de la ruta1
      delete ruta1;

//--------------------
//Creo mi imagen final
//--------------------

//reservo memoria para la matriz bidimnesional, diciendole
(matriz,fila,columna)
      float **ImagenFinal=new float *[fila_imagen];
      CreacionMatriz(ImagenFinal,fila_imagen,columna_imagen);
```

```
//-------------------------------------------------------
//Aplico centrado-escalado,PCA,análisis discriminante y su
clasificación, es decir, el algoritmo PCA CE 6CP
//-------------------------------------------------------

//activo el timer para saber cuanto tarda mi algoritmo.
	time_t start_time=clock();

//crea mi imagen-->matriz bidimensional,(matriz3D donde estan todas
las imagenes,donde almaceno mi ImagenFinal,archivo media,archivo
varianza,archivo constant4es,archivo coeficientes,archivopca,
	//,Filas de los tamaños de los vectores que aprovecho ya que es
igual que profundidad lo utilizo para las dos cosas, Componentes que
tenemos en el analisis, Clases que hay en la clasificaicion,
	//,filas que contiene la imgane,columnas que contiene la imagen)
	CreacionImagen(Matriz,ImagenFinal,MediaModelo,VarianzaModelo,Con
stantesModelo,CoeficientesModelo,PCAModelo,Filas,Componentes,Clases,fi
la_imagen,columna_imagen);

//Calculo el tiempo que ha pasado desde la activacion hasta ahora en
segundos.
	float time1 = (float) (clock() - start_time) / CLOCKS_PER_SEC;

	cout << "La duracion del algoritmo es de " << time1 << "
segundos\n";


//-----------------------------------------------------------------
//Muestro por pantalla la imagen final tras pasar el tratamiento y
almaceno la imagen en una ruta predefinida
//-----------------------------------------------------------------

//(matriz bidimensional donde esta la ImagenFinal, filas de la
imagen,columnas de la imagen)
	mostrarImagen(ImagenFinal,fila_imagen,columna_imagen);

//-------------------------------------------------------
//Liberacion de Memoria de todos los punteros restantes
//-------------------------------------------------------

	delete VarianzaModelo;
	delete MediaModelo;
	delete ConstantesModelo;
	libero(CoeficientesModelo,Componentes);
	libero(PCAModelo,Filas);
	libero(ImagenFinal,fila_imagen);
	libero3D(Matriz,Filas,fila_imagen);
	system("PAUSE");

}
```

7.3.2 Funciones

```
%------------------------------------------------------------
% Universidad Politécnica de Valencia
% Escuela Técnica Superior de Ingenieros de Telecomunicación
%------------------------------------------------------------
% Proyecto Final de Carrera: Aceleración de algoritmos de
% visión hiperespectral mediante GPU
%------------------------------------------------------------
% Nombre del archivo: Algoritmo PCA CE 6CP
%------------------------------------------------------------

// Para que el compilador no haga caso de los warnings de los sprintf
y strcpy
#define _CRT_SECURE_NO_WARNINGS

//--------------------------------------
//includes para las distintas librerias
//--------------------------------------
#include <stdio.h>
//libreria de matematicas
#include <math.h>
//operaciones de entrada/salida en el lenguaje de programación C++
#include <iostream>
//manejadores de los archivos de entrada y salida C++
#include <iomanip>
using namespace std;
//libreria para escritura y lectura de archivos
#include <fstream>
//para poder acceder al tiempo del sistema
#include <time.h>

//---------------------------------------
//Declaración de las funciones a utilizar
//---------------------------------------
extern "C"
void Cargaimagenes(const char *ruta,float **imag);

//-------------------------------
//Implementacion de las funciones
//-------------------------------

//leo el archivo seguna la ruta expecificada y lo almaceno en el
vector
extern "C"
void leerArchivo(float *vector, const char *ruta,const int filas)
{
// abrir archivo para lectura

      ifstream fichero;
//Lo abro como entrada
      fichero.open(ruta,ios::in);
// verificar la apertura del archivo
    if ( !fichero ) {
        cerr << "Error al tratar de abrir el archivo"<<endl;
            exit(1);
    }
```

```
    // lectura de datos
      while ( ! fichero.eof()) {
            for(int i=0;i<filas;i++)
//Lee omitiendo blancos, es lo que quiero!!    y guardo en vector
            fichero >> vector[i];
      }
//cierro el descriptor
      fichero.close();
}

//leo un archivo que contiene dentro una matriz y lo almaceno en la
matriz bidimensional
extern "C"
void leerMatriz(float **vector,const char *ruta, const int filas,const
int columnas)
{
// abrir archivo para lectura
      ifstream fichero;
//Lo abro como entrada
      fichero.open(ruta,ios::in);

    // verificar la apertura del archivo
    if ( !fichero ) {
        cerr << "Error al tratar de abrir el archivo"<<endl;
            exit(1);
    }
    // lectura de datos
      while (! fichero.eof()) {
            for(int i =0; i<filas;i++){
                  for(int j=0;j<columnas;j++)
//Lee omitiendo blancos, es lo que quiero!! y guardo en la posicion
adecuada en la matriz
                  fichero >> vector[i][j];
            }
      }
//Cierrro el descriptor
      fichero.close();
}

//Esta función es para mostrar por pantalla el vector introducido
extern "C"
void imprimir(float *vector, int const tamaño)
{
//Me sirve para saber si los archivos introducios son leidos
correctamente
      for(int i=0;i<tamaño;i++)
            cout << vector[i] << "\n";
}

//Esta función es para mostrar por pantalla la matriz introducida
extern "C"
void imprimirMatriz(float **vector, int const fila,const int columna)
{
//Me sirve para saber si los archivos con una matriz dentro han sido
leidos correctamente
      for(int i=0;i<fila;i++){
            for(int j=0;j<columna;j++)
                  cout << vector[i][j] << "  ";
            cout << "\n";
      }
}
```

```
//Esta función sirve para reserva memoria de una matriz bidimensional
extern "C"
void CreacionMatriz(float **matriz,const int filas,const int columnas)
{
            for(int i=0;i<filas;i++)
            matriz[i] = new float [columnas];
}

//Esta función sirve para liberar memoria de una matriz bidimensional
extern "C"
void libero(float **vector,const int filas)
{
            for(int i=0;i<filas;i++)
            delete vector[i];

      delete vector;
}

//Esta función sirve para reserva memoria de una matriz tridimensional
extern "C"
void Creacion3D(float ***matriz,const int profundidad,const int
filas,const int columnas)
{
            for(int i=0;i<profundidad;i++)
            matriz[i] = new float *[filas];

      for(int i=0;i<profundidad;i++)
            for(int j=0;j<filas;j++)
                  matriz[i][j] = new float [columnas];
}

//Esta función sirve para liberar memoria de una matriz tridimensional
extern "C"
void libero3D(float ***matriz, const int profundidad, const int filas)
{
      for(int i=0;i<profundidad;i++)
            for(int j=0;j<filas;j++)
                  delete matriz[i][j];

      for (int i=0;i<profundidad;i++)
            delete matriz[i];

      delete matriz;
}

//Función que me almacena las 256 imagenes en un cubo de imagenes
extern "C"
void lectura3D(float ***matriz,int profundidad,int fila,int
columna,char *buffer,char *ruta1,char *ruta)
{
      for (int i=0;i<profundidad;i++){
//Hago esto porque el nombre de la imagen solo tiene un cero, por eso
los tengo que separar
            if(i==10)
                  ruta1="C:\\Users\\-\\Desktop\\c++\\Modelo PCA CE
6CP_Lomo1\\Lomo_Todo_1\\Img_0";

//Aqui no tiene nigun cero por eso los tengo que separar
            if(i==100)
                  ruta1="C:\\Users\\-\\Desktop\\c++\\Modelo PCA CE
6CP_Lomo1\\Lomo_Todo_1\\Img_";
```

```
//Creo con sprintf la ruta completa y la almaceno en buffer
            sprintf(buffer,"%s%d%s",ruta1,i,ruta);
//Me creo una variable auxiliar que almacenara una imagen
      float **Imagen=new float *[fila];
      CreacionMatriz(Imagen,fila,columna);

//Me guarda la imagen de la ruta introducida en la variable imagen y a
continuacion recorro dicha variable para guardarla en mi cubo de
imagenes
      Cargaimagenes(buffer,Imagen);
      for(int j=0;j<fila;j++)
                  for(int t=0;t<columna;t++)
                        matriz[i][j][t]=Imagen[j][t];

//Libero memoria de la variable auxiliar imagen
            libero(Imagen,fila);
      }
}

//Aplico algoritomo PCA CE 6CP
extern "C"
void CreacionImagen(float ***matriz,float **imagen,float *media,float
*varianza,float *constantes,float **coeficientes,float **PCA,int
filas,int componentes,int clases,int fila_imagen,int columna_imagen)
{
      //componentes=6
      //filas=256
      //clases=10
      float *adquisicion=new float[filas];
      float *scores=new float[componentes];
      float suma=0;
      float maximo=-100;
      int     posicion;

      time_t centrado;
      time_t coef;
      time_t analisis;
      float sumaCentrado=0;
      float sumaCoef=0;
      float sumaAnalisis=0;

      for(int h=0;h<fila_imagen;h++){
            for(int t=0;t<columna_imagen;t++){
                  centrado=clock();

                  for(int s=0;s<componentes;s++){
                        for(int r=0;r<filas;r++){
                  //Cogo cada pixel de cada una de las imagenes 256
                           adquisicion[r]=matriz[r][h][t];
                  //Realizo centrado y escalado y a la vez la
                  multiplicacion con PCA, almacenado las sumas
                  parciales y sus multiplicaciones en suma
                           suma=suma+((adquisicion[r]-
                              media[r])/sqrt(varianza[r]))*PCA[r][s];
                        }
            //Voy obteniendo los distintos scores
                        scores[s]=suma;
            //Reinicio las sumas y multiplicaciones parciales
                        suma=0;
                  }
```

```
                  sumaCentrado=sumaCentrado+(float)(clock()-
                        centrado)/CLOCKS_PER_SEC;

                  for(int i=0;i<clases;i++){
                        coef=clock();
                        for(int j=0;j<componentes;j++){
      //Hago la multiplicacion por los coeficientes y alamacenando sus
      sumas parciales como antes
                              suma=suma+scores[j]*coeficientes[j][i];
                        }
      //sumo la constantes que toca como nos indica la formula del
      Analisis discriminante
                        suma=suma+constantes[i];
                        sumaCoef=sumaCoef+(float)(clock()-
                              coef)/CLOCKS_PER_SEC;
                        analisis=clock();

                        if(maximo < suma){
            //Para obtener en que posicion se encuentra el maximo!!!
                              maximo=suma;
            //Empezamos desde cero!!! luego habra que sumarle siempre
            uno
                              posicion=i+1;
                        }
                        suma=0;
                  }

//Lo igualo a -100 porque hay a veces que los valores de la ecuacion
del analisis discriminante son todos negativos!! y por lo tanto no
entraria en estos casos, obteniendo una posicion equivocada de ese
pixel!!!
                  maximo=-100;
//Guardo en la posicion que toca al grupo que pertenece ese pixel
                  imagen[h][t]=(float)posicion;
                  sumaAnalisis=sumaAnalisis+(float)(clock()-
                        analisis)/CLOCKS_PER_SEC;
            }
      }
printf("Centrado y PCA tarda %f\n",sumaCentrado);
printf("Coeficientes  tarda %f\n",sumaCoef);
printf(" Analisis tarda %f\n",sumaAnalisis);
}
```

7.3.3 Imágenes

```
%------------------------------------------------------------
% Universidad Politécnica de Valencia
% Escuela Técnica Superior de Ingenieros de Telecomunicación
%------------------------------------------------------------
% Proyecto Final de Carrera: Aceleración de algoritmos de
% visión hiperespectral mediante GPU
%------------------------------------------------------------
% Nombre del archivo: Algoritmo PCA CE 6CP
%------------------------------------------------------------

//-------------------
//Librerias de OpenCV
//-------------------
#include "cv.h"
#include "highgui.h"
//-------------------------------------
//includes para las distintas librerias
//-------------------------------------
#include <stdio.h>
//operaciones de entrada/salida en el lenguaje de programación C++
#include <iostream>
//manejadores de los archivos de entrada y salida C++
#include <iomanip>
using namespace std;
//libreria para escritura y lectura de archivos
#include <fstream>
//libreria para manejo de cadenas
#include <string.h>

//------------------------------
//Implementacion de las funciones
//------------------------------

//Carga una imagen de una ruta especificada
extern "C"
void Cargaimagenes(const char *ruta,float **imag)
{
//inicializo imagen,tipo de variable definido en librerias OpenCV
IplImage* imagen=NULL;
// cargamos la imagen,ruta donde se encuentra la imagen,y con -1
indicamos que carga la imagen con los canales que tenga la
imagen(rgb,gris,...)
imagen=cvLoadImage(ruta,-1);

if(imagen==NULL)
      perror( "No se ha podido cargar la imagen correctamente\n");

//definimos la estructura para acceder a los datos de la imagen
//imagen->width=320
//imagen->height=887
CvScalar s;
```

```
	for(int i=0;i<imagen->height;i++){
		for(int j=0;j<imagen->width;j++){
	//cvGet2D definido en las librerias de OpenCV se obtiene el
	valor (i,j) del píxel, que queda almacenado en s.val[0] valor
	double;
			s=cvGet2D(imagen,i,j);
	//Convierto el dato    double en float y lo almaceno en la
	matriz bidimensional dada
			imag[i][j]=(float)s.val[0];
		}
	}
//liberar el espacio de memoria de la imagen
	cvReleaseImage(&imagen);
}

//Funcion que me da el tamaño de la columna de una imagen. Le pasamos
la ruta donde se encuentra dicha imagen
extern "C"
int columnaImagen(char *ruta)
{
IplImage* imagen=NULL;//inicializo imagen
imagen=cvLoadImage(ruta,-1);// cargamos la imagen,
if(imagen==NULL)
	perror( "No se ha podido cargar la imagen correctamente\n");

return(imagen->width); //devuelvo el valor de la columna. IplImage
contiene la clase->width donde se almacena dicho valor tipo int
}

//Funcion que me da el tamaño de la fila de una imagen. Le pasamos la
ruta donde se encuentra dicha imagen
extern "C"
int filaImagen(char *ruta)
{
IplImage* imagen=NULL;//inicializo imagen
imagen=cvLoadImage(ruta,-1);// cargamos la imagen,
if(imagen==NULL)
	perror( "No se ha podido cargar la imagen correctamente\n");

return(imagen->height);	//devuelvo el valor de la fila. IplImage
contiene la clase->height donde se almacena dicho valor tipo int
}

//Funcion que imprime por pantalla la matriz pasada. Los colores de
cada pixel dependera de a que grupo pertenezca
extern "C"
void mostrarImagen(float **Imagen,int fila,int columna)
{

//Reservo memoria para la imagen diciendole el tamaño,valor de los
pixeles, canales que tendra la imagen---> 3 es bgr(rgb)
IplImage* ImagenNueva=cvCreateImage(cvSize(columna,fila),
IPL_DEPTH_8U, 3);

//Para acceder a los pixeles de la imagen nueva
CvScalar s;
//1=verde		Green PET		6=blanco	PE film
//2=amarillo		Transp PET		7=rojo			Bone
//3=marron		HD PE			8=verde oscuro	Fat
//4=gris		Metal			9=naranja	Pork Loin
//5=azul		Insect			10=negro	Background
```

```
	for(int i=0;i<fila;i++){
		for(int j=0;j<columna;j++){
			switch((int)Imagen[i][j])
			{
			case 1:
//s es un vector donde cada componente es blue(0)-green(1)-red(2), y
segun que valores tenga estas componente tendra un color u otro
				s=cvScalar(0,255,0);
				break;
			case 2:
				s=cvScalar(0,255,255);
				break;
			case 3:
				s=cvScalar(0,90,141);
				break;
			case 4:
				s=cvScalar(128,128,128);
				break;
			case 5:
				s=cvScalar(255,0,0);
				break;
			case 6:
				s=cvScalar(255,255,255);
				break;
			case 7:
				s=cvScalar(0,0,255);
				break;
			case 8:
				s=cvScalar(0,103,26);
				break;
			case 9:
				s=cvScalar(0,90,218);
				break;
			case 10:
				s=cvScalar(0,0,0);
				break;
			default:
				s=cvScalar(0,0,0);
				break;
			}
//Asigno el color que corresponda a ese pixel(rgb) que esta en la
variable s
			cvSet2D(ImagenNueva,i,j,s);
		}
	}
//Creo nuna ventana con el titulo lomo1 y con el 1 diremos que se
ajuste a la imagen a dibujar
	cvNamedWindow("Lomo1",1);
//Muestra la imagen por pantalla creada
	cvShowImage("Lomo1",ImagenNueva);
// se pulsa tecla para terminar
	cvWaitKey(0);
// guardamos la imagen en una ruta predefinida
	cvSaveImage("C:\\Users\\-\\Desktop\\c++\\Modelo PCA CE
6CP_Lomo1\\Lomo1_C++.jpg",ImagenNueva);
// destruimos todas las ventanas
	cvDestroyAllWindows();
	//liberamos memoria de la imagen
	cvReleaseImage(&ImagenNueva);
}
```

7.4 Código desarrollado para el Algoritmo PCA 1D15CE 8CP - C++

7.4.1 Programa Principal

```
%-------------------------------------------------------------
% Universidad Politécnica de Valencia
% Escuela Técnica Superior de Ingenieros de Telecomunicación
%-------------------------------------------------------------
% Proyecto Final de Carrera: Aceleración de algoritmos de
% visión hiperespectral mediante GPU
%-------------------------------------------------------------
% Nombre del archivo: Algoritmo PCA 1D15CE 8CP
%-------------------------------------------------------------


#define _CRT_SECURE_NO_WARNINGS   // Para que el compilador no haga
caso de los warnings de los sprintf y strcpy

//--------------------------------------
//includes para las distintas librerias
//--------------------------------------
#include <stdio.h>
#include <math.h>
//operaciones de entrada/salida en el lenguaje de programación C++
#include <iostream>
/manejadores de los archivos de entrada y salida C++
#include <iomanip>
using namespace std;
//libreria para escritura y lectura de archivos
#include <fstream>
//libreria para manejo de cadenas
#include <string.h>
//para poder acceder al tiempo del sistema
#include <time.h>
//----------------------------
//Variables Globales definidas
//----------------------------
const int Filas=256;
const int Clases=10;
const int Componentes=6;

//---------------------------------------
//Declaración de las funciones a utilizar
//---------------------------------------
extern "C"
void leerArchivo(float *vector, const char *ruta,const int filas);
extern "C"
void leerMatriz(float **vector,const char *ruta, const int filas,const
int columnas);
extern "C"
void imprimir(float *vector, int const tamaño);
extern "C"
void imprimirMatriz(float **vector, int const fila,const int columna);
extern "C"
void CreacionMatriz(float **matriz,const int filas,const int
columnas);
extern "C"
void libero(float **vector,const int filas);
```

```
extern "C"
void Cargaimagenes(const char *ruta,float **imag);
extern "C"
int columnaImagen(char *ruta);
extern "C"
int filaImagen(char *ruta);
extern "C"
void Creacion3D(float ***matriz,const int profundiad,const int
filas,const int columnas);
extern "C"
void lectura3D(float ***matriz,int profundidad,int fila,int
columna,char *buffer,char *ruta1,char *ruta);
extern "C"
void mostrarImagen(float **Imagen,int fila,int columna);
extern "C"
void CreacionImagen(float ***matriz,float **imagen,float *media,float
*varianza,float *constantes,float **coeficientes,float **PCA,int
filas,int componentes,int clases,int fila_imagen,int columna_imagen);
extern "C"
void libero3D(float ***matriz, const int filas, const int columnas);

//------------------
//Programa Principal
//------------------

int main(int argc, char **argv)
{
//-----------------------------------------------------------------
//Declaracion de variables donde guardare los datos de los archivos
cargados con su correspondiente reserva de memoria.
//-----------------------------------------------------------------

      float *MediaModelo=new float[Filas];
      float *VarianzaModelo=new float[Filas];
      float *ConstantesModelo=new float[Clases];


float **CoeficientesModelo=new float *[Componentes];
//reservo memoria para la matriz bidimnesional, diciendole
(matriz,fila,columna)
CreacionMatriz(CoeficientesModelo,Componentes,Clases);
float **PCAModelo=new float *[Filas];
//reservo memoria para la matriz[][] bidimnesional, diciendole
(matriz,fila,columna)
CreacionMatriz(PCAModelo,Filas,Componentes);
float **SGModelo=new float *[Filas];
//reservo memoria para la matriz bidimnesional, diciendole
(matriz,fila,columna)
CreacionMatriz(SGModelo,Filas,Filas);
//-----------------------------------------------------------------
//Rutas donde se encuentran los archivos a cargar!! Podria meterlas ya
directamente en la funcion leer
//-----------------------------------------------------------------

      const char *rutamedia="C:\\Users\\-\\Desktop\\c++\\Modelo PCA CE
6CP_Lomo1\\media.txt";
      const int lonM=strlen(rutamedia);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *media=new char[lonM+1];
 //copio rutamedia a una nueva area de memoria apuntada por media
      strcpy(media,rutamedia);
```

```
      const char *rutavarianza="C:\\Users\\-\\Desktop\\c++\\Modelo PCA
CE 6CP_Lomo1\\varianza.txt";
      const int lonV=strlen(rutavarianza);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *varianza=new char[lonV+1];
   //copio rutavarianza a una nueva area de memoria apuntada por
varianza
      strcpy(varianza,rutavarianza);

      const char *rutaconstantes="C:\\Users\\-\\Desktop\\c++\\Modelo
PCA CE 6CP_Lomo1\\constantes.txt";
      const int lonC=strlen(rutaconstantes);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *constantes=new char[lonC+1];
//copio rutaconstantes a una nueva area de memoria apuntada por
constantes
      strcpy(constantes,rutaconstantes);

      const char *rutacoeficientes="C:\\Users\\-\\Desktop\\c++\\Modelo
PCA CE 6CP_Lomo1\\coeficientes.txt";
      const int lonCo=strlen(rutacoeficientes);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *coeficientes=new char[lonCo+1];
    //copio rutacoeficientes a una nueva area de memoria apuntada por
coeficientes
      strcpy(coeficientes,rutacoeficientes);

      const char *rutaPCA="C:\\Users\\-\\Desktop\\c++\\Modelo PCA CE
6CP_Lomo1\\PCA.txt";
      const int lonP=strlen(rutaPCA);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *PCA=new char[lonP+1];
//copio rutaPCA a una nueva area de memoria apuntada por PCA
strcpy(PCA,rutaPCA);

      const char *rutaSG="C:\\Users\\-\\Desktop\\c++\\Modelo PCA
1D15CE 8CP_Lomo1\\Mat_SG_1D51.txt";
      const int lonSG=strlen(rutaSG);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *SG=new char[lonSG+1];
    //copio rutaSG a una nueva area de memoria apuntada por SG
      strcpy(SG,rutaSG);

//-------------------------------------------------
//Cargar los archivos necesarios para el algoritmo
//-------------------------------------------------

//Tengo en MediaModelo mis datos,(vector donde se almacenan los datos,
ruta,tamaño de la fila a leer)
      leerArchivo(MediaModelo,media,Filas);
//Libero memoria media,ruta.
      delete media;

//Tengo en VarianzaModelo mis datos,(vecotr donde se almacenan los
datos, ruta,tamaño de la fila a leer)
      leerArchivo(VarianzaModelo,varianza,Filas);
//Libero memoria varianza,ruta.
      delete varianza;
```

```
//Tengo en ConstantesModelo mis datos,(vector donde se almacenan los
datos, ruta,tamaño de la fila a leer)
      leerArchivo(ConstantesModelo,constantes,Clases);
      //Libero memoria constantes,ruta.
      delete constantes;

//Tengo en CoeficienteModelo mis datos,(matriz donde se almacenan los
datos, ruta,tamaño de la fila a leer,tamaño de la columna a leer)
      leerMatriz(CoeficientesModelo,coeficientes,Componentes,Clases);
//Libero memoria coeficientes,ruta.
      delete coeficientes;

//Tengo en PCAModelo mis datos,(matriz donde se almacenan los datos,
ruta,tamaño de la fila a leer,tamaño de la columna a leer)
      leerMatriz(PCAModelo,PCA,Filas,Componentes);
//Libero memoria PCA,ruta.
      delete PCA;

//Tengo en PCAModelo mis datos,(matriz donde se almacenan los datos,
ruta,tamaño de la fila a leer,tamaño de la columna a leer)
      leerMatriz(SGModelo,SG,Filas,Filas);
      delete SG;

//-------------------------------------------------
//Rutas donde se encuentran las IMAGENES a cargar!!
//-------------------------------------------------

      const char *R1="C:\\Users\\-\\Desktop\\c++\\Modelo PCA CE
6CP_Lomo1\\Lomo_Todo_1\\Img_00";
      const int lonR1=strlen(R1);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *ruta1=new char[lonR1+1];
//copio R1 a una nueva area de memoria apuntada por ruta1
      strcpy(ruta1,R1);

      const char *R=".tif";
      const int lonR=strlen(R);
      char *ruta=new char[lonR+1];
//copio R a una nueva area de memoria apuntada por ruta
      strcpy(ruta,R);
//asigno memoria,en buffer formare la ruta completa. Es 1+1 para
guardar el caracter '/0' y el 0 ---->Img_00'0'.tif;
      char *buffer=new char[lonR1+1+1+lonR];


//---------------------------
//Creo    mi cubo de imagenes
//---------------------------

//Leo la primera imagen para saber su tamaño y asi crear el cubo de
imagenes, sabiendo que el resto tiene las mismas dimensiones

      float ***Matriz=new float **[Filas];
      sprintf(buffer,"%s%d%s",ruta1,0,ruta);
      int fila_imagen=filaImagen(buffer);
      int columna_imagen=columnaImagen(buffer);
//reservo memoria para la matriz[][][] tridimnesional, diciendole
(matriz3D,Profundidad,fila,columna)
      Creacion3D(Matriz,Filas,fila_imagen,columna_imagen);
```

```
//--------------------------
//Carga del cubo de imágenes
//--------------------------
      time_t leer=clock();
//Guardo en Matriz todas las imagenes, (matriz3D donde guarda las
imagenes,profundidad,filas de la imagen,columna de la imagen
      lectura3D(Matriz,Filas,fila_imagen,columna_imagen,buffer,ruta1,r
uta);

//,buffer donde formo ruta completa,trozo de la ruta primera ruta1,
ruta la segunda parte .tif!!
      printf("Lectura 3D %f\n",(float)(clock()-leer)/CLOCKS_PER_SEC);
//libero memoria de la ruta
      delete ruta;
//libero memoria del buffer
      delete buffer;
//libero memoria de la ruta1
      delete ruta1;

//--------------------
//Creo mi imagen final
//--------------------

//reservo memoria para la matriz bidimnesional, diciendole
(matriz,fila,columna)
      float **ImagenFinal=new float *[fila_imagen];
      CreacionMatriz(ImagenFinal,fila_imagen,columna_imagen);

//-----------------------------------------------------------
//Aplico centrado-escalado,PCA,análisis discriminante y su
clasificación, es decir, el algoritmo PCA CE 6CP
//-----------------------------------------------------------

//activo el timer para saber cuanto tarda mi algoritmo.
      time_t start_time=clock();

//crea mi imagen-->matriz bidimensional,(matriz3D donde estan todas
las imagenes,donde almaceno mi ImagenFinal,archivo media,archivo
varianza,archivo constant4es,archivo coeficientes,archivopca,
      //,Filas de los tamaños de los vectores que aprovecho ya que es
igual que profundidad lo utilizo para las dos cosas, Componentes que
tenemos en el analisis, Clases que hay en la clasificaicion,
      //,filas que contiene la imgane,columnas que contiene la imagen)
      CreacionImagen(Matriz,ImagenFinal,MediaModelo,VarianzaModelo,Con
stantesModelo,CoeficientesModelo,PCAModelo,Filas,Componentes,Clases,fi
la_imagen,columna_imagen);

//Calculo el tiempo que ha pasado desde la activacion hasta ahora en
segundos.
      float time1 = (float) (clock() - start_time) / CLOCKS_PER_SEC;

      cout << "La duracion del algoritmo es de " << time1 << "
segundos\n";
```

```
//-------------------------------------------------------------------
//Muestro por pantalla la imagen final tras pasar el tratamiento y
almaceno la imagen en una ruta predefinida
//-------------------------------------------------------------------

//(matriz bidimensional donde esta la ImagenFinal, filas de la
imagen,columnas de la imagen)
      mostrarImagen(ImagenFinal,fila_imagen,columna_imagen);

//-----------------------------------------------------
//Liberacion de Memoria de todos los punteros restantes
//-----------------------------------------------------

      delete VarianzaModelo;
      delete MediaModelo;
      delete ConstantesModelo;
      libero(CoeficientesModelo,Componentes);
      libero(PCAModelo,Filas);
      libero(ImagenFinal,fila_imagen);
      libero3D(Matriz,Filas,fila_imagen);
      system("PAUSE");

}
```

7.4.2 Funciones

```
%-------------------------------------------------------------
% Universidad Politécnica de Valencia
% Escuela Técnica Superior de Ingenieros de Telecomunicación
%-------------------------------------------------------------
% Proyecto Final de Carrera: Aceleración de algoritmos de
% visión hiperespectral mediante GPU
%-------------------------------------------------------------
% Nombre del archivo: Algoritmo PCA CE 6CP
%-------------------------------------------------------------

// Para que el compilador no haga caso de los warnings de los sprintf
y strcpy
#define _CRT_SECURE_NO_WARNINGS

//-------------------------------------
//includes para las distintas librerias
//-------------------------------------
#include <stdio.h>
//libreria de matematicas
#include <math.h>
//operaciones de entrada/salida en el lenguaje de programación C++
#include <iostream>
//manejadores de los archivos de entrada y salida C++
#include <iomanip>
using namespace std;
//libreria para escritura y lectura de archivos
#include <fstream>
//para poder acceder al tiempo del sistema
#include <time.h>


//-------------------------------------
```

```
//Declaración de las funciones a utilizar
//---------------------------------------
extern "C"
void Cargaimagenes(const char *ruta,float **imag);

//-------------------------------
//Implementacion de las funciones
//-------------------------------

//leo el archivo seguna la ruta expecificada y lo almaceno en el
vector
extern "C"
void leerArchivo(float *vector, const char *ruta,const int filas)
{
// abrir archivo para lectura

      ifstream fichero;
//Lo abro como entrada
      fichero.open(ruta,ios::in);
// verificar la apertura del archivo
    if ( !fichero ) {
        cerr << "Error al tratar de abrir el archivo"<<endl;
            exit(1);
    }


    // lectura de datos
      while ( ! fichero.eof()) {
            for(int i=0;i<filas;i++)
//Lee omitiendo blancos, es lo que quiero!!    y guardo en vector
            fichero >> vector[i];
      }
//cierro el descriptor
      fichero.close();
}

//leo un archivo que contiene dentro una matriz y lo almaceno en la
matriz bidimensional
extern "C"
void leerMatriz(float **vector,const char *ruta, const int filas,const
int columnas)
{
// abrir archivo para lectura
      ifstream fichero;
//Lo abro como entrada
      fichero.open(ruta,ios::in);

    // verificar la apertura del archivo
    if ( !fichero ) {
        cerr << "Error al tratar de abrir el archivo"<<endl;
            exit(1);
    }
    // lectura de datos
      while (! fichero.eof()) {
            for(int i =0; i<filas;i++){
                  for(int j=0;j<columnas;j++)
//Lee omitiendo blancos, es lo que quiero!! y guardo en la posicion
adecuada en la matriz
                  fichero >> vector[i][j];
            }
      }
```

```
//Cierrro el descriptor
	fichero.close();
}

//Esta función es para mostrar por pantalla el vector introducido
extern "C"
void imprimir(float *vector, int const tamaño)
{
//Me sirve para saber si los archivos introducios son leidos
correctamente
	for(int i=0;i<tamaño;i++)
		cout << vector[i] << "\n";
}

//Esta función es para mostrar por pantalla la matriz introducida
extern "C"
void imprimirMatriz(float **vector, int const fila,const int columna)
{
//Me sirve para saber si los archivos con una matriz dentro han sido
leidos correctamente
	for(int i=0;i<fila;i++){
		for(int j=0;j<columna;j++)
			cout << vector[i][j] << "  ";
		cout << "\n";
	}
}
//Esta función sirve para reserva memoria de una matriz bidimensional
extern "C"
void CreacionMatriz(float **matriz,const int filas,const int columnas)
{
		for(int i=0;i<filas;i++)
		matriz[i] = new float [columnas];
}

//Esta función sirve para liberar memoria de una matriz bidimensional
extern "C"
void libero(float **vector,const int filas)
{
		for(int i=0;i<filas;i++)
		delete vector[i];

	delete vector;
}

//Esta función sirve para reserva memoria de una matriz tridimensional
extern "C"
void Creacion3D(float ***matriz,const int profundidad,const int
filas,const int columnas)
{
		for(int i=0;i<profundidad;i++)
		matriz[i] = new float *[filas];

	for(int i=0;i<profundidad;i++)
		for(int j=0;j<filas;j++)
			matriz[i][j] = new float [columnas];
}

//Esta función sirve para liberar memoria de una matriz tridimensional
extern "C"
void libero3D(float ***matriz, const int profundidad, const int filas)
{
```

```
	for(int i=0;i<profundidad;i++)
		for(int j=0;j<filas;j++)
			delete matriz[i][j];

	for (int i=0;i<profundidad;i++)
		delete matriz[i];

	delete matriz;
}

//Función que me almacena las 256 imagenes en un cubo de imagenes
extern "C"
void lectura3D(float ***matriz,int profundidad,int fila,int
columna,char *buffer,char *ruta1,char *ruta)
{
	for (int i=0;i<profundidad;i++){
//Hago esto porque el nombre de la imagen solo tiene un cero, por eso
los tengo que separar
		if(i==10)
			ruta1="C:\\Users\\-\\Desktop\\c++\\Modelo PCA CE
6CP_Lomo1\\Lomo_Todo_1\\Img_0";

//Aqui no tiene nigun cero por eso los tengo que separar
		if(i==100)
			ruta1="C:\\Users\\-\\Desktop\\c++\\Modelo PCA CE
6CP_Lomo1\\Lomo_Todo_1\\Img_";
//Creo con sprintf la ruta completa y la almaceno en buffer
		sprintf(buffer,"%s%d%s",ruta1,i,ruta);
//Me creo una variable auxiliar que almacenara una imagen
	float **Imagen=new float *[fila];
	CreacionMatriz(Imagen,fila,columna);

//Me guarda la imagen de la ruta introducida en la variable imagen y a
continuacion recorro dicha variable para guardarla en mi cubo de
imagenes
	Cargaimagenes(buffer,Imagen);
	for(int j=0;j<fila;j++)
			for(int t=0;t<columna;t++)
				matriz[i][j][t]=Imagen[j][t];

//Libero memoria de la variable auxiliar imagen
		libero(Imagen,fila);
	}
}

//Aplico algoritomo PCA CE 6CP
extern "C"
void CreacionImagen(float ***matriz,float **imagen,float *media,float
*varianza,float *constantes,float **coeficientes,float **PCA,int
filas,int componentes,int clases,int fila_imagen,int columna_imagen)
{
	//componentes=6
	//filas=256
	//clases=10
	float *adquisicion=new float[filas];
	float *scores=new float[componentes];
	float suma=0;
	float maximo=-100;
	int     posicion;

	time_t derivada;
```

```
time_t centrado;
time_t coef;
time_t analisis;
float sumaDerivada=0;
float sumaCentrado=0;
float sumaCoef=0;
float sumaAnalisis=0;

for(int h=0;h<fila_imagen;h++){
      for(int t=0;t<columna_imagen;t++){

            derivada=clock();
            for(int i=0;i<filas;i++){
                  for(int j=0;j<filas;j++)
                        suma=suma+matriz[j][h][t]*SG[j][i];
            adquisicion[i]=suma;
            suma=0;
            }

            centrado=clock();
            for(int s=0;s<componentes;s++){
                  for(int r=0;r<filas;r++){
            //Cogo cada pixel de cada una de las imagenes 256
                     adquisicion[r]=matriz[r][h][t];
            //Realizo centrado y escalado y a la vez la
            multiplicacion con PCA, almacenado las sumas
            parciales y sus multiplicaciones en suma
                     suma=suma+((adquisicion[r]-
                        media[r])/sqrt(varianza[r]))*PCA[r][s];
                  }
      //Voy obteniendo los distintos scores
                  scores[s]=suma;
      //Reinicio las sumas y multiplicaciones parciales
                  suma=0;
            }
            sumaCentrado=sumaCentrado+(float)(clock()-
                  centrado)/CLOCKS_PER_SEC;

            for(int i=0;i<clases;i++){
                  coef=clock();
                  for(int j=0;j<componentes;j++){
//Hago la multiplicacion por los coeficientes y alamacenando sus
sumas parciales como antes
                        suma=suma+scores[j]*coeficientes[j][i];
                  }
//sumo la constantes que toca como nos indica la formula del
Analisis discriminante
                  suma=suma+constantes[i];
                  sumaCoef=sumaCoef+(float)(clock()-
                        coef)/CLOCKS_PER_SEC;
                  analisis=clock();

                  if(maximo < suma){
      //Para obtener en que posicion se encuentra el maximo!!!
                        maximo=suma;
      //Empezamos desde cero!!! luego habra que sumarle siempre
      uno
                        posicion=i+1;
                  }
                  suma=0;
            }
```

```
//Lo igualo a -100 porque hay a veces que los valores de la ecuacion
del analisis discriminante son todos negativos!! y por lo tanto no
entraria en estos casos, obteniendo una posicion equivocada de ese
pixel!!!
                    maximo=-100;
//Guardo en la posicion que toca al grupo que pertenece ese pixel
                    imagen[h][t]=(float)posicion;
                    sumaAnalisis=sumaAnalisis+(float)(clock()-
                          analisis)/CLOCKS_PER_SEC;
              }
       }
printf("Centrado y PCA tarda %f\n",sumaCentrado);
printf("Coeficientes  tarda %f\n",sumaCoef);
printf(" Analisis tarda %f\n",sumaAnalisis);
}
```

7.4.3 Imágenes

```
%-------------------------------------------------------------
% Universidad Politécnica de Valencia
% Escuela Técnica Superior de Ingenieros de Telecomunicación
%-------------------------------------------------------------
% Proyecto Final de Carrera: Aceleración de algoritmos de
% visión hiperespectral mediante GPU
%-------------------------------------------------------------
% Nombre del archivo: Algoritmo PCA CE 6CP
%-------------------------------------------------------------

//-------------------
//Librerias de OpenCV
//-------------------
#include "cv.h"
#include "highgui.h"
//-------------------------------------
//includes para las distintas librerias
//-------------------------------------
#include <stdio.h>
//operaciones de entrada/salida en el lenguaje de programación C++
#include <iostream>
//manejadores de los archivos de entrada y salida C++
#include <iomanip>
using namespace std;
//libreria para escritura y lectura de archivos
#include <fstream>
//libreria para manejo de cadenas
#include <string.h>

//-------------------------------
//Implementacion de las funciones
//-------------------------------

//Carga una imagen de una ruta especificada
extern "C"
void Cargaimagenes(const char *ruta,float **imag)
{
//inicializo imagen,tipo de variable definido en librerias OpenCV
IplImage* imagen=NULL;
```

```
// cargamos la imagen,ruta donde se encuentra la imagen,y con -1
indicamos que carga la imagen con los canales que tenga la
imagen(rgb,gris,...)
imagen=cvLoadImage(ruta,-1);

if(imagen==NULL)
      perror( "No se ha podido cargar la imagen correctamente\n");

//definimos la estructura para acceder a los datos de la imagen
//imagen->width=320
//imagen->height=887
CvScalar s;

      for(int i=0;i<imagen->height;i++){
            for(int j=0;j<imagen->width;j++){
      //cvGet2D definido en las librerias de OpenCV se obtiene el
      valor (i,j) del píxel, que queda almacenado en s.val[0] valor
      double;
                  s=cvGet2D(imagen,i,j);
      //Convierto el dato     double en float y lo almaceno en la
      matriz bidimensional dada
                  imag[i][j]=(float)s.val[0];
            }
      }
//liberar el espacio de memoria de la imagen
      cvReleaseImage(&imagen);
}

//Funcion que me da el tamaño de la columna de una imagen. Le pasamos
la ruta donde se encuentra dicha imagen
extern "C"
int columnaImagen(char *ruta)
{
IplImage* imagen=NULL;//inicializo imagen
imagen=cvLoadImage(ruta,-1);// cargamos la imagen,
if(imagen==NULL)
      perror( "No se ha podido cargar la imagen correctamente\n");

return(imagen->width); //devuelvo el valor de la columna. IplImage
contiene la clase->width donde se almacena dicho valor tipo int
}

//Funcion que me da el tamaño de la fila de una imagen. Le pasamos la
ruta donde se encuentra dicha imagen
extern "C"
int filaImagen(char *ruta)
{
IplImage* imagen=NULL;//inicializo imagen
imagen=cvLoadImage(ruta,-1);// cargamos la imagen,
if(imagen==NULL)
      perror( "No se ha podido cargar la imagen correctamente\n");

return(imagen->height);       //devuelvo el valor de la fila. IplImage
contiene la clase->height donde se almacena dicho valor tipo int
}
```

```
//Funcion que imprime por pantalla la matriz pasada. Los colores de
cada pixel dependera de a que grupo pertenezca
extern "C"
void mostrarImagen(float **Imagen,int fila,int columna)
{

//Reservo memoria para la imagen diciendole el tamaño,valor de los
pixeles, canales que tendra la imagen---> 3 es bgr(rgb)
IplImage* ImagenNueva=cvCreateImage(cvSize(columna,fila),
IPL_DEPTH_8U, 3);

//Para acceder a los pixeles de la imagen nueva
CvScalar s;
//1=verde               Green PET          6=blanco       PE film
//2=amarillo            Transp PET         7=rojo               Bone
//3=marron              HD PE              8=verde oscuro Fat
//4=gris                Metal              9=naranja      Pork Loin
//5=azul                Insect             10=negro       Background

      for(int i=0;i<fila;i++){
            for(int j=0;j<columna;j++){
                  switch((int)Imagen[i][j])
                  {
                  case 1:
//s es un vector donde cada componente es blue(0)-green(1)-red(2), y
segun que valores tenga estas componente tendra un color u otro
                        s=cvScalar(0,255,0);
                        break;
                  case 2:
                        s=cvScalar(0,255,255);
                        break;
                  case 3:
                        s=cvScalar(0,90,141);
                        break;
                  case 4:
                        s=cvScalar(128,128,128);
                        break;
                  case 5:
                        s=cvScalar(255,0,0);
                        break;
                  case 6:
                        s=cvScalar(255,255,255);
                        break;
                  case 7:
                        s=cvScalar(0,0,255);
                        break;
                  case 8:
                        s=cvScalar(0,103,26);
                        break;
                  case 9:
                        s=cvScalar(0,90,218);
                        break;
                  case 10:
                        s=cvScalar(0,0,0);
                        break;
                  default:
                        s=cvScalar(0,0,0);
                        break;
                  }
//Asigno el color que corresponda a ese pixel(rgb) que esta en la
variable s
```

```
                    cvSet2D(ImagenNueva,i,j,s);
                }
        }
//Creo nuna ventana con el titulo lomo1 y con el 1 diremos que se
ajuste a la imagen a dibujar
        cvNamedWindow("Lomo1",1);
//Muestra la imagen por pantalla creada
        cvShowImage("Lomo1",ImagenNueva);
// se pulsa tecla para terminar
        cvWaitKey(0);
// guardamos la imagen en una ruta predefinida
        cvSaveImage("C:\\Users\\-\\Desktop\\c++\\Modelo PCA CE
6CP_Lomo1\\Lomo1_C++.jpg",ImagenNueva);
// destruimos todas las ventanas
        cvDestroyAllWindows();
        //liberamos memoria de la imagen
        cvReleaseImage(&ImagenNueva);
}
```

7.5 Código desarrollado para el Algoritmo PCA CE 6CP - CUDA

7.5.1 Programa Principal

```
%-------------------------------------------------------------
% Universidad Politécnica de Valencia
% Escuela Técnica Superior de Ingenieros de Telecomunicación
%-------------------------------------------------------------
% Proyecto Final de Carrera: Aceleración de algoritmos de
% visión hiperespectral mediante GPU
%-------------------------------------------------------------
% Nombre del archivo: Algoritmo PCA CE 6CP
%-------------------------------------------------------------


#define _CRT_SECURE_NO_WARNINGS   // Para que el compilador no haga
caso de los warnings de los sprintf y strcpy

//--------------------------------------
//includes para las distintas librerias
//--------------------------------------
#include <stdio.h>
#include <math.h>
//operaciones de entrada/salida en el lenguaje de programación C++
#include <iostream>
/manejadores de los archivos de entrada y salida C++
#include <iomanip>
using namespace std;
//libreria para escritura y lectura de archivos
#include <fstream>
//libreria para manejo de cadenas
#include <string.h>
//para poder acceder al tiempo del sistema
#include <time.h>
#include <cublas.h>
#include <cuda_runtime.h>
#include <cuda.h>
```

```
// Libreria con los kernels realizados
#include "kernel.cu"

//----------------------------
//Variables Globales definidas
//----------------------------
const int Filas=256;
const int Clases=10;
const int Componentes=6;

//-------------------
//Variables de la GPU
//-------------------

__device__   float* media_GPU;
__device__   float* varianza_GPU;
__device__   float* constantes_GPU;
__device__   float* coeficientes_GPU;
__device__   float* PCA_GPU;
__device__   float* Cubo_GPU;
__device__   float* Centrado_GPU;
__device__   float* Coef_GPU;
__device__   float* Matriz_GPU;
__device__   float* Scores_GPU;
__device__   float* Imagen_GPU;
__device__   float* Final_GPU;
__device__   float* Vector_GPU;

//---------------------------------------
//Declaración de las funciones a utilizar
//---------------------------------------
extern "C"
void leerArchivo(float *vector, const char *ruta,const int filas);
extern "C"
void leerMatriz(float *vector,const char *ruta, const int filas,const
int columnas);
extern "C"
void imprimir(float *vector, int const tamaño);
extern "C"
void imprimirMatriz(float *vector, int const fila,const int columna);
extern "C"
void imprimirMatriz3D(float *vector,int const profundidad,int const
fila,const int columna);
extern "C"
void Cargaimagenes(const char *ruta,float *imag);
extern "C"
int columnaImagen(char *ruta);
extern "C"
int filaImagen(char *ruta);
extern "C"
void lectura3D(float *matriz,int profundidad,int fila,int columna,char
*buffer,char *ruta1,char *ruta);
extern "C"
void mostrarImagen(float *Imagen,int fila,int columna);
extern "C"
void CreacionImagen(float *matriz,float *imagen,float *media,float
*varianza,float *constantes,float *coeficientes,float *PCA,int
filas,int componentes,int clases,int fila_imagen,int columna_imagen);
```

```
///GPU funciones
extern "C"
void memoriaGPU(float *vector, int tamaño);
extern "C"
void copiarGPU(float *origen, float *GPU,int tamaño);
extern "C"
void copiarMatrizGPU(float *origen, float *GPU,int fila,int columna);


int main(int argc, char **argv)
{

//---------------------------------------------------------------------
//Declaracion de variables donde guardare los datos de los archivos
cargados con su correspondiente reserva de memoria.
//---------------------------------------------------------------------

      float *MediaModelo=new float[Filas];
      float *VarianzaModelo=new float[Filas];
      float *ConstantesModelo=new float[Clases];
//reservo memoria para la matriz bidimnesional, diciendole
(matriz,fila,columna)
      float *CoeficientesModelo=new float [Componentes];
//reservo memoria para la matriz[][] bidimnesional, diciendole
(matriz,fila,columna)
      float *PCAModelo=new float *[Filas];


//---------------------------------------------------------------------
//Rutas donde se encuentran los archivos a cargar!! Podria meterlas ya
directamente en la funcion leer
//---------------------------------------------------------------------

      const char *rutamedia="C:\\Users\\-\\Desktop\\c++\\Modelo PCA CE
6CP_Lomo1\\media.txt";
      const int lonM=strlen(rutamedia);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *media=new char[lonM+1];
 //copio rutamedia a una nueva area de memoria apuntada por media
      strcpy(media,rutamedia);

      const char *rutavarianza="C:\\Users\\-\\Desktop\\c++\\Modelo PCA
CE 6CP_Lomo1\\varianza.txt";
      const int lonV=strlen(rutavarianza);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *varianza=new char[lonV+1];
   //copio rutavarianza a una nueva area de memoria apuntada por
varianza
      strcpy(varianza,rutavarianza);

      const char *rutaconstantes="C:\\Users\\-\\Desktop\\c++\\Modelo
PCA CE 6CP_Lomo1\\constantes.txt";
      const int lonC=strlen(rutaconstantes);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *constantes=new char[lonC+1];
//copio rutaconstantes a una nueva area de memoria apuntada por
constantes
      strcpy(constantes,rutaconstantes);
```

```
      const char *rutacoeficientes="C:\\Users\\-\\Desktop\\c++\\Modelo
PCA CE 6CP_Lomo1\\coeficientes.txt";
      const int lonCo=strlen(rutacoeficientes);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *coeficientes=new char[lonCo+1];
    //copio rutacoeficientes a una nueva area de memoria apuntada por
coeficientes
      strcpy(coeficientes,rutacoeficientes);

      const char *rutaPCA="C:\\Users\\-\\Desktop\\c++\\Modelo PCA CE
6CP_Lomo1\\PCA.txt";
      const int lonP=strlen(rutaPCA);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *PCA=new char[lonP+1];
//copio rutaPCA a una nueva area de memoria apuntada por PCA
      strcpy(PCA,rutaPCA);


//-------------------------------------------------
//Cargar los archivos necesarios para el algoritmo
//-------------------------------------------------

//Tengo en MediaModelo mis datos,(vector donde se almacenan los datos,
ruta,tamaño de la fila a leer)
      leerArchivo(MediaModelo,media,Filas);
//Libero memoria media,ruta.
      delete media;

//Tengo en VarianzaModelo mis datos,(vecotr donde se almacenan los
datos, ruta,tamaño de la fila a leer)
      leerArchivo(VarianzaModelo,varianza,Filas);
//Libero memoria varianza,ruta.
      delete varianza;

//Tengo en ConstantesModelo mis datos,(vector donde se almacenan los
datos, ruta,tamaño de la fila a leer)
      leerArchivo(ConstantesModelo,constantes,Clases);
      //Libero memoria constantes,ruta.
      delete constantes;

//Tengo en CoeficienteModelo mis datos,(matriz donde se almacenan los
datos, ruta,tamaño de la fila a leer,tamaño de la columna a leer)
      leerMatriz(CoeficientesModelo,coeficientes,Componentes,Clases);
//Libero memoria coeficientes,ruta.
      delete coeficientes;

//Tengo en PCAModelo mis datos,(matriz donde se almacenan los datos,
ruta,tamaño de la fila a leer,tamaño de la columna a leer)
      leerMatriz(PCAModelo,PCA,Filas,Componentes);
//Libero memoria PCA,ruta.
      delete PCA;
```

```
//----------------------------
//Carga de los archivos en GPU
//----------------------------

cublasInit();
if(cublasAlloc (Filas,sizeof(float),(void**)&media_GPU)!=cudaSuccess)
{
      printf("Error copiando datos de CPU a GPU\n");
      return(-1);
}
copiarGPU(MediaModelo,media_GPU,Filas);
delete MediaModelo;

if(cublasAlloc
(Filas,sizeof(float),(void**)&varianza_GPU)!=cudaSuccess)
{
      printf("Error copiando datos de CPU a GPU\n");
      return(-1);
}
copiarGPU(VarianzaModelo,varianza_GPU,Filas);
delete VarianzaModelo;


if(cublasAlloc
(Clases,sizeof(float),(void**)&constantes_GPU)!=cudaSuccess)
{
      printf("Error copiando datos de CPU a GPU\n");
      return(-1);
}
copiarGPU(ConstantesModelo,constantes_GPU,Clases);
delete ConstantesModelo;

if(cublasAlloc
(Componentes*Clases,sizeof(float),(void**)&coeficientes_GPU)!=cudaSucc
ess)
{
      printf("Error copiando datos de CPU a GPU\n");
      return(-1);
}
copiarMatrizGPU(CoeficientesModelo,coeficientes_GPU,Componentes,Clases
);
delete CoeficientesModelo;

if(cublasAlloc
(Filas*Componentes,sizeof(float),(void**)&PCA_GPU)!=cudaSuccess)
{
      printf("Error copiando datos de CPU a GPU\n");
      return(-1);
}
copiarMatrizGPU(PCAModelo,PCA_GPU,Filas,Componentes);
delete PCAModelo;
```

```
//-------------------------------------------------
//Rutas donde se encuentran las IMAGENES a cargar!!
//-------------------------------------------------

      const char *R1="C:\\Users\\-\\Desktop\\c++\\Modelo PCA CE
6CP_Lomo1\\Lomo_Todo_1\\Img_00";
      const int lonR1=strlen(R1);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *ruta1=new char[lonR1+1];
//copio R1 a una nueva area de memoria apuntada por ruta1
      strcpy(ruta1,R1);

      const char *R=".tif";
      const int lonR=strlen(R);
      char *ruta=new char[lonR+1];
//copio R a una nueva area de memoria apuntada por ruta
      strcpy(ruta,R);
//asigno memoria,en buffer formare la ruta completa. Es 1+1 para
guardar el caracter '/0' y el 0 ---->Img_00'0'.tif;
      char *buffer=new char[lonR1+1+1+lonR];

//-------------------------------
//Creo mi cubo de imagenes en CPU
//-------------------------------

//Leo la primera imagen para saber su tamaño y asi crear el cubo de
imagenes, sabiendo que el resto tiene las mismas dimensiones
      sprintf(buffer,"%s%d%s",ruta1,0,ruta);
      int fila_imagen=filaImagen(buffer);
      int columna_imagen=columnaImagen(buffer);
//reservo memoria para la matriz[][][] tridimnesional, diciendole
(matriz3D,Profundidad,fila,columna)
      float *Matriz=new float[Filas*fila_imagen*columna_imagen];

//--------------------------
//Carga del cubo de imagenes
//--------------------------

//Guardo en Matriz todas las imagenes, (matriz3D donde guarda las
imagenes,profundidad,filas de la imagen,columna de la imagen
,buffer donde formo ruta completa,trozo de la ruta primera ruta1, ruta
la segunda parte .tif!!

lectura3D(Matriz,Filas,fila_imagen,columna_imagen,buffer,ruta1,ruta);

//libero memoria de la ruta
      delete ruta;
//libero memoria del buffer
      delete buffer;
//libero memoria de la ruta1
      delete ruta1;

//-----------------------------------------
//Transpaso del cubo de imagenes de CPU a GPU
//-----------------------------------------

if(cublasAlloc
(Filas*fila_imagen*columna_imagen,sizeof(float),(void**)&Cubo_GPU)!=
cudaSuccess) {
      printf("Error copiando datos de CPU a GPU\n");
       return(-1);}
```

```
if(cudaMemcpy(Cubo_GPU,Matriz,Filas*fila_imagen*columna_imagen*b
sizeof(float),cudaMemcpyHostToDevice)!=cudaSuccess)
 {
      printf("Error copiando datos de CPU a GPU\n");
      return(-1);
}

delete Matriz;


//-----------------------------------------------------
//Reserva de memoria de todas mis variables de la GPU
//-----------------------------------------------------

if(cublasAlloc (Filas*columna_imagen*fila_imagen,
sizeof(float),(void**)&Centrado_GPU)!=cudaSuccess)
{
      printf("1Error copiando datos de CPU a GPU\n");
      return(-1);
}

if(cublasAlloc (Componentes*columna_imagen*fila_imagen,
sizeof(float),(void**)&Scores_GPU)!=cudaSuccess)
{
      printf("3Error copiando datos de CPU a GPU\n");
      return(-1);
}

if(cublasAlloc (Clases*columna_imagen*fila_imagen,
sizeof(float),(void**)&Coef_GPU)!=cudaSuccess)
{
      printf("4Error copiando datos de CPU a GPU\n");
      return(-1);
}

if(cublasAlloc (fila_imagen*columna_imagen,
sizeof(float),(void**)&Final_GPU)!=cudaSuccess)
{
      printf("Error copiando datos de CPU a GPU\n");
      return(-1);
}

//----------------------------
//Numero de hilos y de bloques
//----------------------------
// Lanzar el kernel    de centrado y escalado
//Definir el tamaño del grid y de los threads
      dim3 dimBloque(Filas,1);
      dim3 dimGrid(columna_imagen,fila_imagen);

//Definir el tamaño del grid y de los threads del resto de Kernels
      dim3  grid(columna_imagen*fila_imagen/Filas,1);
      dim3  threads(Filas,1);
```

```
//---------------------------------------
//Ejecuto algortimot PCA CE 6CP en la GPU
//---------------------------------------
//-----------------------------
//Centrado y Escalado en la GPU
//-----------------------------

//Ejecuto Centrado y escalado en toda la matriz 3D de imagenes a la
vez!!!
CentradoyEscalado<<<dimGrid,dimBloque>>>(Cubo_GPU,media_GPU,
varianza_GPU,Centrado_GPU);

//-------------------------------------------------
//Multiplicacion con PCA y Coeficienstes en la GPU
//-------------------------------------------------

//Multiplicar por PCA y coeficientes

cublasSgemm('n','t',Componentes,columna_imagen*fila_imagen,Filas,1,PCA
_GPU,Componentes,Centrado_GPU,columna_imagen*fila_imagen,0,Scores_GPU,
Componentes);
cublasSgemm('n','n',Clases,columna_imagen*fila_imagen,Componentes,1,co
eficientes_GPU,Clases,Scores_GPU,Componentes,0,Coef_GPU,Clases);

//Imagen Final
fin<<<grid,threads>>>(Coef_GPU,constantes_GPU,Final_GPU);


//-----------------------------------------------
//Envio a la CPU la imagen final y la represento
//-----------------------------------------------

float *Fin=new float[columna_imagen*fila_imagen];
cublasGetMatrix (887,320,sizeof(float),Final_GPU, 887, Fin, 887);
mostrarImagen(Fin,fila_imagen,columna_imagen);

//-------------------------------------
//Libero todas las variables de la GPU
//-------------------------------------

cublasFree(Cubo_GPU);
cublasFree(media_GPU);
cublasFree(varianza_GPU);
cublasFree(constantes_GPU);
cublasFree(coeficientes_GPU);
cublasFree(PCA_GPU);
cublasFree(Matriz_GPU);
cublasFree(Centrado_GPU);
cublasFree(Scores_GPU);
cublasFree(Coef_GPU);
cublasFree(Imagen_GPU);
cublasFree(Vector_GPU);
cublasFree(Final_GPU);
```

7.5.2 Funciones

```
%-------------------------------------------------------------
% Universidad Politécnica de Valencia
% Escuela Técnica Superior de Ingenieros de Telecomunicación
%-------------------------------------------------------------
% Proyecto Final de Carrera: Aceleración de algoritmos de
% visión hiperespectral mediante GPU
%-------------------------------------------------------------
% Nombre del archivo: Algoritmo PCA CE 6CP
%-------------------------------------------------------------

// Para que el compilador no haga caso de los warnings de los sprintf
y strcpy
#define _CRT_SECURE_NO_WARNINGS

//--------------------------------------
//includes para las distintas librerias
//--------------------------------------
#include <stdio.h>
//libreria de matematicas
#include <math.h>
//operaciones de entrada/salida en el lenguaje de programación C++
#include <iostream>
//manejadores de los archivos de entrada y salida C++
#include <iomanip>
using namespace std;
//libreria para escritura y lectura de archivos
#include <fstream>
//para poder acceder al tiempo del sistema
#include <time.h>


//---------------------------------------
//Declaración de las funciones a utilizar
//---------------------------------------
extern "C"
void Cargaimagenes(const char *ruta,float **imag);
//-------------------------------
//Implementacion de las funciones
//-------------------------------

//leo el archivo seguna la ruta expecificada y lo almaceno en el
vector
extern "C"
void leerArchivo(float *vector, const char *ruta,const int filas)
{
// abrir archivo para lectura

      ifstream fichero;
//Lo abro como entrada
      fichero.open(ruta,ios::in);
// verificar la apertura del archivo
    if ( !fichero ) {
        cerr << "Error al tratar de abrir el archivo"<<endl;
            exit(1);
    }
```

```
    // lectura de datos
      while ( ! fichero.eof()) {
            for(int i=0;i<filas;i++)
//Lee omitiendo blancos, es lo que quiero!!    y guardo en vector
            fichero >> vector[i];
      }
//cierro el descriptor
      fichero.close();
}

//leo un archivo que contiene dentro una matriz y lo almaceno en la
matriz bidimensional
extern "C"
void leerMatriz(float **vector,const char *ruta, const int filas,const
int columnas)
{
// abrir archivo para lectura
      ifstream fichero;
//Lo abro como entrada
      fichero.open(ruta,ios::in);

    // verificar la apertura del archivo
    if ( !fichero ) {
        cerr << "Error al tratar de abrir el archivo"<<endl;
            exit(1);
    }
    // lectura de datos
      while (! fichero.eof()) {
            for(int i =0; i<filas;i++){
                  for(int j=0;j<columnas;j++)
//Lee omitiendo blancos, es lo que quiero!! y guardo en la posicion
adecuada en la matriz
                  fichero >> vector[i][j];
            }
      }
//Cierrro el descriptor
      fichero.close();
}

//Esta función es para mostrar por pantalla el vector introducido
extern "C"
void imprimir(float *vector, int const tamaño)
{
//Me sirve para saber si los archivos introducios son leidos
correctamente
      for(int i=0;i<tamaño;i++)
            cout << vector[i] << "\n";
}

//Esta función es para mostrar por pantalla la matriz introducida
extern "C"
void imprimirMatriz(float **vector, int const fila,const int columna)
{
//Me sirve para saber si los archivos con una matriz dentro han sido
leidos correctamente
      for(int i=0;i<fila;i++){
            for(int j=0;j<columna;j++)
                  cout << vector[i][j] << "  ";
            cout << "\n";
      }
}
```

```
//Esta función es para mostrar por pantalla la matriz introducida
//Me sirve para saber si los archivos con una matriz dentro han sido
leidos correctamente
extern "C"
void imprimirMatriz3D(float *vector,int const profundidad,int const
fila,const int columna)
{
      for(int t=0;t<profundidad;t++){
            for(int i=0;i<fila;i++){
                  for(int j=0;j<columna;j++)
                    cout << vector[fila*columna*t+columna*i+j] << "  ";
                    cout << "\n";
            }
            cout << " <<<<<<<<<< <<< <<<< " << "\n";
      }
}

//Función que me almacena las 256 imagenes en un cubo de imagenes
extern "C"
void lectura3D(float *matriz,int profundidad,int fila,int columna,char
*buffer,char *ruta1,char *ruta)
{
      for (int i=0;i<profundidad;i++){
            //Hago esto porque el nombre de la imagen solo tiene un
cero, por eso los tengo que separar
            if(i==10)
                  ruta1="C:\\Users\\-\\Desktop\\CUDA\\Modelo PCA CE
6CP_Lomo1\\Lomo_Todo_1\\Img_0";
            //Aqui no tiene nigun cero por eso los tengo que separar
            if(i==100)
                  ruta1="C:\\Users\\-\\Desktop\\CUDA\\Modelo PCA CE
6CP_Lomo1\\Lomo_Todo_1\\Img_";

//Creo con sprintf la ruta completa y la almaceno en buffer
            sprintf(buffer,"%s%d%s",ruta1,i,ruta);
//Me creo una variable auxiliar que almacenara una imagen
            float *Imagen=new float [fila*columna];

//Me guarda la imagen de la ruta introducida en la variable imagen y a
continuacion recorro dicha variable para guardarla en mi cubo de
imagenes
            Cargaimagenes(buffer,Imagen);

            for(int j=0;j<fila;j++)
                  for(int t=0;t<columna;t++)
      matriz[fila*columna*i+columna*j+t]=Imagen[j*columna+t];

      //Libero memoria de la variable auxiliar imagen
            delete Imagen;
      }
}
```

7.5.3 Imágenes

```
%-------------------------------------------------------------
% Universidad Politécnica de Valencia
% Escuela Técnica Superior de Ingenieros de Telecomunicación
%-------------------------------------------------------------
% Proyecto Final de Carrera: Aceleración de algoritmos de
% visión hiperespectral mediante GPU
%-------------------------------------------------------------
% Nombre del archivo: Algoritmo PCA CE 6CP
%-------------------------------------------------------------

//-------------------
//Librerias de OpenCV
//-------------------
#include "cv.h"
#include "highgui.h"
//-------------------------------------
//includes para las distintas librerias
//-------------------------------------
#include <stdio.h>
//operaciones de entrada/salida en el lenguaje de programación C++
#include <iostream>
//manejadores de los archivos de entrada y salida C++
#include <iomanip>
using namespace std;
//libreria para escritura y lectura de archivos
#include <fstream>
//libreria para manejo de cadenas
#include <string.h>

//-------------------------------
//Implementacion de las funciones
//-------------------------------

//Carga una imagen de una ruta especificada
extern "C"
void Cargaimagenes(const char *ruta,float **imag)
{
//inicializo imagen,tipo de variable definido en librerias OpenCV
IplImage* imagen=NULL;
// cargamos la imagen,ruta donde se encuentra la imagen,y con -1
indicamos que carga la imagen con los canales que tenga la
imagen(rgb,gris,...)
imagen=cvLoadImage(ruta,-1);

if(imagen==NULL)
      perror( "No se ha podido cargar la imagen correctamente\n");

//definimos la estructura para acceder a los datos de la imagen
//imagen->width=320
//imagen->height=887
CvScalar s;
```

```
	for(int i=0;i<imagen->height;i++){
		for(int j=0;j<imagen->width;j++){
	//cvGet2D definido en las librerias de OpenCV se obtiene el
	valor (i,j) del píxel, que queda almacenado en s.val[0] valor
	double;
			s=cvGet2D(imagen,i,j);
	//Convierto el dato    double en float y lo almaceno en la
	matriz bidimensional dada
			imag[i][j]=(float)s.val[0];
		}
	}
//liberar el espacio de memoria de la imagen
	cvReleaseImage(&imagen);
}

//Funcion que me da el tamaño de la columna de una imagen. Le pasamos
la ruta donde se encuentra dicha imagen
extern "C"
int columnaImagen(char *ruta)
{
IplImage* imagen=NULL;//inicializo imagen
imagen=cvLoadImage(ruta,-1);// cargamos la imagen,
if(imagen==NULL)
	perror( "No se ha podido cargar la imagen correctamente\n");

return(imagen->width); //devuelvo el valor de la columna. IplImage
contiene la clase->width donde se almacena dicho valor tipo int
}

//Funcion que me da el tamaño de la fila de una imagen. Le pasamos la
ruta donde se encuentra dicha imagen
extern "C"
int filaImagen(char *ruta)
{
IplImage* imagen=NULL;//inicializo imagen
imagen=cvLoadImage(ruta,-1);// cargamos la imagen,
if(imagen==NULL)
	perror( "No se ha podido cargar la imagen correctamente\n");

return(imagen->height);	//devuelvo el valor de la fila. IplImage
contiene la clase->height donde se almacena dicho valor tipo int
}

//Funcion que imprime por pantalla la matriz pasada. Los colores de
cada pixel dependera de a que grupo pertenezca
extern "C"
void mostrarImagen(float **Imagen,int fila,int columna)
{

//Reservo memoria para la imagen diciendole el tamaño,valor de los
pixeles, canales que tendra la imagen---> 3 es bgr(rgb)
IplImage* ImagenNueva=cvCreateImage(cvSize(columna,fila),
IPL_DEPTH_8U, 3);

//Para acceder a los pixeles de la imagen nueva
CvScalar s;
//1=verde		Green PET		6=blanco	PE film
//2=amarillo		Transp PET		7=rojo			Bone
//3=marron		HD PE			8=verde oscuro	Fat
//4=gris		Metal			9=naranja	Pork Loin
//5=azul		Insect			10=negro	Background
```

```
	for(int i=0;i<fila;i++){
		for(int j=0;j<columna;j++){
			switch((int)Imagen[i][j])
			{
			case 1:
//s es un vector donde cada componente es blue(0)-green(1)-red(2), y
segun que valores tenga estas componente tendra un color u otro
				s=cvScalar(0,255,0);
				break;
			case 2:
				s=cvScalar(0,255,255);
				break;
			case 3:
				s=cvScalar(0,90,141);
				break;
			case 4:
				s=cvScalar(128,128,128);
				break;
			case 5:
				s=cvScalar(255,0,0);
				break;
			case 6:
				s=cvScalar(255,255,255);
				break;
			case 7:
				s=cvScalar(0,0,255);
				break;
			case 8:
				s=cvScalar(0,103,26);
				break;
			case 9:
				s=cvScalar(0,90,218);
				break;
			case 10:
				s=cvScalar(0,0,0);
				break;
			default:
				s=cvScalar(0,0,0);
				break;
			}
//Asigno el color que corresponda a ese pixel(rgb) que esta en la
variable s
			cvSet2D(ImagenNueva,i,j,s);
		}
	}
//Creo nuna ventana con el titulo lomo1 y con el 1 diremos que se
ajuste a la imagen a dibujar
	cvNamedWindow("Lomo1",1);
//Muestra la imagen por pantalla creada
	cvShowImage("Lomo1",ImagenNueva);
// se pulsa tecla para terminar
	cvWaitKey(0);
// guardamos la imagen en una ruta predefinida
	cvSaveImage("C:\\Users\\-\\Desktop\\c++\\Modelo PCA CE
6CP_Lomo1\\Lomo1_CUDA.jpg",ImagenNueva);
// destruimos todas las ventanas
	cvDestroyAllWindows();
	//liberamos memoria de la imagen
	cvReleaseImage(&ImagenNueva);
}
```

7.5.4 GPU

```
%-------------------------------------------------------------
% Universidad Politécnica de Valencia
% Escuela Técnica Superior de Ingenieros de Telecomunicación
%-------------------------------------------------------------
% Proyecto Final de Carrera: Aceleración de algoritmos de
% visión hiperespectral mediante GPU
%-------------------------------------------------------------
% Nombre del archivo: Algoritmo PCA CE 6CP
%-------------------------------------------------------------


#define _CRT_SECURE_NO_WARNINGS   // Para que el compilador no haga
caso de los warnings de los sprintf y strcpy

//--------------------------------------
//includes para las distintas librerias
//--------------------------------------
#include <stdio.h>
#include <math.h>
//operaciones de entrada/salida en el lenguaje de programación C++
#include <iostream>
/manejadores de los archivos de entrada y salida C++
#include <iomanip>
using namespace std;
//libreria para escritura y lectura de archivos
#include <fstream>
//libreria para manejo de cadenas
#include <string.h>
//para poder acceder al tiempo del sistema
#include <time.h>
#include <cublas.h>
#include <cuda_runtime.h>
#include <cuda.h>

extern "C"
void memoriaGPU(float *vector, int longitud)
{
      cublasStatus_t estado;

      estado=cublasAlloc (longitud,sizeof(float),(void**)&vector);
      if(estado==CUBLAS_STATUS_NOT_INITIALIZED)
            cout << "Libreria no inicializada\n";
      if(estado==CUBLAS_STATUS_INVALID_VALUE)
            cout << "parametros mal\n";
      if(estado==CUBLAS_STATUS_ALLOC_FAILED)
            cout << "falta de recursos\n";
            if(estado==CUBLAS_STATUS_SUCCESS)
            cout << "Todo correcto\n";
}




extern "C"
void copiarGPU(float *origen, float *GPU,int tamaño)
```

```
{
      cublasStatus_t estado;
      estado=cublasSetVector(tamaño,sizeof(float),origen,1,GPU,1);
      if(estado==CUBLAS_STATUS_NOT_INITIALIZED)
            cout << "Libreria no inicializada\n";
      if(estado==CUBLAS_STATUS_INVALID_VALUE)
            cout << "parametros mal\n";
      if(estado==CUBLAS_STATUS_MAPPING_ERROR)
            cout << "error\n";
            if(estado==CUBLAS_STATUS_SUCCESS)
            cout << "Todo correcto\n";
}

extern "C"
void copiarMatrizGPU(float *origen, float *GPU,int fila,int columna)
{
      cublasStatus_t estado;
      estado=cublasSetMatrix(fila,columna,
sizeof(float),origen,fila,GPU,fila);
      if(estado==CUBLAS_STATUS_NOT_INITIALIZED)
            cout << "Libreria no inicializada\n";
      if(estado==CUBLAS_STATUS_INVALID_VALUE)
            cout << "parametros mal\n";
      if(estado==CUBLAS_STATUS_MAPPING_ERROR)
            cout << "error\n";
            if(estado==CUBLAS_STATUS_SUCCESS)
            cout << "Todo correcto\n";
}
```

7.5.5 Kernels

```
%------------------------------------------------------------
% Universidad Politécnica de Valencia
% Escuela Técnica Superior de Ingenieros de Telecomunicación
%------------------------------------------------------------
% Proyecto Final de Carrera: Aceleración de algoritmos de
% visión hiperespectral mediante GPU
%------------------------------------------------------------
% Nombre del archivo: Algoritmo PCA CE 6CP
%------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <math.h>
#include <cuda.h>


__global__ void CentradoyEscalado(float *GPU, float *media,float
*varianza,float *Salida)
{
      int x=threadIdx.x;
      int bx=blockIdx.x;
      int by=blockIdx.y;

Salida[887*320*x+320*by+bx]=(GPU[887*320*x+320*by+bx]-
media[x])*rsqrtf(varianza[x]);
}
```

```
__global__ void fin(float *Coef,float *Cte,float *Final){

	int bx=blockIdx.x*10;
	int pixel=bx*256+threadIdx.x*10;
	float valor;
	int maximo=-100;
	int posicion;

	for(int j=0;j<10;j++)
		{
			valor=Coef[j+pixel]+Cte[j];
			if(maximo < valor)
			{
				maximo=valor;
				posicion=j+1;
			}

	}

	Final[pixel/10]=(float)posicion;
}
```

7.6 Código desarrollado para el Algoritmo PCA 1D15CE 8CP - CUDA

7.6.1 Programa Principal

```
%-------------------------------------------------------------
% Universidad Politécnica de Valencia
% Escuela Técnica Superior de Ingenieros de Telecomunicación
%-------------------------------------------------------------
% Proyecto Final de Carrera: Aceleración de algoritmos de
% visión hiperespectral mediante GPU
%-------------------------------------------------------------
% Nombre del archivo: Algoritmo PCA 1D15CE 8CP
%-------------------------------------------------------------

#define _CRT_SECURE_NO_WARNINGS   // Para que el compilador no haga
caso de los warnings de los sprintf y strcpy

//--------------------------------------
//includes para las distintas librerias
//--------------------------------------
#include <stdio.h>
#include <math.h>
//operaciones de entrada/salida en el lenguaje de programación C++
#include <iostream>
/manejadores de los archivos de entrada y salida C++
#include <iomanip>
using namespace std;
//libreria para escritura y lectura de archivos
#include <fstream>
//libreria para manejo de cadenas
#include <string.h>
//para poder acceder al tiempo del sistema
#include <time.h>
#include <cublas.h>
```

```
#include <cuda_runtime.h>
#include <cuda.h>


// Libreria con los kernels realizados
#include "kernel.cu"

//----------------------------
//Variables Globales definidas
//----------------------------
const int Filas=256;
const int Clases=10;
const int Componentes=6;

//-------------------
//Variables de la GPU
//-------------------

__device__   float* media_GPU;
__device__   float* varianza_GPU;
__device__   float* constantes_GPU;
__device__   float* coeficientes_GPU;
__device__   float* PCA_GPU;
__device__   float* SG_GPU;
__device__   float* Cubo_GPU;
__device__   float* Centrado_GPU;
__device__   float* Coef_GPU;
__device__   float* Matriz_GPU;
__device__   float* Scores_GPU;
__device__   float* Imagen_GPU;
__device__   float* Final_GPU;
__device__   float* Vector_GPU;

//---------------------------------------
//Declaración de las funciones a utilizar
//---------------------------------------
extern "C"
void leerArchivo(float *vector, const char *ruta,const int filas);
extern "C"
void leerMatriz(float *vector,const char *ruta, const int filas,const
int columnas);
extern "C"
void imprimir(float *vector, int const tamaño);
extern "C"
void imprimirMatriz(float *vector, int const fila,const int columna);
extern "C"
void imprimirMatriz3D(float *vector,int const profundidad,int const
fila,const int columna);
extern "C"
void Cargaimagenes(const char *ruta,float *imag);
extern "C"
int columnaImagen(char *ruta);
extern "C"
int filaImagen(char *ruta);
extern "C"
void lectura3D(float *matriz,int profundidad,int fila,int columna,char
*buffer,char *ruta1,char *ruta);
extern "C"
void mostrarImagen(float *Imagen,int fila,int columna);
extern "C"
```

```
void CreacionImagen(float *matriz,float *imagen,float *media,float
*varianza,float *constantes,float *coeficientes,float *PCA,int
filas,int componentes,int clases,int fila_imagen,int columna_imagen);



///GPU funciones
extern "C"
void memoriaGPU(float *vector, int tamaño);
extern "C"
void copiarGPU(float *origen, float *GPU,int tamaño);
extern "C"
void copiarMatrizGPU(float *origen, float *GPU,int fila,int columna);


int main(int argc, char **argv)
{

//---------------------------------------------------------------
//Declaracion de variables donde guardare los datos de los archivos
cargados con su correspondiente reserva de memoria.
//---------------------------------------------------------------

      float *MediaModelo=new float[Filas];
      float *VarianzaModelo=new float[Filas];
      float *ConstantesModelo=new float[Clases];
//reservo memoria para la matriz bidimnesional, diciendole
(matriz,fila,columna)
      float *CoeficientesModelo=new float [Componentes];
//reservo memoria para la matriz[][] bidimnesional, diciendole
(matriz,fila,columna)
      float *PCAModelo=new float *[Filas];
//reservo memoria para la matriz[][] bidimnesional, diciendole
(matriz,fila,columna)
      float *SGModelo=new float [Filas*Filas];


//---------------------------------------------------------------------
//Rutas donde se encuentran los archivos a cargar!! Podria meterlas ya
directamente en la funcion leer
//---------------------------------------------------------------------

      const char *rutamedia="C:\\Users\\-\\Desktop\\c++\\Modelo PCA CE
6CP_Lomo1\\media.txt";
      const int lonM=strlen(rutamedia);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *media=new char[lonM+1];
 //copio rutamedia a una nueva area de memoria apuntada por media
      strcpy(media,rutamedia);

      const char *rutavarianza="C:\\Users\\-\\Desktop\\c++\\Modelo PCA
CE 6CP_Lomo1\\varianza.txt";
      const int lonV=strlen(rutavarianza);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *varianza=new char[lonV+1];
   //copio rutavarianza a una nueva area de memoria apuntada por
varianza
      strcpy(varianza,rutavarianza);

      const char *rutaconstantes="C:\\Users\\-\\Desktop\\c++\\Modelo
PCA CE 6CP_Lomo1\\constantes.txt";
```

```
      const int lonC=strlen(rutaconstantes);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *constantes=new char[lonC+1];
//copio rutaconstantes a una nueva area de memoria apuntada por
constantes
      strcpy(constantes,rutaconstantes);

      const char *rutacoeficientes="C:\\Users\\-\\Desktop\\c++\\Modelo
PCA CE 6CP_Lomo1\\coeficientes.txt";
      const int lonCo=strlen(rutacoeficientes);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *coeficientes=new char[lonCo+1];
    //copio rutacoeficientes a una nueva area de memoria apuntada por
coeficientes
      strcpy(coeficientes,rutacoeficientes);

      const char *rutaPCA="C:\\Users\\-\\Desktop\\c++\\Modelo PCA CE
6CP_Lomo1\\PCA.txt";
      const int lonP=strlen(rutaPCA);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *PCA=new char[lonP+1];
//copio rutaPCA a una nueva area de memoria apuntada por PCA
      strcpy(PCA,rutaPCA);


      const char *rutaSG="C:\\Users\\-\\Desktop\\CUDA\\Modelo PCA
1D15CE 8CP_Lomo1OK\\Mat_SG_1D51.txt";
      const int lonSG=strlen(rutaSG);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *SG=new char[lonSG+1];
//copio rutaSG a una nueva area de memoria apuntada por SG
      strcpy(SG,rutaSG);


//-------------------------------------------------
//Cargar los archivos necesarios para el algoritmo
//-------------------------------------------------

//Tengo en MediaModelo mis datos,(vector donde se almacenan los datos,
ruta,tamaño de la fila a leer)
      leerArchivo(MediaModelo,media,Filas);
//Libero memoria media,ruta.
      delete media;

//Tengo en VarianzaModelo mis datos,(vecotr donde se almacenan los
datos, ruta,tamaño de la fila a leer)
      leerArchivo(VarianzaModelo,varianza,Filas);
//Libero memoria varianza,ruta.
      delete varianza;

//Tengo en ConstantesModelo mis datos,(vector donde se almacenan los
datos, ruta,tamaño de la fila a leer)
      leerArchivo(ConstantesModelo,constantes,Clases);
      //Libero memoria constantes,ruta.
      delete constantes;

//Tengo en CoeficienteModelo mis datos,(matriz donde se almacenan los
datos, ruta,tamaño de la fila a leer,tamaño de la columna a leer)
      leerMatriz(CoeficientesModelo,coeficientes,Componentes,Clases);
//Libero memoria coeficientes,ruta.
      delete coeficientes;
```

```
//Tengo en PCAModelo mis datos,(matriz donde se almacenan los datos,
ruta,tamaño de la fila a leer,tamaño de la columna a leer)
      leerMatriz(PCAModelo,PCA,Filas,Componentes);
//Libero memoria PCA,ruta.
      delete PCA;

//Tengo en PCAModelo mis datos,(matriz donde se almacenan los datos,
ruta,tamaño de la fila a leer,tamaño de la columna a leer)
      leerMatriz(SGModelo,SG,Filas,Filas);
//Libero memoria PCA,ruta.
      delete SG;

//----------------------------
//Carga de los archivos en GPU
//----------------------------

cublasInit();
if(cublasAlloc (Filas,sizeof(float),(void**)&media_GPU)!=cudaSuccess)
{
      printf("Error copiando datos de CPU a GPU\n");
      return(-1);
}
copiarGPU(MediaModelo,media_GPU,Filas);
delete MediaModelo;

if(cublasAlloc
(Filas,sizeof(float),(void**)&varianza_GPU)!=cudaSuccess)
{
      printf("Error copiando datos de CPU a GPU\n");
      return(-1);
}
copiarGPU(VarianzaModelo,varianza_GPU,Filas);
delete VarianzaModelo;


if(cublasAlloc
(Clases,sizeof(float),(void**)&constantes_GPU)!=cudaSuccess)
{
      printf("Error copiando datos de CPU a GPU\n");
      return(-1);
}
copiarGPU(ConstantesModelo,constantes_GPU,Clases);
delete ConstantesModelo;

if(cublasAlloc
(Componentes*Clases,sizeof(float),(void**)&coeficientes_GPU)!=cudaSucc
ess)
{
      printf("Error copiando datos de CPU a GPU\n");
      return(-1);
}
copiarMatrizGPU(CoeficientesModelo,coeficientes_GPU,Componentes,Clases
);
delete CoeficientesModelo;

if(cublasAlloc
(Filas*Componentes,sizeof(float),(void**)&PCA_GPU)!=cudaSuccess)
{
      printf("Error copiando datos de CPU a GPU\n");
      return(-1);
}
```

```
copiarMatrizGPU(PCAModelo,PCA_GPU,Filas,Componentes);
delete PCAModelo;

if(cublasAlloc (Filas*Filas,
sizeof(float),(void**)&SG_GPU)!=cudaSuccess)
{
      printf("Error copiando datos de CPU a GPU\n");
       return(-1);
}
copiarMatrizGPU(SGModelo,SG_GPU,Filas,Filas);
delete SGModelo;


//-------------------------------------------------
//Rutas donde se encuentran las IMAGENES a cargar!!
//-------------------------------------------------

      const char *R1="C:\\Users\\-\\Desktop\\c++\\Modelo PCA CE
6CP_Lomo1\\Lomo_Todo_1\\Img_00";
      const int lonR1=strlen(R1);
//asigno memoria y es mas 1 para guardar el caracter '/0'
      char *ruta1=new char[lonR1+1];
//copio R1 a una nueva area de memoria apuntada por ruta1
      strcpy(ruta1,R1);

      const char *R=".tif";
      const int lonR=strlen(R);
      char *ruta=new char[lonR+1];
//copio R a una nueva area de memoria apuntada por ruta
      strcpy(ruta,R);
//asigno memoria,en buffer formare la ruta completa. Es 1+1 para
guardar el caracter '/0' y el 0 ---->Img_00'0'.tif;
      char *buffer=new char[lonR1+1+1+lonR];

//--------------------------------
//Creo mi cubo de imagenes en CPU
//--------------------------------

//Leo la primera imagen para saber su tamaño y asi crear el cubo de
imagenes, sabiendo que el resto tiene las mismas dimensiones
      sprintf(buffer,"%s%d%s",ruta1,0,ruta);
      int fila_imagen=filaImagen(buffer);
      int columna_imagen=columnaImagen(buffer);
//reservo memoria para la matriz[][][] tridimnesional, diciendole
(matriz3D,Profundidad,fila,columna)
      float *Matriz=new float[Filas*fila_imagen*columna_imagen];

//--------------------------
//Carga del cubo de imagenes
//--------------------------

//Guardo en Matriz todas las imagenes, (matriz3D donde guarda las
imagenes,profundidad,filas de la imagen,columna de la imagen
,buffer donde formo ruta completa,trozo de la ruta primera ruta1, ruta
la segunda parte .tif!!

lectura3D(Matriz,Filas,fila_imagen,columna_imagen,buffer,ruta1,ruta);

//libero memoria de la ruta
      delete ruta;
//libero memoria del buffer
      delete buffer;
```

```
//libero memoria de la ruta1
      delete ruta1;

//-----------------------------------------
//Transpaso del cubo de imagenes de CPU a GPU
//-----------------------------------------

if(cublasAlloc
(Filas*fila_imagen*columna_imagen,sizeof(float),(void**)&Cubo_GPU)!=
cudaSuccess) {
      printf("Error copiando datos de CPU a GPU\n");
       return(-1);}
if(cudaMemcpy(Cubo_GPU,Matriz,Filas*fila_imagen*columna_imagen*b
sizeof(float),cudaMemcpyHostToDevice)!=cudaSuccess)
 {
      printf("Error copiando datos de CPU a GPU\n");
      return(-1);
}

delete Matriz;


//-----------------------------------------------------
//Reserva de memoria de todas mis variables de la GPU
//-----------------------------------------------------

if(cublasAlloc (Filas*columna_imagen*fila_imagen,
sizeof(float),(void**)&Centrado_GPU)!=cudaSuccess)
{
      printf("1Error copiando datos de CPU a GPU\n");
      return(-1);
}

if(cublasAlloc (Componentes*columna_imagen*fila_imagen,
sizeof(float),(void**)&Scores_GPU)!=cudaSuccess)
{
      printf("3Error copiando datos de CPU a GPU\n");
      return(-1);
}

if(cublasAlloc (Clases*columna_imagen*fila_imagen,
sizeof(float),(void**)&Coef_GPU)!=cudaSuccess)
{
      printf("4Error copiando datos de CPU a GPU\n");
      return(-1);
}

if(cublasAlloc (fila_imagen*columna_imagen,
sizeof(float),(void**)&Final_GPU)!=cudaSuccess)
{
      printf("Error copiando datos de CPU a GPU\n");
      return(-1);
}

if(cublasAlloc (Filas*columna_imagen*fila_imagen,
sizeof(float),(void**)&Matriz_GPU)!=cudaSuccess)
{
      printf("1Error copiando datos de CPU a GPU\n");
      return(-1);
}
//----------------------------
//Numero de hilos y de bloques
```

```
//----------------------------
//Numero de hilos y de bloques
//----------------------------

// Lanzar el kernel     de centrado y escalado
//Definir el tamaño del grid y de los threads
      dim3 dimBloque(Filas,1);
      dim3 dimGrid(columna_imagen,fila_imagen);

//Definir el tamaño del grid y de los threads del resto de Kernels
      dim3  grid(columna_imagen*fila_imagen/Filas,1);
      dim3  threads(Filas,1);


//-----------------------------------------
//Ejecuto algortimot PCA 1D15CE 8CP en la GPU
//-----------------------------------------

//---------------------------------
//Derivada Saviztky-Golay en la GPU
//---------------------------------

cublasSgemm('n','t', columna_imagen*fila_imagen,Filas,Filas,1,
Cubo_GPU, columna_imagen*fila_imagen,SG_GPU,Filas,0,Matriz_GPU,
columna_imagen*fila_imagen);

//-----------------------------
//Centrado y Escalado en la GPU
//-----------------------------

//Ejecuto Centrado y escalado en toda la matriz 3D de imagenes a la
vez!!!
CentradoyEscalado<<<dimGrid,dimBloque>>>(Matriz_GPU,media_GPU,
varianza_GPU,Centrado_GPU);

//-------------------------------------------------
//Multiplicacion con PCA y Coeficienstes en la GPU
//-------------------------------------------------

//Multiplicar por PCA y coeficientes

cublasSgemm('n','t',Componentes,columna_imagen*fila_imagen,Filas,1,PCA
_GPU,Componentes,Centrado_GPU,columna_imagen*fila_imagen,0,Scores_GPU,
Componentes);
cublasSgemm('n','n',Clases,columna_imagen*fila_imagen,Componentes,1,co
eficientes_GPU,Clases,Scores_GPU,Componentes,0,Coef_GPU,Clases);

//Imagen Final
fin<<<grid,threads>>>(Coef_GPU,constantes_GPU,Final_GPU);


//-----------------------------------------------
//Envio a la CPU la imagen final y la represento
//-----------------------------------------------

float *Fin=new float[columna_imagen*fila_imagen];
cublasGetMatrix (887,320,sizeof(float),Final_GPU, 887, Fin, 887);
mostrarImagen(Fin,fila_imagen,columna_imagen);
```

```
//-------------------------------------
//Libero todas las variables de la GPU
//-------------------------------------

cublasFree(Cubo_GPU);
cublasFree(media_GPU);
cublasFree(varianza_GPU);
cublasFree(constantes_GPU);
cublasFree(coeficientes_GPU);
cublasFree(PCA_GPU);
cublasFree(Matriz_GPU);
cublasFree(Centrado_GPU);
cublasFree(Scores_GPU);
cublasFree(Coef_GPU);
cublasFree(Imagen_GPU);
cublasFree(Vector_GPU);
cublasFree(Final_GPU);
```

7.6.2 Funciones

```
%-------------------------------------------------------------
% Universidad Politécnica de Valencia
% Escuela Técnica Superior de Ingenieros de Telecomunicación
%-------------------------------------------------------------
% Proyecto Final de Carrera: Aceleración de algoritmos de
% visión hiperespectral mediante GPU
%-------------------------------------------------------------
% Nombre del archivo: Algoritmo PCA 1D15CE 8CP
%-------------------------------------------------------------

// Para que el compilador no haga caso de los warnings de los sprintf
y strcpy
#define _CRT_SECURE_NO_WARNINGS

//-------------------------------------
//includes para las distintas librerias
//-------------------------------------
#include <stdio.h>
//libreria de matematicas
#include <math.h>
//operaciones de entrada/salida en el lenguaje de programación C++
#include <iostream>
//manejadores de los archivos de entrada y salida C++
#include <iomanip>
using namespace std;
//libreria para escritura y lectura de archivos
#include <fstream>
//para poder acceder al tiempo del sistema
#include <time.h>


//---------------------------------------
//Declaración de las funciones a utilizar
//---------------------------------------
extern "C"
void Cargaimagenes(const char *ruta,float **imag);
```

```
//-------------------------------
//Implementacion de las funciones
//-------------------------------

//leo el archivo seguna la ruta expecificada y lo almaceno en el
vector
extern "C"
void leerArchivo(float *vector, const char *ruta,const int filas)
{
// abrir archivo para lectura

      ifstream fichero;
//Lo abro como entrada
      fichero.open(ruta,ios::in);
// verificar la apertura del archivo
    if ( !fichero ) {
        cerr << "Error al tratar de abrir el archivo"<<endl;
            exit(1);
    }


    // lectura de datos
      while ( ! fichero.eof()) {
            for(int i=0;i<filas;i++)
//Lee omitiendo blancos, es lo que quiero!!    y guardo en vector
            fichero >> vector[i];
      }
//cierro el descriptor
      fichero.close();
}

//leo un archivo que contiene dentro una matriz y lo almaceno en la
matriz bidimensional
extern "C"
void leerMatriz(float **vector,const char *ruta, const int filas,const
int columnas)
{
// abrir archivo para lectura
      ifstream fichero;
//Lo abro como entrada
      fichero.open(ruta,ios::in);

    // verificar la apertura del archivo
    if ( !fichero ) {
        cerr << "Error al tratar de abrir el archivo"<<endl;
            exit(1);
    }
    // lectura de datos
      while (! fichero.eof()) {
            for(int i =0; i<filas;i++){
                  for(int j=0;j<columnas;j++)
//Lee omitiendo blancos, es lo que quiero!! y guardo en la posicion
adecuada en la matriz
                  fichero >> vector[i][j];
            }
      }
//Cierrro el descriptor
      fichero.close();
}

//Esta función es para mostrar por pantalla el vector introducido
```

```
extern "C"
void imprimir(float *vector, int const tamaño)
{
//Me sirve para saber si los archivos introducios son leidos
correctamente
      for(int i=0;i<tamaño;i++)
            cout << vector[i] << "\n";
}

//Esta función es para mostrar por pantalla la matriz introducida
extern "C"
void imprimirMatriz(float **vector, int const fila,const int columna)
{
//Me sirve para saber si los archivos con una matriz dentro han sido
leidos correctamente
      for(int i=0;i<fila;i++){
            for(int j=0;j<columna;j++)
                  cout << vector[i][j] << "  ";
            cout << "\n";
      }
}
//Esta función es para mostrar por pantalla la matriz introducida
//Me sirve para saber si los archivos con una matriz dentro han sido
leidos correctamente
extern "C"
void imprimirMatriz3D(float *vector,int const profundidad,int const
fila,const int columna)
{
      for(int t=0;t<profundidad;t++){
            for(int i=0;i<fila;i++){
                  for(int j=0;j<columna;j++)
                    cout << vector[fila*columna*t+columna*i+j] << "  ";
                    cout << "\n";
            }
            cout << " <<<<<<<<<< <<< <<<< " << "\n";
      }
}

//Función que me almacena las 256 imagenes en un cubo de imagenes
extern "C"
void lectura3D(float *matriz,int profundidad,int fila,int columna,char
*buffer,char *ruta1,char *ruta)
{
      for (int i=0;i<profundidad;i++){
            //Hago esto porque el nombre de la imagen solo tiene un
cero, por eso los tengo que separar
            if(i==10)
                  ruta1="C:\\Users\\-\\Desktop\\CUDA\\Modelo PCA 1D15CE
8CP_Lomo1\\Lomo_Todo_1\\Img_0";
            //Aqui no tiene nigun cero por eso los tengo que separar
            if(i==100)
                  ruta1="C:\\Users\\-\\Desktop\\CUDA\\Modelo PCA 1D15CE
8CP_Lomo1\\Lomo_Todo_1\\Img_";

//Creo con sprintf la ruta completa y la almaceno en buffer
            sprintf(buffer,"%s%d%s",ruta1,i,ruta);
//Me creo una variable auxiliar que almacenara una imagen
            float *Imagen=new float [fila*columna];
```

```
//Me guarda la imagen de la ruta introducida en la variable imagen y a
continuacion recorro dicha variable para guardarla en mi cubo de
imagenes
            Cargaimagenes(buffer,Imagen);

            for(int j=0;j<fila;j++)
                  for(int t=0;t<columna;t++)
      matriz[fila*columna*i+columna*j+t]=Imagen[j*columna+t];

      //Libero memoria de la variable auxiliar imagen
            delete Imagen;
      }
}
```

7.6.3 Imágenes

```
%-------------------------------------------------------------
% Universidad Politécnica de Valencia
% Escuela Técnica Superior de Ingenieros de Telecomunicación
%-------------------------------------------------------------
% Proyecto Final de Carrera: Aceleración de algoritmos de
% visión hiperespectral mediante GPU
%-------------------------------------------------------------
% Nombre del archivo: Algoritmo PCA 1D15CE 8CP
%-------------------------------------------------------------

//-------------------
//Librerias de OpenCV
//-------------------
#include "cv.h"
#include "highgui.h"
//-------------------------------------
//includes para las distintas librerias
//-------------------------------------
#include <stdio.h>
//operaciones de entrada/salida en el lenguaje de programación C++
#include <iostream>
//manejadores de los archivos de entrada y salida C++
#include <iomanip>
using namespace std;
//libreria para escritura y lectura de archivos
#include <fstream>
//libreria para manejo de cadenas
#include <string.h>

//-------------------------------
//Implementacion de las funciones
//-------------------------------

//Carga una imagen de una ruta especificada
extern "C"
void Cargaimagenes(const char *ruta,float **imag)
{
//inicializo imagen,tipo de variable definido en librerias OpenCV
IplImage* imagen=NULL;
// cargamos la imagen,ruta donde se encuentra la imagen,y con -1
indicamos que carga la imagen con los canales que tenga la
imagen(rgb,gris,...)
imagen=cvLoadImage(ruta,-1);
```

```
if(imagen==NULL)
      perror( "No se ha podido cargar la imagen correctamente\n");

//definimos la estructura para acceder a los datos de la imagen
//imagen->width=320
//imagen->height=887
CvScalar s;

      for(int i=0;i<imagen->height;i++){
            for(int j=0;j<imagen->width;j++){
      //cvGet2D definido en las librerias de OpenCV se obtiene el
      valor (i,j) del píxel, que queda almacenado en s.val[0] valor
      double;
                  s=cvGet2D(imagen,i,j);
      //Convierto el dato     double en float y lo almaceno en la
      matriz bidimensional dada
                  imag[i][j]=(float)s.val[0];
            }
      }
//liberar el espacio de memoria de la imagen
      cvReleaseImage(&imagen);
}

//Funcion que me da el tamaño de la columna de una imagen. Le pasamos
la ruta donde se encuentra dicha imagen
extern "C"
int columnaImagen(char *ruta)
{
IplImage* imagen=NULL;//inicializo imagen
imagen=cvLoadImage(ruta,-1);// cargamos la imagen,
if(imagen==NULL)
      perror( "No se ha podido cargar la imagen correctamente\n");

return(imagen->width); //devuelvo el valor de la columna. IplImage
contiene la clase->width donde se almacena dicho valor tipo int
}

//Funcion que me da el tamaño de la fila de una imagen. Le pasamos la
ruta donde se encuentra dicha imagen
extern "C"
int filaImagen(char *ruta)
{
IplImage* imagen=NULL;//inicializo imagen
imagen=cvLoadImage(ruta,-1);// cargamos la imagen,
if(imagen==NULL)
      perror( "No se ha podido cargar la imagen correctamente\n");

return(imagen->height);       //devuelvo el valor de la fila. IplImage
contiene la clase->height donde se almacena dicho valor tipo int
}

//Funcion que imprime por pantalla la matriz pasada. Los colores de
cada pixel dependera de a que grupo pertenezca
extern "C"
void mostrarImagen(float **Imagen,int fila,int columna)
{

//Reservo memoria para la imagen diciendole el tamaño,valor de los
pixeles, canales que tendra la imagen---> 3 es bgr(rgb)
IplImage* ImagenNueva=cvCreateImage(cvSize(columna,fila),
IPL_DEPTH_8U, 3);
```

```
//Para acceder a los pixeles de la imagen nueva
CvScalar s;
//1=verde               Green PET          6=blanco       PE film
//2=amarillo            Transp PET         7=rojo               Bone
//3=marron              HD PE              8=verde oscuro  Fat
//4=gris                Metal              9=naranja       Pork Loin
//5=azul                Insect             10=negro        Background

      for(int i=0;i<fila;i++){
            for(int j=0;j<columna;j++){
                  switch((int)Imagen[i][j])
                  {
                  case 1:
//s es un vector donde cada componente es blue(0)-green(1)-red(2), y
segun que valores tenga estas componente tendra un color u otro
                        s=cvScalar(0,255,0);
                        break;
                  case 2:
                        s=cvScalar(0,255,255);
                        break;
                  case 3:
                        s=cvScalar(0,90,141);
                        break;
                  case 4:
                        s=cvScalar(128,128,128);
                        break;
                  case 5:
                        s=cvScalar(255,0,0);
                        break;
                  case 6:
                        s=cvScalar(255,255,255);
                        break;
                  case 7:
                        s=cvScalar(0,0,255);
                        break;
                  case 8:
                        s=cvScalar(0,103,26);
                        break;
                  case 9:
                        s=cvScalar(0,90,218);
                        break;
                  case 10:
                        s=cvScalar(0,0,0);
                        break;
                  default:
                        s=cvScalar(0,0,0);
                        break;
                  }
//Asigno el color que corresponda a ese pixel(rgb) que esta en la
variable s
                  cvSet2D(ImagenNueva,i,j,s);
            }
      }
//Creo nuna ventana con el titulo lomo1 y con el 1 diremos que se
ajuste a la imagen a dibujar
      cvNamedWindow("Lomo1",1);
//Muestra la imagen por pantalla creada
      cvShowImage("Lomo1",ImagenNueva);
// se pulsa tecla para terminar
      cvWaitKey(0);
```

```
// guardamos la imagen en una ruta predefinida
      cvSaveImage("C:\\Users\\-\\Desktop\\c++\\Modelo PCA 1D15CE
8CP_Lomo1\\Lomo1SG_CUDA.jpg",ImagenNueva);
// destruimos todas las ventanas
      cvDestroyAllWindows();
      //liberamos memoria de la imagen
      cvReleaseImage(&ImagenNueva);
}
```

7.6.4 GPU

```
%-------------------------------------------------------------
% Universidad Politécnica de Valencia
% Escuela Técnica Superior de Ingenieros de Telecomunicación
%-------------------------------------------------------------
% Proyecto Final de Carrera: Aceleración de algoritmos de
% visión hiperespectral mediante GPU
%-------------------------------------------------------------
% Nombre del archivo: Algoritmo PCA 1D15CE 8CP
%-------------------------------------------------------------


#define _CRT_SECURE_NO_WARNINGS   // Para que el compilador no haga
caso de los warnings de los sprintf y strcpy

//-------------------------------------
//includes para las distintas librerias
//-------------------------------------
#include <stdio.h>
#include <math.h>
//operaciones de entrada/salida en el lenguaje de programación C++
#include <iostream>
/manejadores de los archivos de entrada y salida C++
#include <iomanip>
using namespace std;
//libreria para escritura y lectura de archivos
#include <fstream>
//libreria para manejo de cadenas
#include <string.h>
//para poder acceder al tiempo del sistema
#include <time.h>
#include <cublas.h>
#include <cuda_runtime.h>
#include <cuda.h>

extern "C"
void memoriaGPU(float *vector, int longitud)
{
      cublasStatus_t estado;

      estado=cublasAlloc (longitud,sizeof(float),(void**)&vector);
      if(estado==CUBLAS_STATUS_NOT_INITIALIZED)
            cout << "Libreria no inicializada\n";
      if(estado==CUBLAS_STATUS_INVALID_VALUE)
            cout << "parametros mal\n";
      if(estado==CUBLAS_STATUS_ALLOC_FAILED)
            cout << "falta de recursos\n";
            if(estado==CUBLAS_STATUS_SUCCESS)
            cout << "Todo correcto\n";
}
```

```
extern "C"
void copiarGPU(float *origen, float *GPU,int tamaño)
{
      cublasStatus_t estado;
      estado=cublasSetVector(tamaño,sizeof(float),origen,1,GPU,1);
      if(estado==CUBLAS_STATUS_NOT_INITIALIZED)
            cout << "Libreria no inicializada\n";
      if(estado==CUBLAS_STATUS_INVALID_VALUE)
            cout << "parametros mal\n";
      if(estado==CUBLAS_STATUS_MAPPING_ERROR)
            cout << "error\n";
            if(estado==CUBLAS_STATUS_SUCCESS)
            cout << "Todo correcto\n";
}

extern "C"
void copiarMatrizGPU(float *origen, float *GPU,int fila,int columna)
{
      cublasStatus_t estado;
      estado=cublasSetMatrix(fila,columna,
sizeof(float),origen,fila,GPU,fila);
      if(estado==CUBLAS_STATUS_NOT_INITIALIZED)
            cout << "Libreria no inicializada\n";
      if(estado==CUBLAS_STATUS_INVALID_VALUE)
            cout << "parametros mal\n";
      if(estado==CUBLAS_STATUS_MAPPING_ERROR)
            cout << "error\n";
            if(estado==CUBLAS_STATUS_SUCCESS)
            cout << "Todo correcto\n";
}
```

7.6.5 Kernels

```
%-------------------------------------------------------------
% Universidad Politécnica de Valencia
% Escuela Técnica Superior de Ingenieros de Telecomunicación
%-------------------------------------------------------------
% Proyecto Final de Carrera: Aceleración de algoritmos de
% visión hiperespectral mediante GPU
%-------------------------------------------------------------
% Nombre del archivo: Algoritmo PCA 1D15CE 8CP
%-------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <math.h>
#include <cuda.h>


__global__ void CentradoyEscalado(float *GPU, float *media,float
*varianza,float *Salida)
{
      int x=threadIdx.x;
      int bx=blockIdx.x;
      int by=blockIdx.y;

Salida[887*320*x+320*by+bx]=(GPU[887*320*x+320*by+bx]-
media[x])*rsqrtf(varianza[x]);
}
```

```
__global__ void fin(float *Coef,float *Cte,float *Final){

      int bx=blockIdx.x*10;
      int pixel=bx*256+threadIdx.x*10;
      float valor;
      int maximo=-100;
      int posicion;

      for(int j=0;j<10;j++)
            {
                  valor=Coef[j+pixel]+Cte[j];
                  if(maximo < valor)
                  {
                        maximo=valor;
                        posicion=j+1;
                  }

      }

      Final[pixel/10]=(float)posicion;
}
```

7.7 Instalación del kit de diseño de Cuda

Antes de instalar nada se tendrá que estar seguro de que la GPU es Nvidia y que además soporta CUDA. Para ello, se verificará el modelo de la GPU. Esto se puede hacer clicando con el botón derecho en el escritorio y yendo al panel de control de Nvidia. A continuación, se irá a Ayuda ->Información del Sistema y ahí se encontrará el modelo de la GPU en el caso que sea Nvidia. Si no se puede realizar estas acciones es que no se va a poder trabajar con el kit de desarrollo CUDA.

Una vez sabido nuestro modelo de GPU se irá a la siguiente página, http://developer.nvidia.com/cuda-gpus, para comprobar si es compatible con CUDA.

Tras comprobar todos los requisitos anteriores solo hay que descargarse de la página oficial de Nvidia, http://developer.nvidia.com/cuda-toolkit-41, las siguientes herramientas e instalarlas:

- **CUDA Toolkit Downloads:** C/C++ compiler, CUDA-GDB, Visual Profiler, CUDA Memcheck, GPU-accelerated libraries, Other tools & Documentation.

- **Developer Drivers Downloads:** Drivers actualizado para la GPU.

- **GPU Computing SDK Downloads:** Códigos de ejemplos.

7.8 Instalación de Visual C++ 2008

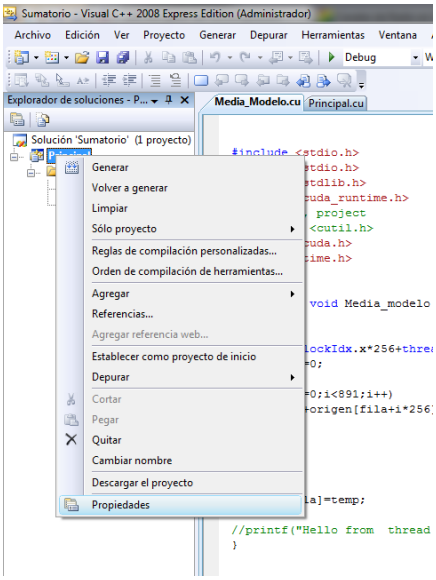
Se descargará Visual C++ 2008 de la siguiente página http://www.microsoft.com/downloads/es-es/details.aspx?FamilyID=9b2da534-3e03-4391-8a4d-074b9f2bc1bf. Una vez descargado se instalará. Después de instalarlo se tendrá que registrar el programa. Este último paso es muy intuitivo y siguiendo las indicaciones del programa de instalación no hay ningún inconveniente.

7.9 Configuración de Visual C++ 2008 para utilizar CUDA

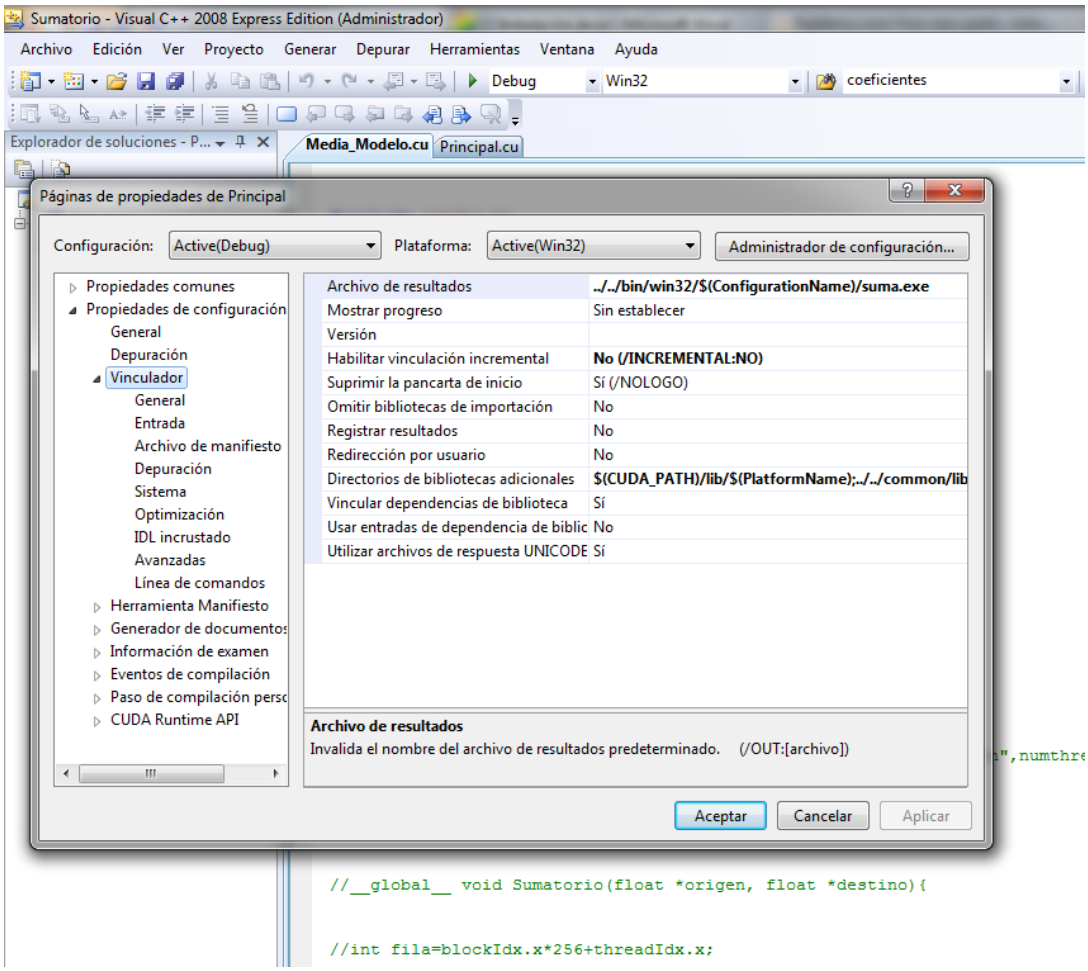
Hay dos posibilidades, una fácil y otra algo más complicada.

La primera es coger uno de los ejemplos que hay en el kit de herramientas SDK, C:\ProgramData\NVIDIA Corporation\NVIDIA GPU Computing SDK 4.1\C\src, y cambiarles los nombres de los archivos y trabajar sobre ellos.

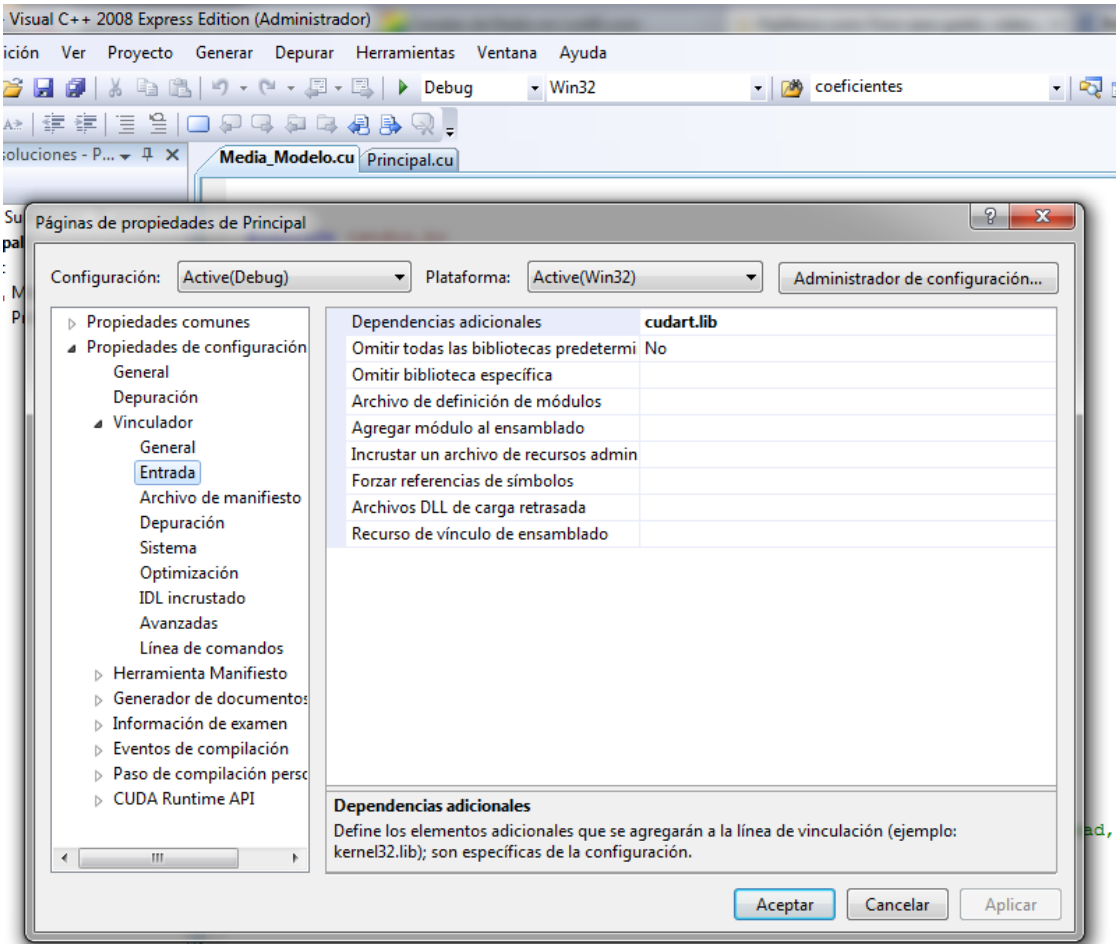
La segunda se trata de configurar desde el principio el proyecto de Visual C++. Se tendrá que abrir un proyecto vacío y del tipo de una aplicación de consola Win32. Una vez abierto se tendrá que clicar sobre el nombre del proyecto con el botón derecho e ir a propiedades.



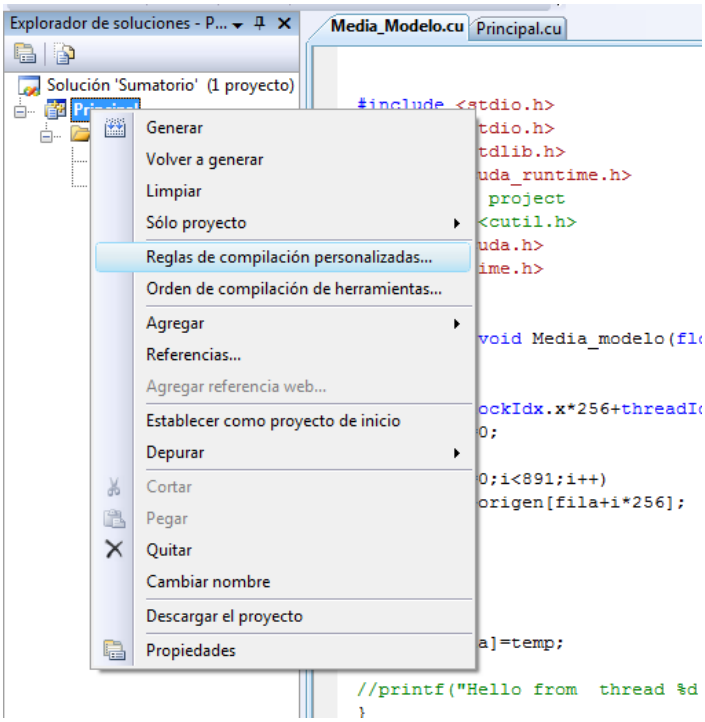
Una vez dentro de propiedades se va a Propiedades de configuración y a su vez a Vinculador. Una vez dentro del vinculador se clica en la casilla de Directorios de bibliotecas adicionales y ahí se introduces el path donde se encuentra las librerías de CUDA.



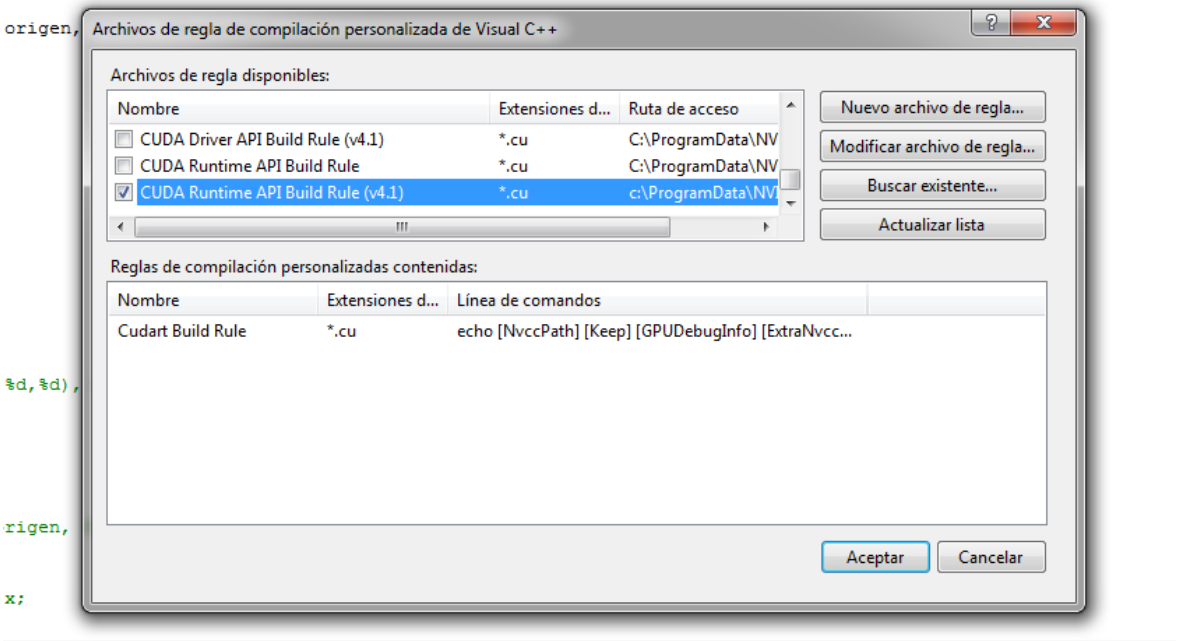
Luego dentro del apartado vinculador se selecciona entrada y en la casilla dependencias adicionales se escribi cudart.lib como muestra la imagen.



Por último lo que queda es compilar con reglas CUDA y esto se hace dando clic en nombre del proyecto con el botón derecho y yendo al apartado que pone regla de compilación personalizadas .



Una vez aquí, se elegí la regla CUDA Runtime API Build Rule como se muestra en la imagen y ya se tendrá el proyecto listo para utilizar CUDA.

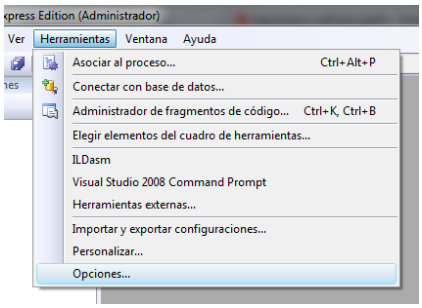


7.10 Configuración de Visual C++ 2008 para utilizar OpenCV

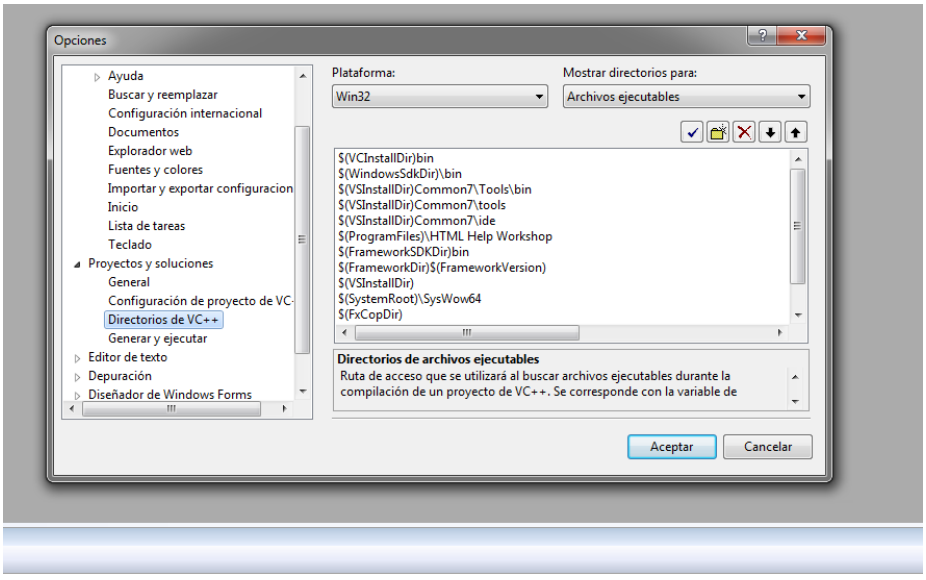
Las siglas Opencv provienen de los términos anglosajones “Open Source Computer Vision Library”. Por lo tanto, Opencv es una librería de tratamiento de imágenes, destinada principalmente a aplicaciones de visión por computador en tiempo real.

En primer lugar se bajará la librería de la siguiente página y se procederá a su instalación en una carpeta sin espacios , como por ejemplo, C: \ OpenCV2.1 \. Durante la instalación se habilitará la opción "Añadir OpenCV en el PATH del sistema para todos los usuarios. http://sourceforge.net/projects/opencvlibrary/files/opencv-win/2.1/

Ahora se configurará el Visual C++ para poder utilizar dicha librería. Una vez. abierto el Visual C++ se va a Herramientas > Opciones.



En la ventana que se abrirá, se clicará: Proyectos y Soluciones > Directorios de VC++.



Una vez llegados a este punto, se tendrá que añadir las correspondientes rutas de las librerías en los directorios que toca. En la siguiente tabla se ilustra las relaciones de estas rutas a añadir en los respectivos directorios.

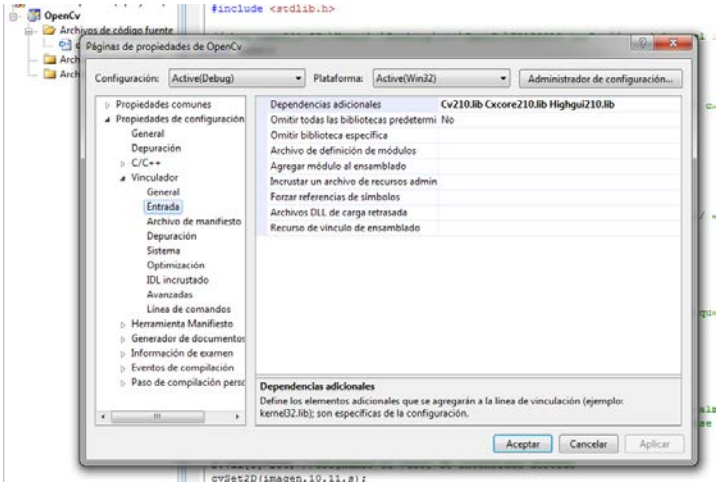
Mostrar Directorio para	*Archivos de Inclusión*	*Archivos de Biblioteca*	*Archivos de Código Fuente*
Ruta a Añadir	C:\OpenCV2.1\include\opencv	C:\OpenCV2.1\lib	C:\OpenCV2.1\src\cv C:\OpenCV2.1\src\cvaux C:\OpenCV2.1\src\cxcore C:\OpenCV2.1\src\highgui

Después de todo esto, se creará un proyecto en el que se tendrá que añadir las dependencias OpenCV. Esto se hace de la siguiente manera:

Abrir las propiedades del proyecto e ir a Propiedades de configuración < Vinculador < Entrada. Una vez aquí, se selecciona "Dependencias adicionales" y se pone en cada línea:

1. "Cv210.lib"
2. "Cxcore210.lib"
3. "Highgui210.lib"

Cuando ya se tenga todo esto hecho, ya se tendrá el proyecto listo para poder utilizar la librería OpenCv. Evidentemente, en el código se tendrá que añadir las cabeceras pertinentes.



8. Bibliografía

[1] J. E. Stone, D. J. Hardy, J. Saam, K. L. Vandivort, and K. Schulten. GPU-accelerated computation and interactive display of molecular orbitals. Editor, GPU Computing Gems, vol. 1, pp. 5-18. Morgan Kaufmann Publishers, 2011.

[2] J. E. Stone, D. J. Hardy, B. Isralewitz, and K. Schulten. GPU algorithms for molecular modeling. Editors, *Scientific Computing with Multicore and Accelerators*, vol. 16, pp. 351-371. Chapman & Hall/CRC Press, 2011.

[3] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger. Fast GPU-Based CT Reconstruction using the Common Unified Device Architecture (CUDA). Editor, Nuclear Science Symposium Conference Record, vol. 6, pp. 4464-4466, IEE, 2007.

[4] SeismicCity. Tesla and CUDA Technologies Transform the Oil and Gas Industry. Editors, DE. 2008.

[5] I. Schmerken. Wall Street Accelerates Options Analysis With GPU Technology. Editors, At large. Advanced Trading and Wall Street & Technology,2009.

[6] R. Díaz, L. Cervera, S. Fenollosa. Visión Espectral aplicada al Control de Calidad en la Industria Agroalimentaria. Alimentación, equipos y tecnología, Septiembre, 2011.

[7] R. Díaz, L. Cervera, S. Fenollosa, C. Ávila, J. Belenguer. Hyperspectral system for the detection of foreign bodies in meat products. Eurosensors XXV, Septiembre, 2011.

[8] R. Bro, A.K Smilde, Centering and scaling in component analysis, Journal of Chemometrics, v.17, pp. 16-33, 2003.

[9] M. Zeaiter, D. Rutledge, Preprocessing Methods. Comprehensive Chemometrics, Elsevier, pp. 121-231, 2009.

[10] A. Savitzky, M.J.E. Golay, “Smoothing and differentiation of data by simplified least squares procedures”, vol 36, pp. 1627, Analytical Chemistry,1964.

[11] H.W. Borchers, “Savitzky-Golay smoothing an R implementation” in

http://tolstoy.newcastle.edu.au/R/help/04/02/0385.html

[12] W.H. Press, B.P Flannery, S.A. Teukolsky, W.T. Vetterling, “Numerical recipes in C: The Art of Scientific Computing”, Cambridge Univesity Press, Cambridge, 1992.

[13] W.H Press, S.A. Teukolsky, “Savitzky-Golay smoothing filters”, Computers in Physics, p.669, 1990.

[14] J.E. Jackson, Principal component and factor analysis: Part 1-Principal components, vol 13,pp. 1, J.Qual. Tech, 1981.

[15] J.V. Kresta, J.F. MacGregor, T.E. Marlin, Multivariate statistical monitoring of process operating performance, vol 69, pp. 35-47, Can. J. of Chem. Eng, 1991.

[16] J.W. Gardner, Detection of vapours and odours from a multisensor array using pattern recognition. Part 1: principal components and cluster analyses, vol 4, pp. 108-116, Sensors and Actuators B, 1991.

[17] W. P. Hays, “DSPs: Back to the Future”. Magazine Queue , vol 1, pp. 42-51, ACM, Mar. 2004.

[18] S. Guccione, *Programming Fine-grained Reconfigurable Architectures*, Ph. D. Thesis, University of Texas at Austin, 1995.

[19] J. Oldfield, R. Dorf, "Field-Programmable Gate Arrays. Reconfigurable Logic for Rapid Prototyping and Implementation of Digital Sistems", John Wiley & Son. 1995.

[20] M. M. Parelkar, FPGA Security – Bitstream Authentication, George Mason University, Fairfax.

[21] S.D. Brown. FPGA Architectural Research: A Survey. Test of Computers, vol. 13, pp. 9-15, IEEE,1996.

[22] S. M. Kuo, W.S. Gan. “Digital Signal Processors: Architectures, Implementations, and Applications”, pp. 1-39, Pearson Prentice Hall, 2005.

[23] C. E. Shannon, "Communication in the presence of noise", Proc. Institute of Radio Engineers, vol. 37, pp. 10–21,IEE Jan. 1949.

[24] M. J. Flynn, Computer architecture: pipelined and parallel processor design,1995.

[26] D.H. Patterson, Computer organization and design: the hardware/software interface, Morgan Kaufmann, 2008.

[27] K. Hwang. Advanced computer architecture: parallelism, calability, programmability, McGraw-Hill, 2003.

[28] E.K Sanders, CUDA by Example, An Introduction to General Purpose GPU Programming, Addison-Weslley, 2010.

[29] NVIDIA CUDA C Programming Guide: Version 4.2, Abril, 2012.

[30] CUBLAS Library. CUDA Toolkit 4.2. Febrero, 2012.

[31] GeForce GTX 560 Ti

http://www.nvidia.es/object/product-geforce-gtx-560ti-es.html

[32] GeForce GTX 260

http://www.nvidia.es/object/geforce_gtx_260_es.html