UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

INGENIERÍA DE TELECOMUNICACIÓN

Trabajo Fin de Carrera

"Paralelización de algoritmos para el procesado de imágenes de teledetección"

Alberto Monescillo Álvarez

Noviembre 2009

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

INGENIERÍA DE TELECOMUNICACIÓN

Trabajo Fin de Carrera

"Paralelización de algoritmos para el procesado de imágenes de teledetección"

Alumno: Alberto Monescillo Álvarez

Director: Felipe Bertrand Galarza

Tutor: Javier Macías Guarasa

Tribunal:

Presidente: D. Daniel Pizarro Pérez.

Vocal 1º: D. Alfredo Gardel Vicente.

Vocal 2º: D. Javier Macías Guarasa.

Calificación: .....

Fecha: .....

Agradecimientos

En primer lugar quiero dar las gracias a mis padres, Joaquín y Rosa, por haberme apoyado durante todos estos años y haber estado siempre a mi lado, ya que sin ellos hoy no sería la persona que soy.

También a mi hermano Carlos, por haberme aguantado durante todos los años de carrera.

A mi novia Sandra, que sin duda es la persona que mejor me comprende y que mas me ha animado en los momentos dificiles de la carrera. Sin ella el camino hacia la meta final hubiera sido mucho mas complicado. Eres lo mejor que me llevo de mi experiencia en la universidad. Gracias por estar siempre a mi lado.

A mis abuelos, Andrés y Paquita, por haber confiado y creído en mi.

Finalmente quiero dar las gracias a mi jefe de proyecto Felipe, por haber confiado en mí y por haberme dado la oportunidad de comenzar mi carrera profesional junto a él, ya que durante estos meses he aprendido muchas cosas gracias a él y he podido disfrutar de una experiencia muy gratificante. También a mi tutor Javier, por haberme resuelto todas las dudas que me surgieron durante el desarrollo del proyecto.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Resumen

La teledetección es la observación y medida de objetos situados sobre la superficie terrestre sin estar en contacto con ellos. Se basa en la detección y registro de la energía reflejada o emitida desde un objeto, y el procesado, análisis y aplicación de esa información. Todos los datos capturados por el sensor del satélite deben ser procesados mediante diferentes algoritmos, lo que implica un tiempo de procesado elevado hasta generar la correspondiente información de salida.

Es por ello que hoy en día uno de los grandes objetivos en el ámbito de la teledetección es implementar algoritmos que sean capaces de reducir el tiempo empleado por la cadena de procesado para generar las imágenes de salida, y de esta forma aproximarse cada vez más a un sistema de tiempo real. En este proyecto se pretenden mejorar estas prestaciones mediante el uso de técnicas de paralelización, con el objetivo final de suministrar datos de teledetección en tiempo casi-real.

Nos vamos a centrar en analizar la tecnología *CUDA* de las tarjetas gráficas *Nvidia* [5]. Se hará una descripción sobre esta tecnología, comentando y analizando las principales características que presenta, así como las ventajas que puede suponer para el procesado de datos espaciales en sistemas de teledetección. Posteriormente se describirán las diferentes pruebas y aplicaciones realizadas. La línea de trabajo seguida se divide en dos partes. Por un lado se ha desarrollado una aplicación que permite realizar reconocimiento de terreno en base a imágenes de *Landsat*, y por otro lado se ha utilizado la cadena de procesado genérica *AIPC* de Deimos Space para la generación de imágenes de teledetección. Dentro de esta segunda parte se han realizado pruebas de paralelización sobre los módulos de Calibración absoluta, Detección de nubes, Deconvolución, Reducción de ruido y Corregistración.

Capítulo 2

Introducción

2.1. Introducción

La teledetección es la observación y medida de objetos situados sobre la superficie terrestre sin estar en contacto con ellos. Se basa en la detección y registro de la energía reflejada o emitida desde un objeto, y el procesado, análisis y aplicación de esa información [6]. El dispositivo que detecta esta radiación electromagnética se llama sensor remoto o simplemente sensor, y suele ser transportado en una plataforma aérea, como un avión o un satélite.

En la Figura 2.1 se muestra un ejemplo de todo el proceso que sigue un sistema de teledetección y los diferentes elementos que intervienen en él.

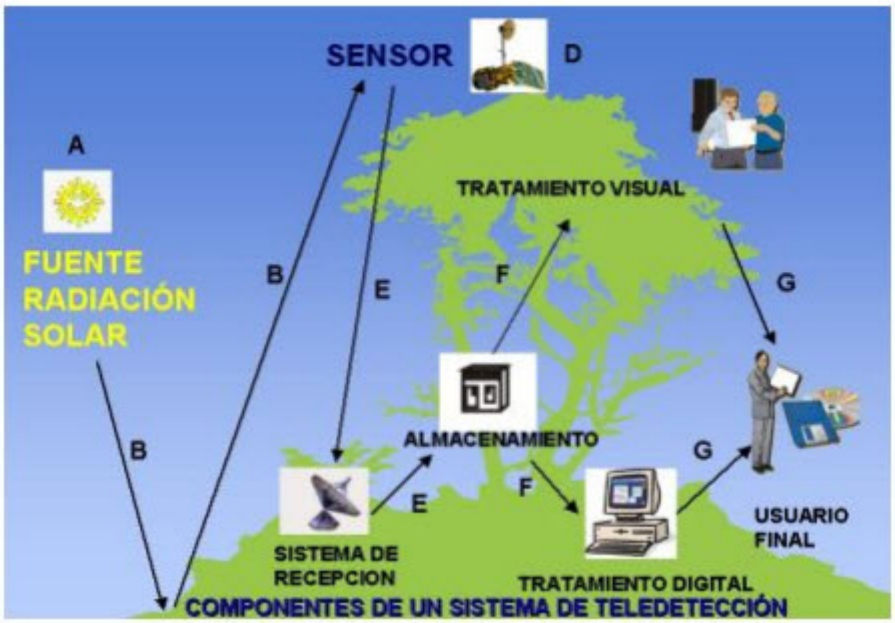

Figura 2.1: Componentes de un sistema de teledetección. (Fuente: [1])

Con la tecnología actual existe un amplio abanico de sensores que permiten realizar una observación de la tierra. Los más utilizados son:

- Instrumentos ópticos de alta resolución: Este tipo de sensores se subclasifican por su resolución y la banda de frecuencias en que trabaja. La resolución determina el detalle espacial y espectral de la imagen, mientras que la banda de frecuencias establece los fenómenos físicos que va a capturar. Las bandas de frecuencias que se suelen utilizar son: banda azul, banda verde, banda roja, infrarrojo cercano e infrarrojo de onda corta.
- Radar de Apertura Sintética (SAR): Es un tipo de sensor activo basado en tecnología RADAR. Es, por lo tanto, un instrumento que usa ondas de radio para estimar la distancia y posición de objetos remotos. La peculiaridad del SAR es que, en lugar de usar una gran antena con muy buena direccionalidad, y por lo tanto muy buena resolución, usa una antena pequeña con mala direccionalidad, pero obtiene muestras desde muchas posiciones distintas, consiguiendo una resolución muy precisa.
- Instrumentos Radiométricos Pasivos: Suelen operar en la banda de las microondas, ya que a estas frecuencias se producen muchos fenómenos y propiedades físicas. La diferencia fundamental con los instrumentos ópticos es su resolución, y que la detección debe realizarse con antenas.

La teledetección espacial proporciona numerosas ventajas, entre las que cabe destacar:

- La superficie de observación del satélite abarca una zona geográfica extensa.
- Se ofrece cobertura periódica.
- La cobertura puede ser global en el caso de órbitas polares.
- Los datos capturados se pueden procesar para obtener productos en tiempo casi real.
- La información multiespectral permite la generación de productos de valor añadido y la obtención de variables biofísicas con importante soporte científico, como índices de vegetación, índices de humedad...
- La teledetección es un método preciso, estandarizado y armonizado de medida que permite comparar datos de zonas geográficas remotas y dispares.
- La teledetección está reconocida y promovida por instituciones europeas y organizaciones multilaterales, tales como la Agencia Espacial Europea (ESA), Comisión Europea, FAO...

El uso actual de la teledetección abarca una gran cantidad de campos, como los ámbitos de la cartografía, agricultura, conservación de bosques, meteorología, climatología, militar, etc.

La Agencia Espacial Europea (ESA), ha desarrollado diferentes misiones basadas en la teledetección, que permiten realizar una observación de la superficie terrestre. Entre ellas cabe destacar Meteosat, ERS y ENVISAT, que proporcionan imágenes para el estudio medioambiental y climático [7]. En función del tiempo que se tarde en procesar los datos capturados por el sensor y generar las correspondientes imágenes, la ESA clasifica los servicios en:

- Near Real Time (NRT)
- 48 hours
- 1 month

Es por ello que hoy en día uno de los grandes objetivos en el ámbito de la teledetección es implementar algoritmos que sean capaces de reducir el tiempo empleado por la cadena de procesado para generar los productos, y de esta forma aproximarse cada vez más a un sistema de tiempo real.

2.2. Desarrollo del proyecto

Para el desarrollo de este proyecto se ha optado por trabajar con los datos capturados por el sensor óptico incorporado en el satélite *Landsat* 5, cuyas especificaciones se pueden ver en la Tabla 2.1.

Asímismo, las características de las bandas espectrales que utiliza se pueden ver en la Tabla 2.2

Una vez que los datos han sido capturados por el sensor, éstos deben pasar por una cadena de procesado formada por una serie de etapas, a través de la cual se van aplicando una serie de algoritmos para corregir posibles errores, georeferenciar las imágenes, etc, para dotarlas de valor científico.

Característica	**Valor**
Altura de vuelo	705 Km
Ciclo de recubrimiento	16 días
Periodo orbital	98.9 minutos
Hora de adquisición en España	9:45 a.m. hora solar
Resolución espacial	30 m
Resolución espectral	7 bandas
Resolución radiométrica	8 bits

Tabla 2.1: Especificaciones sensor incorporado en Landsat 5

Banda	**Longitud de onda (μm)**	**Región del espectro**
1	0.45 - 0.52	Azul
2	0.52 - 0.60	Verde
3	0.63 - 0.69	Rojo
4	0.76 - 0.90	Infrarrojo cercano
5	1.55 - 1.75	Infrarrojo medio 1
6	10 - 12	Infrarrojo térmico
7	2.08 - 2.35	Infrarrojo medio 2

Tabla 2.2: Bandas Landsat 5

Cuando los datos han pasado por todas estas etapas y se ha generado la imagen de salida, ésta puede ser utilizada por una gran cantidad de aplicaciones en función del fin que se persiga. En nuestro caso, el usuario que va a hacer uso de éstas imágenes será la plataforma de España Virtual [6]. España Virtual es un proyecto desarrollado por un consorcio de empresas en el que se pretende sentar las bases de un futuro ecosistema de contenidos multimedia y servicios interactivos que reúna las tecnologías conocidas como Web 2.0 con los aspectos sociales, semánticos y geográficos (ortofotos, imagen satélite, modelos digitales de terreno, edificios 3D...), en una nueva generación de herramientas 3D de interacción con el mundo virtual. Esta plataforma permitirá que exista un mecanismo sencillo e interoperable para publicar información multimedia georreferenciada.

En todo sistema de teledetección es necesario procesar una gran cantidad de información, lo que implica un tiempo elevado desde que el satélite captura los datos hasta que se genera la información de salida.

En la actualidad existen diferentes tecnologías que permiten realizar un procesado masivo de los datos, consiguiendo así reducir el tiempo empleado por la cadena de procesado, y dando lugar a un sistema de tiempo real o casi-real.

En este proyecto vamos a realizar un estudio sobre las arquitecturas de sistemas paralelos, y más concretamente sobre la tecnología basada en tarjetas gráficas, analizando las posibles mejoras que puede introducir en un sistema de teledetección.

Actualmente las tarjetas gráficas disponen de una gran cantidad de nodos de procesamiento, que permiten realizar una paralelización de la aplicación. En este sector existe tecnología tanto de *Nvidia* como de *ATI*. Ambas se caracterizan por el uso de librerías y compiladores específicos proporcionados por los fabricantes, mediante los cuales los programadores de aplicaciones pueden utilizar los cientos de nodos de procesamiento de estas tarjetas para realizar cálculos complejos a gran velocidad y con un coste reducido.

En el caso que nos ocupa se ha optado por utilizar la tecnología *CUDA* de las tarjetas gráficas de *Nvidia* [5].

Para ello la línea de desarrollo seguida se ha dividido en dos partes. Por un lado se ha diseñado una aplicación que permite realizar reconocimiento de terreno a partir de imágenes de *Landsat*, y por otro lado se ha trabajado sobre el proyecto *AIPC* (Autonomous Image Processing Chain) de Deimos Space [8], el cual implementa una cadena de procesado genérica para sistemas de teledetección.

En ambos casos se va a trabajar con imágenes TIFF, por lo que ha sido necesario diseñar una aplicación que permita manejar este tipo de formato [9].

La primera línea de desarrollo se basa en la realización de una aplicación para el reconocimiento de terreno. Esta técnica es muy utilizada en los sistemas de teledetección, ya que tiene infinidad de campos de aplicación como se verá en el correspondiente capítulo.

La línea de investigación seguida en la segunda parte se ha basado en el proyecto *AIPC* (Autonomous Image Processing Chain) de Deimos Space [8], el cual implementa una cadena de procesado genérica para sistemas de teledetección.

Esta cadena de procesado está formada por una serie de algoritmos, agrupados a su vez en diferentes módulos con una funcionalidad específica [10]. Estos módulos implementan todas las etapas necesarias para procesar los datos en crudo capturados por un sensor óptico, y generar las correspondientes imágenes de valor añadido. El esquema utilizado por esta cadena de procesado se puede ver en la Figura 2.2.

Como se puede observar en esta figura, la cadena de procesado genérica AIPC está formada por diferentes módulos [11]. Entre ellos destacan los siguientes:

- Calibración absoluta
- Ecualización
- Detección de nubes
- Deconvolución
- Corregistración
- Ortorectificación
- Reducción de ruido

Partiendo de los algoritmos utilizados en cada uno de estos módulos, se ha pasado a realizar una paralelización de los mismos mediante la tecnología *CUDA* para mejorar sus prestaciones. Concretamente se han realizado pruebas con cinco de ellos, como se describe en los correspondientes capítulos.

2.3. Objetivos y resumen de resultados

El objetivo principal de este proyecto es optimizar una serie de algoritmos de procesado de imágenes de teledetección mediante técnicas de paralelización basadas en la tecnología *CUDA* [6], con el fin de obtener un sistema de baja latencia, es decir, capaz de procesar una gran cantidad de datos a alta velocidad, y así generar las imágenes en un tiempo reducido.

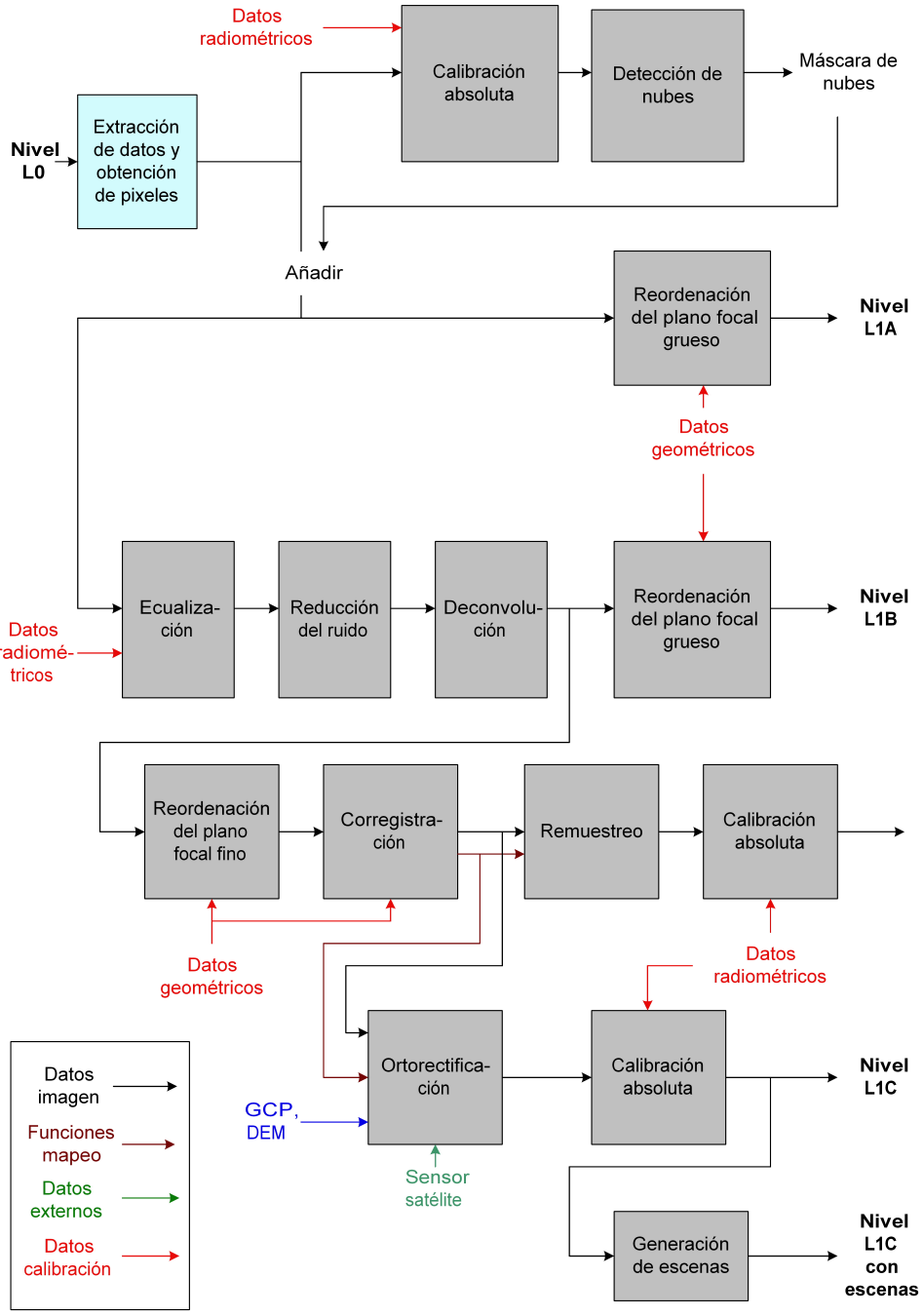

Figura 2.2: Cadena de procesado de imágenes

Para ello se han desarrollado diferentes aplicaciones basadas en un sistema de teledetección, donde cada una de ellas se encarga de implementar toda la funcionalidad asociada a una etapa concreta de la cadena de procesado.

Cada una de las aplicaciones ha sido desarrollada para que pueda ser ejecutada mediante paralelización o sin paralelización, de forma que se puedan comparar los tiempos de ejecución obtenidos en ambos casos.

Estos tiempos han sido calculados promediando el tiempo de procesado obtenido en diferentes ejecuciones de la misma aplicación, para así evitar posibles errores por sobrecargas de los procesadores.

Para verificar que el procesado que se realiza sobre la imagen de entrada mediante paralelización es el mismo que sin paralelización, las imágenes obtenidas en ambos casos han sido comparadas píxel a píxel para comprobar que son exactamente iguales.

A partir de los resultados obtenidos tras la realización del proyecto, se extrae que la tecnología *CUDA* introduce notables mejoras en un sistema de teledetección, ya que reduce considerablemente los tiempos empleados por los diferentes módulos que intervienen en la cadena de procesado. Como se verá en capítulos posteriores, cuanto mayor sea la cantidad de datos a procesar y mayor sea la carga computacional de las operaciones que se realizan, mayor es el ahorro de tiempo obtenido.

2.4. Estructura del documento

Los resultados obtenidos durante la realización del proyecto han sido descritos en diferentes capítulos.

Inicialmente hay un capítulo dedicado a la tecnología *CUDA*, donde se analiza en que consiste esta tecnología, se describen las principales características que presenta, y se comenta la plataforma de desarrollo sobre la que debe trabajar.

A continuación se describen las pruebas realizadas sobre diferentes módulos que componen un sistema de teledetección. Para cada uno de ellos se hará una descripción sobre el objetivo que persigue y sus principales características, se analizarán las aplicaciones desarrolladas en C y *CUDA*, y finalmente se mostrarán los resultados obtenidos. Estos resultados estarán compuestos por los tiempos de ejecución de la aplicación sin paralelización y con paralelización, así como por las imágenes de salida en ambos casos.

Finalmente se extraerán una serie de conclusiones sobre la tecnología *CUDA* y los beneficios que puede introducir en un sistema de teledetección.

Capítulo 3

Tecnología CUDA

3.1. Introducción

La tecnología *CUDA* es una arquitectura de computación paralela desarrollada por uno de los mayores fabricantes de tarjetas gráficas del mercado, *Nvidia*.

Fue introducida en Noviembre de 2006, y se basa en la utilización de un elevado número de nodos de procesamiento para realizar operaciones sobre un gran volumen de datos en paralelo, consiguiendo reducir el tiempo de procesado y obteniendo altas prestaciones.

Uno de los principales objetivos de esta tecnología es resolver problemas complejos que lleven asociados una alta carga computacional sobre la CPU de la forma mas eficiente, haciendo uso de las GPUs (Unidades de procesamiento gráficas) incorporadas en la tarjeta gráfica. Para ello se realiza un procesado masivo de los datos, consiguiendo reducir considerablemente los tiempos de ejecución de la aplicación.

Para desarrollar aplicaciones basadas en esta tecnología se pueden utilizar entornos de desarrollo basados en lenguajes de alto nivel como C, uno de los lenguajes de programación mas utilizados actualmente, aunque se pretende que en un futuro también se puedan utilizar otros lenguajes como C++, Fortran, OpenCL... como se puede apreciar en la Figura 3.1

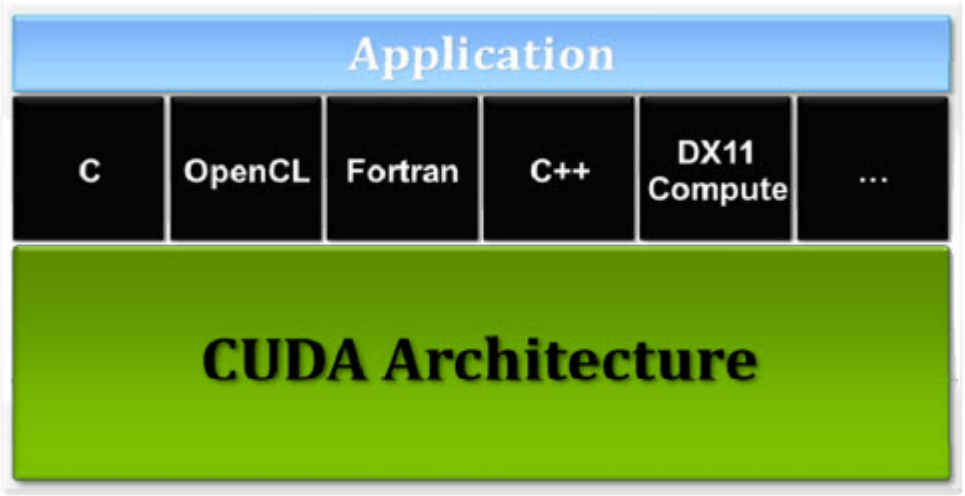

Figura 3.1: Entornos de desarrollo utilizados con CUDA. (Fuente: [2])

El proceso que se sigue consiste en desarrollar aplicaciones basadas en un lenguaje de alto nivel, para posteriormente pasar a un nivel mas bajo mediante la inclusión de una serie de extensiones o instrucciones específicas sobre el código original que permitan paralelizar la aplicación. De esta forma se consigue que parte del código sea ejecutado por los multiprocesadores incorporados en la tarjeta gráfica en vez de por la CPU, con el consiguiente ahorro en tiempo de procesado.

La parte del código que sea ejecutada por los multiprocesadores lo hará mediante paralelización, de forma que una misma operación será ejecutada de forma simultánea sobre varios datos de entrada, (se conoce como arquitectuta SIMP según la taxonomía de Flynn [12]), mientras que el resto de la aplicación será ejecutada de forma secuencial por la CPU.

El beneficio que se obtiene al utilizar esta tecnología se puede comprobar en la Figura 3.2, donde se comparan las máximas prestaciones proporcionadas por diferentes modelos de tarjetas gráficas *Nvidia*, frente a las obtenidas en diferentes modelos de CPUs.

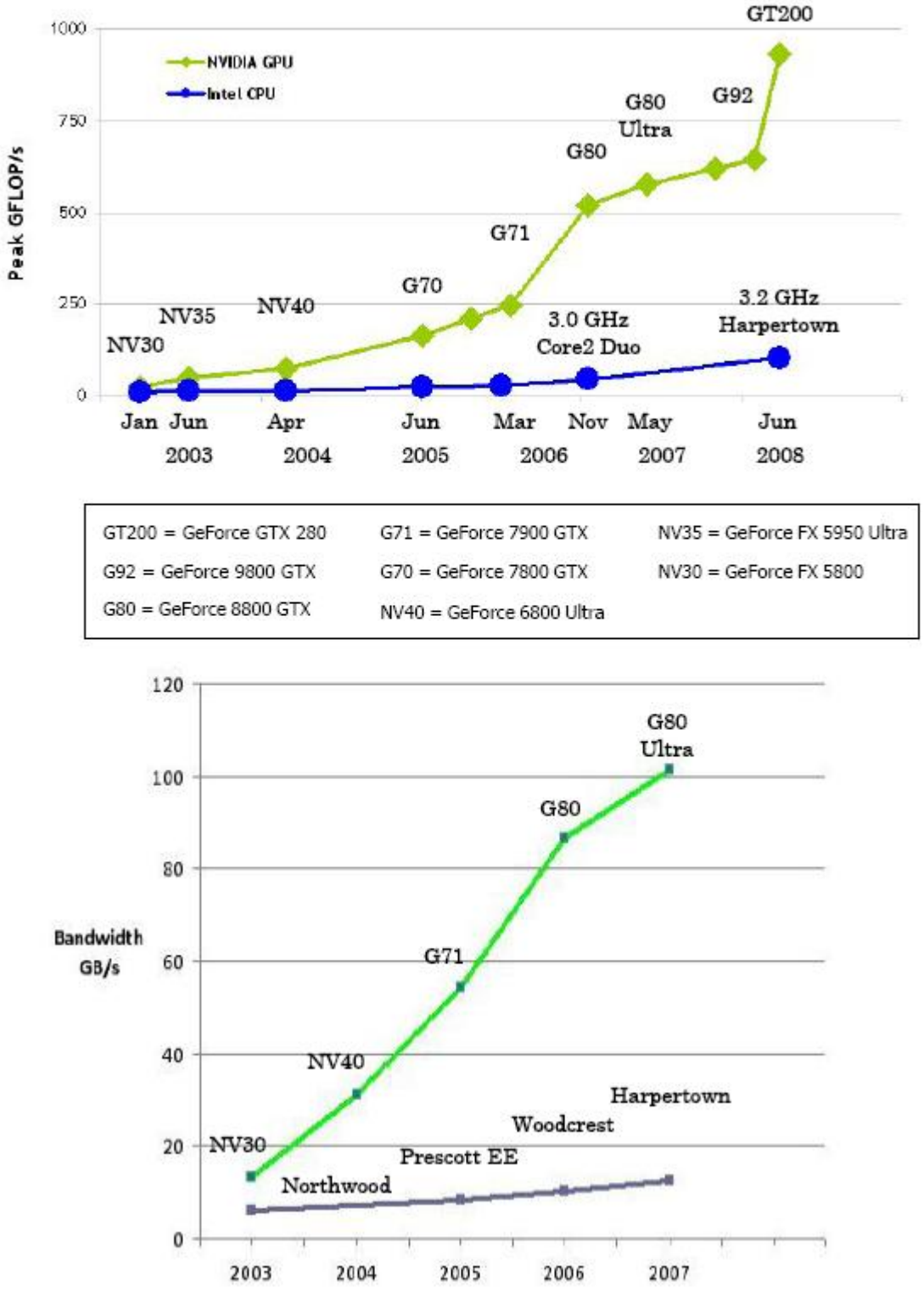

Figura 3.2: Comparativa entre GPU y CPU. (Fuente: [2])

Como se puede observar, la tecnología CUDA permite realizar un mayor número de operaciones por segundo sobre la GPU frente a una CPU convencional, así como obtener un ancho de banda considerablemente superior. Las prestaciones proporcionadas por la GPU dependerán del modelo de tarjeta gráfica que se utilice, y como vemos, éstas han aumentado considerablemente en los últimos años, a diferencia de las CPU's, donde la evolución ha sido menor.

3.2. Plataforma de desarrollo

Para desarrollar aplicaciones con la tecnología *CUDA* es necesario disponer de una tarjeta gráfica *Nvidia* compatible. Actualmente existen una gran cantidad de modelos y productos compatibles con esta tecnología, los cuales se pueden agrupar en los siguientes tres conjuntos:

- NVIDIA GeForce 8, 9, and 200 series GPUs.
- NVIDIA Tesla computing solutions.
- NVIDIA Quadro products.

En cuanto al sistema operativo, cabe mencionar que esta tecnología es compatible con sistemas operativos Windows y Unix, tanto para versiones de 32 como de 64 bits. En el caso de trabajar con un sistema Unix es importante verificar que sea compatible con *CUDA*. Las distribuciones de Linux compatibles son:

- Red Hat Enterprise Linux 4.3-4.7, 5.0-5.2.
- SUSE Enterprise Desktop 10-SP2.
- Open SUSE 10.3 or 11.0.
- Fedora 8 or 9.
- Ubuntu 7.10 or 8.04.

Una vez que ya disponemos del hardware y el sistema operativo adecuado, pasamos a la instalación del software característico de *CUDA*.

Para ello es necesario instalar el paquete de herramientas proporcionado por el fabricante *Nvidia*. Este software se puede descargar de forma gratuita de su pagina web [1].

El proceso que se debe seguir para su correcta instalación es [13]:

- Descargar los drivers de la tarjeta gráfica y proceder a su instalación.
- Descargar el software de la tarjeta gráfica y proceder a su instalación.

Es muy aconsejable tener actualizados los drivers de la tarjeta gráfica a su última versión para asegurar un correcto funcionamiento, así como que el software descargado sea compatible con nuestro sistema operativo.

El software proporcionado por el fabricante está dividido en dos partes:

[1] `http://www.nvidia.com/object/cudaget.html`

- *CUDA toolkit*: Herramientas necesarias para crear aplicaciones y proceder a su compilación. Contiene librerías, cabeceras, compilador...
- *CUDA SDK*: Contiene una serie de ejemplos de aplicaciones realizadas con esta tecnología.

Dentro del *SDK* de *CUDA* hay disponibles una serie de ejemplos que pueden ser utilizados para verificar el correcto funcionamiento de la tarjeta gráfica, así como para obtener las características de la misma. Es aconsejable ejecutar el programa *deviceQuery* [13], el cual nos muestra por pantalla toda la información asociada a nuestro modelo de tarjeta gráfica, como podemos en la Figura 3.3. Estos ejemplos proporcionados por el fabricante son una buena forma de familiarizarse con el entorno de programación, y ver las diferentes partes en las que se suele descomponer una aplicación desarrollada con la tecnología *CUDA*.

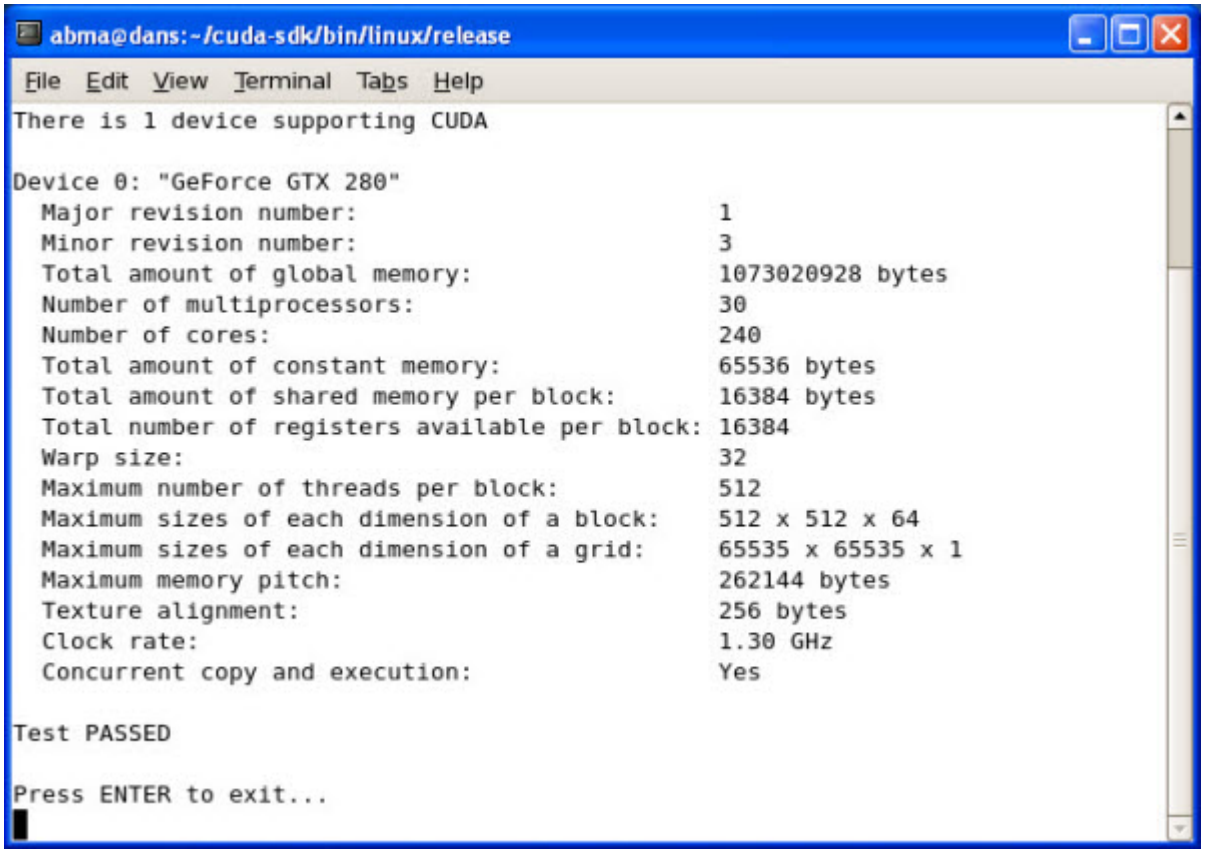

Figura 3.3: Aplicación 'DeviceQuery' del software de CUDA

Una vez hayamos instalado correctamente el software de *CUDA*, ya podemos pasar a desarrollar nuestras aplicaciones mediante paralelización.

Para implementar una aplicación mediante la tecnología *CUDA* hay que tener en cuenta que ésta debe estar formada por al menos dos ficheros, uno con extensión .c o .cpp (en función de si se quiere desarrollar la aplicación en C o en C++), que contendrá el código que será ejecutado por la CPU, y otro con extensión .cu, el cual contendrá todo el código que va a ser ejecutado por la tarjeta gráfica [2].

Esto implica que al tener archivos con diferentes extensiones se necesiten dos compiladores. Uno que realice la compilación del archivo .cu y otro del archivo .c. Para ello se utilizan los siguientes:

- Compilador *nvcc*: Herramienta proporcionada por el fabricante *Nvidia* que permite realizar la compilación del archivo con extensión .cu. Este compilador se encarga de separar el código que va a ser ejecutado por la tarjeta gráfica del que será ejecutado por la CPU, compila únicamente el primero, y genera el correspondiente fichero objeto [2].
- Compilador *gcc*: Herramienta estándar que permite compilar código C. Se encarga de compilar el código que será ejecutado por la CPU, enlazar los correspondientes ficheros objeto (incluyendo el generado por *nvcc*), y generar el archivo ejecutable.

Una vez que ambos ficheros han sido compilados y se han generado los ficheros objeto y el correspondiente archivo ejecutable, ya se tiene disponible la aplicación. Cuando ésta sea ejecutada, parte del código será ejecutado por la CPU de forma secuencial, y el resto por los diferentes procesadores incorporados en la tarjeta gráfica.

Todo este proceso descrito se puede ver gráficamente en la Figura 3.4:

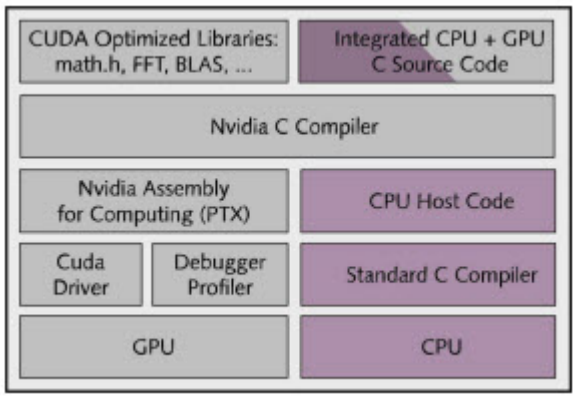

Figura 3.4: Plataforma CUDA para procesamiento paralelo en GPUs. (Fuente: [3])

Finalmente cabe mencionar que la tarjeta gráfica requiere de un proceso de inicialización, el cual implica un tiempo aproximado de un segundo. Este tiempo no será incluido en el tiempo total con paralelización obtenido durante la realización de diferentes pruebas, como se verá en posteriores capítulos.

3.3. Conceptos característicos

A la hora de desarrollar una aplicación con la tecnología *CUDA* hay que tener muy claro una serie de conceptos característicos de esta tecnología. A continuación se irán describiendo algunos de los aspectos mas importantes.

Tres conceptos muy utilizados al diseñar aplicaciones mediante paralelización son: hilo, bloque de hilos y grid.

- Hilo (thread): Cada uno de los hilos de control que van a ejecutar de manera simultánea una función (kernel).
- Bloque de hilos (thread block): Representa un conjunto de hilos que son lanzados sobre el mismo multiprocesador de la tarjeta gráfica. Pueden tener comunicación entre ellos, ya que disponen de un espacio de memoria compartida. Puede haber hasta 512 hilos por bloque.

- Grid: Representa todo el conjunto de bloques de hilos. Los diferentes bloques de hilos se pueden comunicar entre sí por medio del espacio de memoria global.

Estos tres términos se pueden ver gráficamente en la Figura 3.5, donde se puede comprobar como un hilo forma parte de un bloque de hilos, y a su vez un bloque de hilos pertenece a un grid.

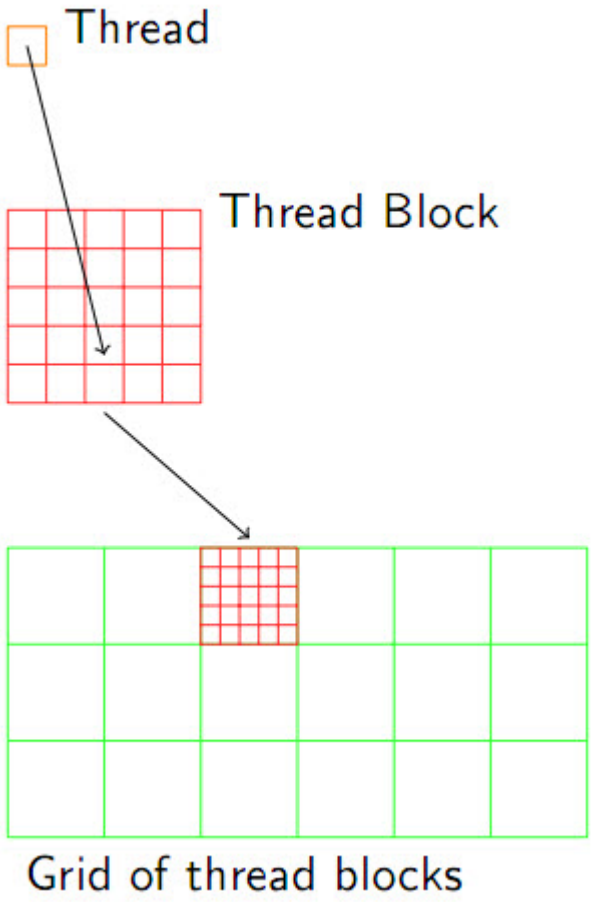

Figura 3.5: Hilo, bloque de hilos y grid. (Fuente: [3])

Otro término muy importante y novedoso de esta tecnología es el concepto de *kernel* [2]. Un *kernel* es una función desarrollada en lenguaje C que será ejecutada de forma simultánea por varios multiprocesadores de la tarjeta gráfica. Por lo tanto, va a contener el código que será ejecutado en paralelo por diferentes hilos.

Un *kernel* se define con el calificativo '__global__ ', como podemos ver en el siguiente fragmento de código:

```
__global__ void Calibracion(double gain, double offset, int size,
 double* imagen_d)
{
        int bx = blockDim.x * blockIdx.x;
        int tx = threadIdx.x;
```

```
        if(bx+tx < size)
        {
                imagen_d[bx+tx] = imagen_d[bx+tx] * gain + offset;
        }
}
```

Cuando se ejecuta un *kernel* hay que indicar el número de bloques de hilos y el número de hilos por bloque que van a ejecutar la función, teniendo en cuenta las limitaciones en cuanto al número máximo en función del modelo de tarjeta gráfica disponible (en nuestro caso ver Tabla 3.1). Estos parámetros se indican en la propia llamada a la función [14] mediante dos variables, como se puede ver en el siguiente ejemplo:

```
Calibracion <<< dimGrid, dimBlock >>> (gain, offset, size, imagen);
```

En este caso 'dimGrid' indica el número de bloques de hilos, y 'dimBlock' el número de hilos por bloque.

Cada hilo y cada bloque de hilos tienen asociados un identificador, que permiten acceder a ellos a través del kernel, como se puede ver en la Figura 3.6. Estos identificadores quedan representados por las variables blockDim.x, blockIdx.x y threadIdx.x, las cuales son accesibles desde el kernel.

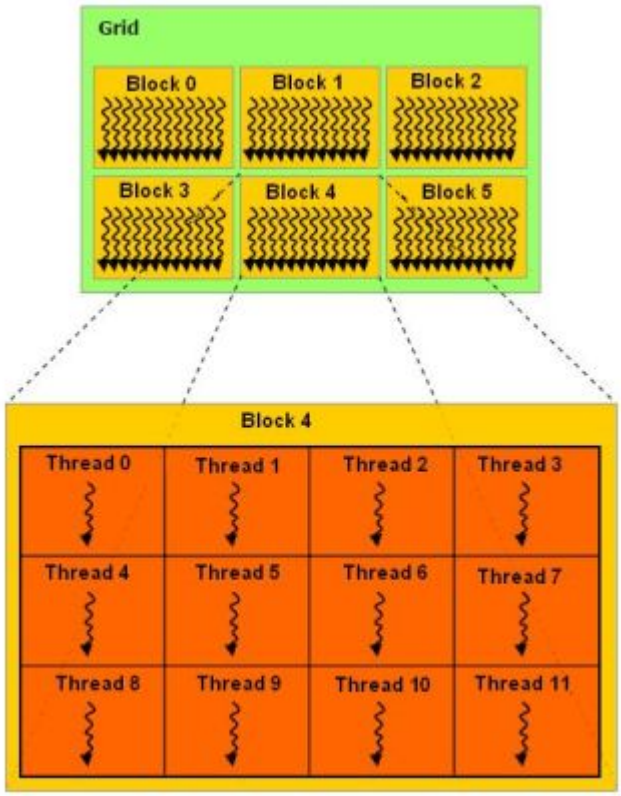

Figura 3.6: Estructura de bloques e hilos. (Fuente: [2])

Dentro de un kernel se pueden establecer puntos de sincronización entre todos los hilos de un mismo bloque. Para ello se usa la instrucción:

__syncthreads() ,

la cual fija una barrera de forma que cada hilo no continúa con la ejecución del kernel hasta que todos los hilos del mismo bloque no hayan llegado al punto de sincronización.

Otro aspecto a tener en cuenta cuando se trabaja con la tecnología *CUDA* es que se dispone de dos espacios totalmente diferenciados. Por un lado todo lo relacionado con la CPU, con su correspondiente espacio de memoria, y por otro lado todo lo relacionado con la tarjeta gráfica, también con su correspondiente espacio de memoria [2].

Esto da lugar a que en toda aplicación desarrollada con esta tecnología, parte de ella sea ejecutada por la CPU por un único proceso, y otra parte por la GPU de forma simultánea por varios hilos, como se puede ver en la Figura 3.7:

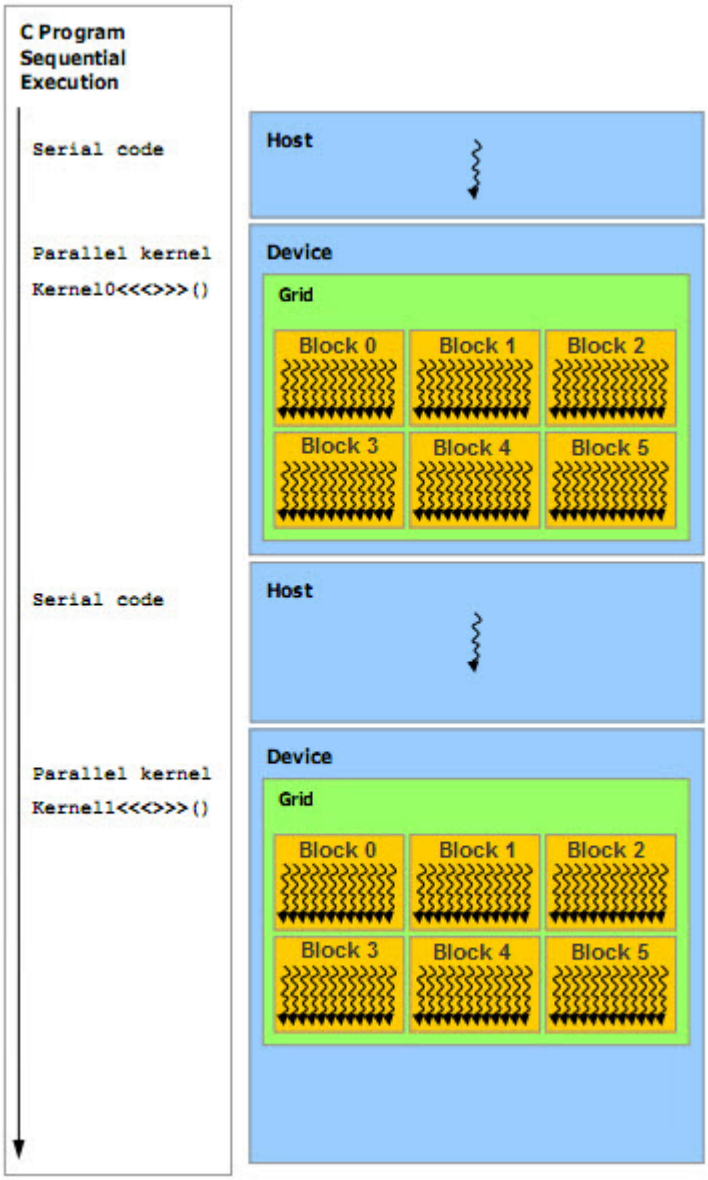

Figura 3.7: Proceso de ejecución de una aplicación con CUDA. (Fuente: [2])

Dentro del espacio de la tarjeta gráfica cabe destacar que la memoria define diferentes jerarquías, como se puede ver en la Figura 3.8:

- Memoria global: Zona de memoria accesible por los hilos de todos los bloques.
- Memoria compartida: Zona de memoria compartida por todos los hilos que pertenecen a un mismo bloque.
- Memoria local privada: Zona de memoria exclusiva para un determinado hilo.

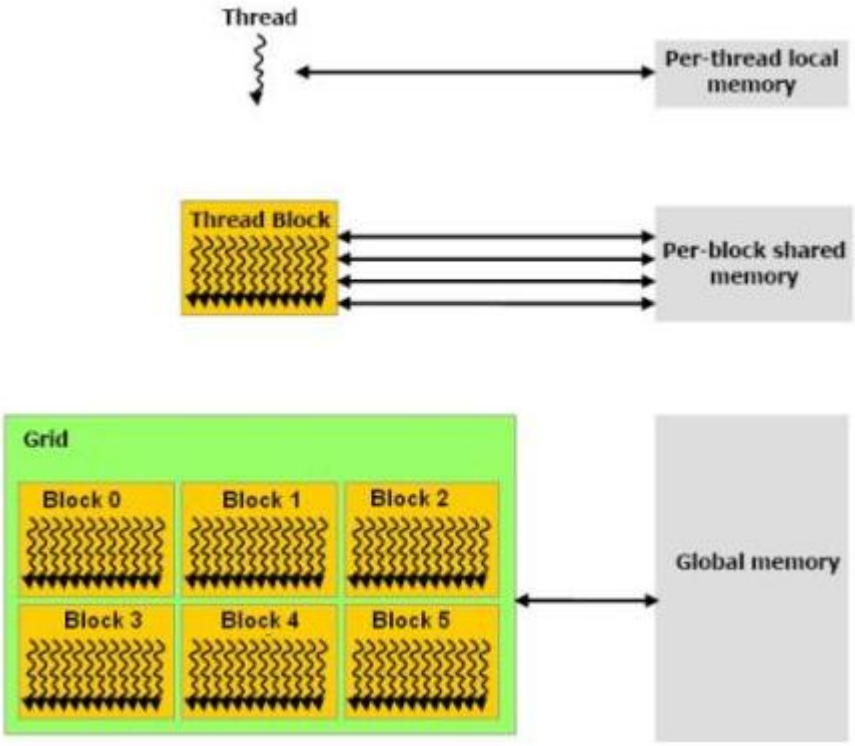

Figura 3.8: Jerarquía de memoria en la GPU. (Fuente: [2])

Hay que resaltar que el espacio de memoria compartida es mas rápido que el global para realizar las operaciones, por lo que conviene trabajar sobre él siempre que la aplicación lo permita.

Para obtener el máximo rendimiento de la tecnología *CUDA* hay que intentar ejecutar el mayor número de hilos en cada uno de los multiprocesadores de la tarjeta gráfica, de forma que todos ellos estén ocupados realizando operaciones, y por lo tanto aprovechando al máximo sus recursos.

Como ya se mencionó anteriormente, para desarrollar una aplicación con *CUDA* se suele utilizar un entorno de desarrollo basado en lenguaje C, al cual se le añaden una serie de extensiones específicas de esta tecnología. A continuación vamos a comentar algunas de las más importantes.

En *CUDA* hay una serie de calificativos para las funciones que permiten indicar si será ejecutada por la CPU o por la GPU, y desde donde puede ser llamada:

- __device__: Indica que la función será ejecutada por la tarjeta gráfica, y únicamente puede ser llamada desde la propia tarjeta gráfica.
- __global__: Como ya vimos anteriormente, este calificativo se utiliza para definir un kernel, el cual es una función que será ejecutada por la GPU y únicamente puede ser llamada desde la CPU.

- __host__: Indica que la función será ejecutada por la CPU, y únicamente puede ser llamada desde la propia CPU. Por lo tanto este calificativo se utiliza para definir una función tradicional de C. Es equivalente a no poner ningún calificativo en la función.

También hay calificativos para las variables que indican donde residirá la variable, así como el tiempo de vida de la misma:

- __device__: Se utiliza para declarar una variable que residirá en la tarjeta gráfica. Si no va a acompañado de ningún otro calificativo, la variable residirá en la memoria global, su tiempo de vida será el mismo que el de la aplicación, y será accesible por todos los hilos.
- __constant__: Declara una variable que residirá en el espacio de memoria constante de la tarjeta gráfica. Su tiempo de vida será el mismo que el de la aplicación, y será accesible por todos los hilos.
- __shared__: Declara una variable que residirá en el espacio de memoria compartida de la tarjeta gráfica. Su tiempo de vida será el mismo que el del bloque de hilos al que pertenece, y solo será accesible por los hilos del bloque de hilos al que pertenece.

Hay que mencionar que estos calificativos no están permitidos al declarar variables pertenecientes a estructuras o uniones.

Otro aspecto a tener en cuenta es cómo se lleva a cabo el mantenimiento de memoria en *CUDA*. Para poder procesar los datos en la tarjeta gráfica es necesario que previamente se hayan transferido a ella desde la CPU. Para ello *CUDA* provee una serie de funciones que permiten reservar memoria en la tarjeta gráfica, transferir la información desde la CPU a la GPU y viceversa. A continuación vamos a describir las mas importantes.

- cudaMalloc: Función que permite reservar un bloque de memoria en la tarjeta gráfica. El formato de la función es

  ```
  cudaError_t cudaMalloc( void** devPtr, size_t count );
  ```

  y tiene un funcionamiento igual que su equivalente en C. El bloque de memoria queda reservado en el espacio de memoria global
- cudaFree: Función que permite liberar un bloque de memoria en la tarjeta gráfica. El formato de la función es

  ```
  cudaError_t cudaFree(void* devPtr);
  ```

  y tiene un funcionamiento igual a su equivalente en C
- cudaMemset: Función que permite inicializar un bloque de memoria de la tarjeta gráfica a un determinado valor. El formato de la función es

  ```
  cudaError_t cudaMemset( void* devPtr, int value, size_t count );
  ```

  y tiene un funcionamiento igual a su equivalente en C.
- cudaMemcpy: Función que permite hacer transferencias de bloques de memoria entre la CPU y la GPU. El formato de la función es

```
cudaError_t cudaMemcpy( void* dst, const void* src, size_t count,
enum cudaMemcpyKind kind);
```

y tiene un funcionamiento similar a su equivalente en C, salvo que en este caso se dispone del parámetro 'cudaMemcpyKind kind' que indica como se va a realizar la transferencia. Básicamente se suelen utilizar dos modos:

- cudaMemcpyHostToDevice: Permite transferir un bloque de memoria de la CPU a la GPU.
- cudaMemcpyDeviceToHost: Permite transferir un bloque de memoria de la GPU a la CPU.

Por lo tanto, el esquema seguido en cuanto a mantenimiento de memoria en la tarjeta gráfica en una aplicación paralelizada suele ser el siguiente:

- Reservar un bloque de memoria en el espacio de memoria global de la tarjeta gráfica para poder copiar los datos de entrada. Para ello se hace uso de la función 'cudaMalloc'.
- Transferir los datos de entrada de la memoria del host a la memoria global de la GPU. Para ello se utiliza la función 'cudaMemcpy' en modo 'cudaMemcpyHostToDevice'.
- En función del tipo de aplicación que estemos desarrollando, puede convenir transferir los datos de la memoria global a la memoria compartida de la GPU, ya que allí se realizan las operaciones de una forma mas rápida.
- Una vez que los datos están en la memoria compartida (o en la memoria global), se realizan los cálculos sobre éstos.
- Cuando los datos ya han sido procesados se transfieren de la memoria compartida a la memoria global (si se hizo la operación inversa anteriormente).
- Los datos de salida se pasan de la memoria global de la GPU a la memoria del host. Para ello se hace uso de la función 'cudaMemcpy' en modo 'cudaMemcpyDeviceToHost'.
- Finalmente se libera el bloque de memoria de la tarjeta gráfica. Para ello se hace uso de la función 'cudaFree'.

Este esquema de trabajo se puede ver en la en la Figura 3.9.

Finalmente vamos a comentar un poco sobre el modo de depuración de *CUDA*. La compilación de una aplicación paralelizada mediante CUDA con nvcc puede hacerse de dos formas. El modo normal, donde la aplicación será ejecutada por medio de la CPU y la GPU según se ha explicado anteriormente, y el modo emulación.

El modo emulación simula una tarjeta gráfica por medio de la CPU. Para ello basta con ejecutar el compilador nvcc con la correspondiente opción:

```
nvcc --device-emulation
```

De esta forma la aplicación va a ser ejecutada completamente por la CPU, que será la encargada de simular los multiprocesadores de la tarjeta gráfica. Con el modo emulación no es necesario disponer de una tarjeta gráfica que permita realizar paralelización, ni tener instalados los correspondientes drivers.

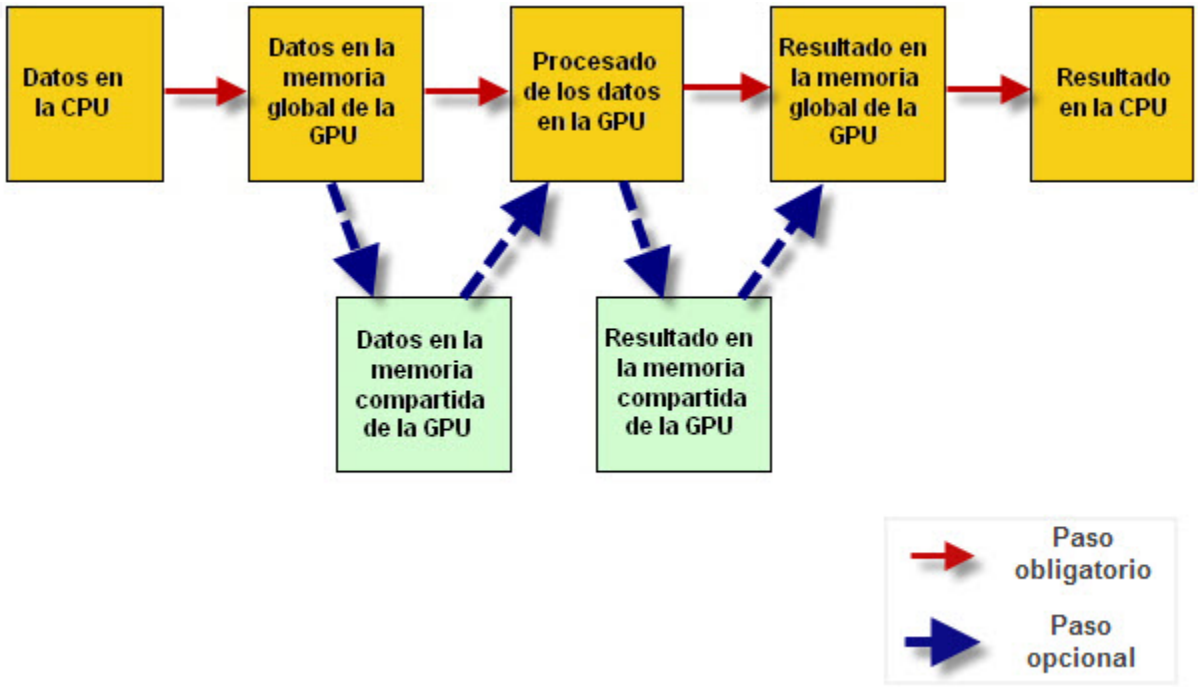

Figura 3.9: Pasos a realizar en una aplicación con CUDA

Por lo tanto, cualquiera puede diseñar una aplicación con la tecnología *CUDA* aunque no disponga de una tarjeta gráfica, pero sin obtener las ventajas que ésta proporciona.

A cambio, se permite realizar una serie de acciones que no están permitidas en la tarjeta gráfica, que van a permitir depurar la aplicación para detectar errores.

Entre estas opciones cabe destacar:

- La aplicación puede ser depurada como cualquier otra aplicación desarrollada en C.
- Se pueden utilizar breakpoints, printf...
- Se puede acceder al valor de cualquier variable perteneciente a la GPU y a la CPU.
- Se pueden detectar fallos en el uso de puntos de sincronización.

Únicamente es aconsejable usar el modo emulación para depurar cuando se producen errores en la ejecución de la aplicación y es necesario acceder al valor de determinadas variables almacenadas en la tarjeta gráfica.

3.4. Modelo de tarjeta gráfica

Para la realización de este proyecto todo el desarrollo se ha llevado a cabo con el modelo de tarjeta gráfica *NVidia GeForce GTX 280*, que presenta las características mostradas en la Tabla 3.1.

Como podemos apreciar, este modelo de tarjeta gráfica permite definir bloques de hilos con un máximo de 512 hilos, así como un máximo de 65535 bloques. Es muy importante no superar

Característica	Valor
Número de multiprocesadores	30
Número de núcleos	240
Memoria global	1073020928 bytes
Memoria constante	65536 bytes
Memoria compartida por bloque	16384 bytes
Número máximo de registros por bloque	16384
Número máximo de hilos por bloque	512
Número máximo de bloques por grid	65535
Frecuencia de reloj	1.3 GHz
Datos en doble precisión	Si

Tabla 3.1: Especificaciones modelo GeForce GTX 280

estos valores durante la ejecución de la aplicación, ya que si no se pueden producir errores o simplemente obtener resultados erróneos.

Otro aspecto a destacar es que este modelo permite trabajar con datos en doble precisión, lo cual es muy importante en un sistema de teledetección, ya que la mayoría de operaciones que se van a realizar requieren una gran precisión en los cálculos.

Finalmente hay que mencionar que el espacio de memoria global disponible es aproximadamente de 1 GB, por lo que hay que procurar no reservar mas memoria en la tarjeta gráfica de la disponible para evitar errores. Esto provoca que todo el procesado de imágenes en un sistema de teledetección mediante paralelización tenga que hacerse por partes, ya que normalmente la cantidad de información que contiene una imagen puede estar próxima a 1GB.

3.5. Optimización de aplicaciones basadas en CUDA

Una vez se ha desarrollado la aplicación conviene intentar optimizarla al máximo para obtener las mejores prestaciones posibles. Algunos consejos útiles para ello son:

- Es aconsejable utilizar un número de hilos por bloque que sea potencia de 2 (128, 256, o 512). Lo normal es usar el mayor número de hilos posible por bloque, para así aprovechar al máximo los recursos disponibles. Por lo tanto normalmente se va a trabajar con bloques de 512 hilos.

- El número de bloques de hilos debe ser como mínimo igual al número de multiprocesadores de la tarjeta, ya que de esta forma todos los núcleos están realizando alguna operación, por lo que se aprovechan al máximo los recursos disponibles. Por otro lado no se debe superar el número máximo de bloques de hilos permitidos, ya que puede provocar errores durante la ejecución de la aplicación.

- Hay que intentar minimizar el paso de información entre la CPU y la GPU, ya que es una operación muy costosa. Por ello conviene transferir todos los datos de la CPU a la GPU durante el inicio de la aplicación. Una vez que los datos están en la GPU ya se realiza todo el procesado, y finalmente, el resultado se transfiere de nuevo a la CPU. Es decir, el paso de información entre la CPU y la GPU solo se debe realizar al comienzo y al final de la aplicación.

- Cuando se necesita que varios hilos compartan datos, estos deben ser puestos en un mismo bloque.
- Un punto a tener en cuenta es que la memoria compartida de cada bloque de hilos tiene un tamaño máximo, el cual no debe excederse para que no se produzcan errores de compilación.
- También hay limitaciones en cuanto al tamaño máximo de la memoria global. Por lo tanto, si la cantidad de información a procesar es muy elevada es conveniente hacer todo el procesado por partes.

3.6. Conclusiones

Una vez se han analizado y descrito todas las características que presenta la tecnología *CUDA*, vamos a comentar algunas de las conclusiones extraídas sobre esta tecnología tras la realización de diversas aplicaciones.

Las diferentes aplicaciones realizadas serán analizadas en capítulos posteriores.

Lo primero que hay que destacar es que para desarrollar aplicaciones con la tecnología *CUDA* no es necesario tener grandes conocimientos previos sobre computación paralela, ya que las nociones básicas asociadas a esta tecnología son bastantes intuitivas de entender. Se puede decir que la iniciación es sencilla y más o menos rápida, y lo que conlleva mayor complejidad es la optimización de los algoritmos.

El desarrollo de las aplicaciones se realiza en un un entorno basado en C, lenguaje de programación muy utilizado por la mayoría de desarrolladores de software, al que se le añaden una serie de extensiones específicas.

Todo el software asociado a la plataforma *CUDA* se puede descargar gratuitamente de la web de Nvidia, por lo que no requiere ningún coste adicional el uso de esta tecnología.

Además existe numerosa documentación sobre diferentes aspectos característicos de *CUDA*, así como foros donde están resueltos la mayoría de problemas que puedan surgir durante el desarrollo de las aplicaciones o donde te pueden ayudar con nuevos problemas.

Hay una gran cantidad de ingenieros de Nvidia dedicados a seguir mejorando las técnicas empleadas en la computación paralela y aplicando su conocimiento al diseño de nuevas técnicas. Por ello se prevee que la tecnología *CUDA* siga creciendo considerablemente en los próximos años.

Una gran ventaja de esta tecnología es que está basada en las tarjetas gráficas de Nvidia, empresa líder en el sector y muy consolidada, que actualmente tiene distribuidas por todo el mundo mas de 50 millones de tarjetas, lo que favorece al crecimiento de *CUDA*.

Otro aspecto a destacar es que el ancho de banda entre la CPU y la GPU es mucho menor que el disponible entre los diferentes espacios de memoria de la tarjeta gráfica y los multiprocesadores incorporados en ella. Por ello conviene hacer el mínimo número de transferencias de memoria entre la CPU y la GPU, y hacerlas únicamente si las operaciones que se van a paralelizar tienen una carga computacional elevada, ya que si no el tiempo empleado para transferir la información entre el host y la GPU será mucho mayor que el empleado por la misma aplicación sin paralelizar, por lo que no obtendremos beneficio al utilizar la tecnología *CUDA*. Es decir, esta tecnología presenta mejoras en los tiempos de procesado cuando el volumen de datos a manejar es muy elevado y la carga computacional de las operaciones a realizar es alta. Estas condiciones las

cumple un sistema de teledetección.

Finalmente hay que mencionar que esta tecnología permite trabajar con datos en doble precisión, aspecto muy importante a la hora de resolver ciertos problemas que lleven asociados una gran carga computacional, ya que estos pueden requerir una alta precisión en los cálculos. Esto facilita la implantación de *CUDA* en diferentes campos de aplicación, donde la precisión en los resultados estará garantizada con esta tecnología.

Capítulo 4

Imágenes en formato TIFF

4.1. Introducción

Un sistema de teledetección suele manejar una gran cantidad de información, lo que implica generar imágenes a la salida de gran tamaño. Normalmente se va a trabajar con imágenes multiespectrales, las cuales contendrán información de diferentes regiones del espectro electromagnético, por lo que será necesario almacenar la información asociada a cada una de las bandas de frecuencia.

En la actualidad existen diferentes formatos para el manejo de imágenes, donde los más utilizados son: JPEG, GIF, TIFF, BMP y PNG.

Para el desarrollo de este proyecto se ha optado por trabajar con imágenes en formato TIFF.

El formato TIFF (Tagged Image File Format) es un formato de imágenes con etiquetas. Esto permite además de almacenar los datos correspondientes a la propia imagen, almacenar información de cualquier otro tipo por medio de etiquetas para su posterior procesado. De esta forma se puede almacenar información sobre las características de la imagen.

Este formato fue desarrollado en 1986 por la empresa Aldus, y actualmente pertenece a Adobe Systems. Una de sus principales características es que permite trabajar con imágenes de gran tamaño (hasta 4 GB), lo cual es un requisito indispensable en todo sistema de teledetección.

A continuación se van a describir algunas de las principales etiquetas que puede incorporar una imagen en formato TIFF.

- TIFFTAG_IMAGEWIDTH: Establece cual es el número de columnas de la imagen. (Número de píxeles por fila).
- TIFFTAG_IMAGELENGTH: Establece cual es el número de filas de la imagen.
- TIFFTAG_SAMPLESPERPIXEL: Determina cual es el número de muestras por píxel. Esta etiqueta es muy importante para el manejo de imágenes multiespectrales, ya que puede establecer cual es el número de bandas de la imagen. Para imágenes en blanco y negro suele valer 1, mientras que para imágenes en color su valor es 3.
- TIFFTAG_BITSPERSAMPLE: Establece el número de bits que se utilizan para codificar cada uno de los píxeles. Puede utilizar 8, 16, 32 o 64 bits por píxel.
- TIFFTAG_PHOTOMETRIC: Determina el espacio de color utilizado para cada dato de la imagen. Entre los valores mas usados están el valor 1 para imágenes en blanco y negro, y el valor 2 para imágenes en color RGB.
- TIFFTAG_SAMPLEFORMAT: Determina el tipo de dato que se ha utilizado para almacenar el valor del píxel. Suele ser de tipo entero sin signo, entero con signo o coma flotante.
- TIFFTAG_COMPRESSION: Determina si se ha utilizado compresión en la imagen, y en ese caso el tipo de compresión utilizada. Un tipo de compresión muy usada es LZW.

Otra característica específica de este formato de imágenes es como almacena internamente la información de la imagen. Básicamente existen dos formas de almacenarla, lo que se conoce como 'stripped' o 'tiled'.

El modo 'stripped' consiste en almacenar toda la información por tiras, donde cada tira comprende un determinado número de filas de la imagen. De esta forma todo el procesado de

la imagen se va haciendo por franjas independientes. Con este modo de almacenamiento de los datos se consigue obtener un rápido acceso aleatorio a los mismos.

Para poder trabajar con imágenes de tipo 'stripped' se dispone de una etiqueta específica:

- TIFFTAG_ROWSPERSTRIP: Indica el número de filas que contiene cada strip.

El otro modo de almacenar la información es el modo 'tiled', que consiste en almacenar toda la información en secciones rectangulares o subimágenes, donde cada una de ellas esta formada por un determinado número de filas y columnas. Cada una de estas subimágenes es tratada de forma independiente.

Para manejar este tipo de imágenes se dispone de las siguientes etiquetas:

- TIFFTAG_TILEWIDTH: Determina el número de columnas en cada 'tile'.
- TIFFTAG_TILELENGTH: Determina el número de filas en cada 'tile'.

Por último cabe mencionar que a partir del formato TIFF se creó el formato de imágenes GeoTIFF. Este formato incluye todas las características mencionadas anteriormente, y además incluye etiquetas específicas que permiten almacenar información de imágenes georeferenciadas, conocidas como 'GeoKeys'.

Con este formato se puede tener información sobre la imagen georeferenciada, como el tipo de proyección, el sistema de coordenadas, la elipsoide, ..., y todo lo necesario para que la imagen pueda ser automáticamente posicionada en un sistema de referencia espacial.

Cabe destacar que este formato es totalmente compatible con el formato TIFF, de forma que cualquier aplicación que sea capaz de manejar imágenes TIFF podrá manejar también este formato de imágenes sin mas que obviar las etiquetas correspondientes a la imagen georeferenciada.

4.2. Aplicación en C para el manejo de imágenes TIFF

Todas las aplicaciones realizadas durante el desarrollo del proyecto están basadas en el manejo de imágenes TIFF. Por ello ha sido necesario la realización de una aplicación que permita trabajar con este formato de imágenes, de forma que se pueda acceder a su contenido.

Para ello se han utilizado las librerías 'libtiff' y 'libgeotiff' para el manejo de imágenes TIFF y GeoTIFF respectivamente. Ambas librerías proveen una serie de herramientas para el manejo de este tipo de imágenes, permitiendo leer y escribir toda la información asociada a la imagen.

Para almacenar toda la información asociada a una imagen TIFF se ha definido una estructura de datos, de forma que en ella se almacenen los campos mas importantes, y sea fácil y rápido el acceso a los mismos. La estructura definida se puede ver en el siguiente fragmento de código:

```
//Estructura que almacena la información básica de la imagen TIFF
struct imagen
{
        double *pixeles;
        uint16_t filas;
```

```
        uint16_t columnas;
        uint16_t bandas;
        uint8_t tiled;
        inf_t inf;
};
```

Como se puede apreciar, la estructura contiene un vector de tipo double con el valor de cada uno de los píxeles de la imagen ordenados por bandas. También se dispone de un campo que indica el número de filas de la imagen, el número de columnas, el número de bandas y el formato en el que está almacenada la información (stripped o tiled). Finalmente se dispone de un último campo que contiene una estructura que almacena información asociada a las etiquetas de la imagen TIFF y GeoTIFF.

El formato de esta estructura se puede ver en el siguiente fragmento de código:

```
//Estructura con los diferentes campos caracteristicos de la
 imagen TIFF
typedef struct inf
{
        //Informacion TIFF
        uint16_t samplesperpixel;
        uint16_t bitsPerSample;
        uint16_t orientation;
        float xres;
        float yres;
        uint16_t resUnit;
        uint32_t tw;
        uint32_t th;
        uint16_t PhotoMetric;
        uint16_t PlanarConfig;
        uint16_t SampleFormat;
        tstrip_t strips;
        tsize_t size;
        uint32_t RowsPerStrip;
        uint16_t Compression;

        //Informacion GTIFF
        struct tag fileTiePoints[6];
        struct tag filePixelScale[3];
        struct tag modelTypeCode;
        struct tag rasterModelType;
        struct tag geoCDTypeCode;
        struct tag geodeticDatumCode;
        struct tag primeMeridiumCode;
        struct tag linearUnitCode;
        struct tag linearUnitValue;
        struct tag angularUnitCode;
        struct tag angularUnitValue;
        struct tag ellipsoideCode;
        struct tag semiMajorAxis;
        struct tag semiMinorAxis;
        struct tag geoInvertFlattening;
        struct tag angularUnitsCode;
        struct tag primeMeridianLongitude;
        struct tag projCSSystemCode;
```

```
        struct tag ProjCode;
        struct tag projCoordTransfCode;
        struct tag linearUnitsCode;
        struct tag linearUnitSize;
        struct tag projStdParallel1;
        struct tag projStdParallel2;
        struct tag projNatOriginLong;
        struct tag projNatOriginLat;
        struct tag projFalseEasting;
        struct tag projFalseNorthing;
        struct tag projFalseOriginLong;
        struct tag projFalseOriginLat;
        struct tag projFalseOriginEasting;
        struct tag projFalseOriginNorthing;
        struct tag projCenterLong;
        struct tag projCenterLat;
        struct tag projCenterEasting;
        struct tag projCenterNorthing;
        struct tag projScaleAtNatOrigin;
        struct tag projScaleAtCenter;
        struct tag projAzimuthAngle;
        struct tag projStraightVertPoleLong;
        struct tag verticalCSType;
        struct tag verticalDatum;
        struct tag verticalUnits;
}inf_t;
```

Como se puede observar se dispone de una serie de campos que contienen toda la información asociada a la imagen TIFF, y otros campos que contienen la información GeoTIFF.

Como la aplicación ha sido diseñada para trabajar tanto con imágenes TIFF como GeoTIFF, lo que se ha hecho es definir cada uno de los campos de la información GeoTIFF como una estructura formada por dos campos. Uno de ellos indica si la correspondiente etiqueta GeoTIFF ha sido definida, y el otro almacena su valor en caso de que así haya sido. El formato de esta estructura se puede ver a continuación:

```
//Estructura para saber si un campo de la imagen GTIFF esta
 definido y en ese caso almacenar su valor
struct tag
{
        uint8_t definida;
        geocode_t valor;
};
```

Por lo tanto, mediante una estructura de tipo 'imagen' se dispone de toda la información necesaria para poder manipular cualquier imagen en formato TIFF y GeoTIFF.

Para poder leer el contenido de una imagen TIFF y almacenarlo en la estructura 'imagen', así como para generar una imagen TIFF a partir de la información almacenada en la estructura, se han definido una serie de algoritmos que implementan toda la funcionalidad necesaria.

```
struct imagen* ReadTIFF(char *fileName);

int8_t WriteTIFF(struct imagen * image, char * output);
```

```
struct imagen* ReadInformationTIFF(TIFF * tif, GTIF * gtif,
 struct imagen * image, uint32_t nImages);

int8_t WriteInformationTIFF(TIFF * tif, GTIF * gtif,
 struct imagen * image, uint32_t banda);
```

El primer método se encarga de implementar un algoritmo que permita acceder al contenido de una imagen TIFF y almacenar toda su información asociada en una estructura del tipo 'imagen'.

Este algoritmo tiene en cuenta si la información asociada a la imagen está almacenada en modo 'stripped' o 'tiled', realizando un procesado diferente en cada caso.

Para el modo 'tiled' la información de la imagen está almacenada en forma de subimágenes, es decir, mediante matrices, donde cada una de ellas está formada por un subconjunto de filas y columnas de la imagen original. En este modo únicamente se trabaja con imágenes con una sola muestra por píxel y una o varias bandas.

Para el modo 'stripped' la información de la imagen está almacenada por tiras, donde cada tira está compuesta por un determinado número de filas de la imagen original. En este caso se trabaja tanto con imágenes con una o varias muestras por píxel, como con imágenes formadas por una o varias bandas.

Cabe destacar que el algoritmo ha sido desarrollado de forma que la aplicación sea lo mas genérica posible en cuanto al tipo de dato de cada uno de los píxeles de la imagen. Se pretende que la aplicación sea capaz de leer el valor de un determinado píxel independientemente del tipo de dato que almacene. Para ello los valores de todos los píxeles de la imagen son almacenados en un vector de tipo double en memoria, para lo cual se realiza una conversión del tipo de dato original a tipo double, ya que este es de mayor precisión. La aplicación puede trabajar con píxeles de los siguientes tipos de datos:

- uint8_t
- uint16_t
- uint32_t
- uint64_t
- int8_t
- int16_t
- int32_t
- int64_t
- float
- double

El segundo método implementa un algoritmo que permite generar una imagen TIFF a partir de la información almacenada en una estructura de tipo 'imagen'. Al igual que el método de lectura, este algoritmo tiene en cuenta el modo en que estaba almacenada la información de la

imagen original. Es decir, sigue procedimientos distintos en función de si la información estaba almacenada en modo 'tiled' o 'stripped'.

Al almacenar el valor de cada píxel en la imagen TIFF se realiza una conversión de tipo double al tipo original de los datos, cuyo valor está almacenado en uno de los campos de la estructura 'imagen'.

Por último se dispone de dos métodos para la lectura y escritura de las etiquetas con información característica de la imagen TIFF. Ambos métodos son llamados durante la lectura y escritura de la imagen respectivamente.

4.3. Conclusiones

El formato de imágenes TIFF es muy utilizado en sistemas de teledetección, ya que permite almacenar una gran cantidad de información, lo cual es un requisito indispensable en este tipo de sistemas. Puede llegar a almacenar hasta 4 GB de datos.

Este formato permite almacenar mucha información sobre las características de la imagen en lo que se conoce como etiquetas. Hay etiquetas específicas para imágenes TIFF y para imágenes GeoTIFF.

Para facilitar el uso de este tipo de imágenes en un sistema de teledetección y poder acceder a su contenido a lo largo de la cadena de procesado, se ha desarrollado una aplicación que permite leer y escribir toda la información asociada a la imagen en una estructura de datos.

Esta aplicación será utilizada en los distintos módulos desarrollados posteriormente.

Capítulo 5

Reconocimiento de terreno

5.1. Introducción

Una de las principales aplicaciones que tiene un sistema de teledetección es realizar un reconocimiento del terreno de la imagen capturada por el satélite, para obtener información sobre la naturaleza del suelo, agua, nieve, nubes, ... Toda esta información puede ser utilizada en diversos campos, como urbanismo, medioambiente, pesca, agricultura, detección de desastres naturales...

Para nuestra línea de investigación se ha optado por trabajar con imágenes proporcionadas por *Landsat* 5, las cuales se pueden descargar gratuitamente de la página web de globis [1].

Estas imágenes son multiespectrales, es decir, contienen información en diferentes bandas del espectro electromagnético. En concreto, este satélite proporciona información de siete bandas, que van desde el visible hasta el infrarrojo medio, como se puede ver en la Tabla 5.1 [15]

Banda	Longitud de onda	G	B	ESSUN	K1	K2
1	0.485 μm	0.765827	-2.29	1983	N/A	N/A
2	0.569 μm	1.448189	-4.29	1796	N/A	N/A
3	0.660 μm	1.043976	-2.21	1536	N/A	N/A
4	0.840 μm	0.876024	-2.39	1031	N/A	N/A
5	1.676 μm	0.120354	-0.49	220.0	N/A	N/A
6	11.435 μm	0.055376	1.18	N/A	607.76	1260.56
7	2.223 μm	0.065551	-0.22	83.44	N/A	N/A

Tabla 5.1: Especificaciones bandas Landsat 5

Las constantes G y B se usan para obtener el valor de radiancia de un determinado píxel, la constante ESSUN para obtener la reflectancia, y las constantes K1 y K2 para obtener el valor de temperatura en kelvins, como se verá mas adelante.

Las bandas 1, 2 y 3 se corresponden con las tres bandas del visible (azul, verde y rojo respectivamente). La banda 4 corresponde al infrarrojo cercano, la banda 5 al infrarrojo medio, la banda 6 es la banda térmica, y finalmente la banda 7 pertenece también al infrarrojo medio. Cabe mencionar que la banda 6 (térmica) tiene la mitad de resolución que el resto de bandas, por lo que es necesario realizar un *resampling* para combinarla con las otras. En la Figura 5.1 se puede ver una imagen capturada por *Landsat* 5, donde se puede observar la subimagen asociada a cada una de las siete bandas espectrales.

El análisis de la información contenida en cada banda espectral para un mismo píxel es lo que nos va a permitir determinar el tipo de terreno que representa. Esto es debido a que cada terreno refleja la energía solar incidente de manera distinta según su naturaleza. Es lo que se denomina "firma espectral" del terreno.

La aplicación desarrollada permite generar diferentes imágenes de salida, en función del tipo de información que queramos obtener:

- Imagen del terreno en color natural.
- Imagen con reconocimiento de terreno.
- Imagen térmica del terreno.

[1] `http://glovis.usgs.gov/`

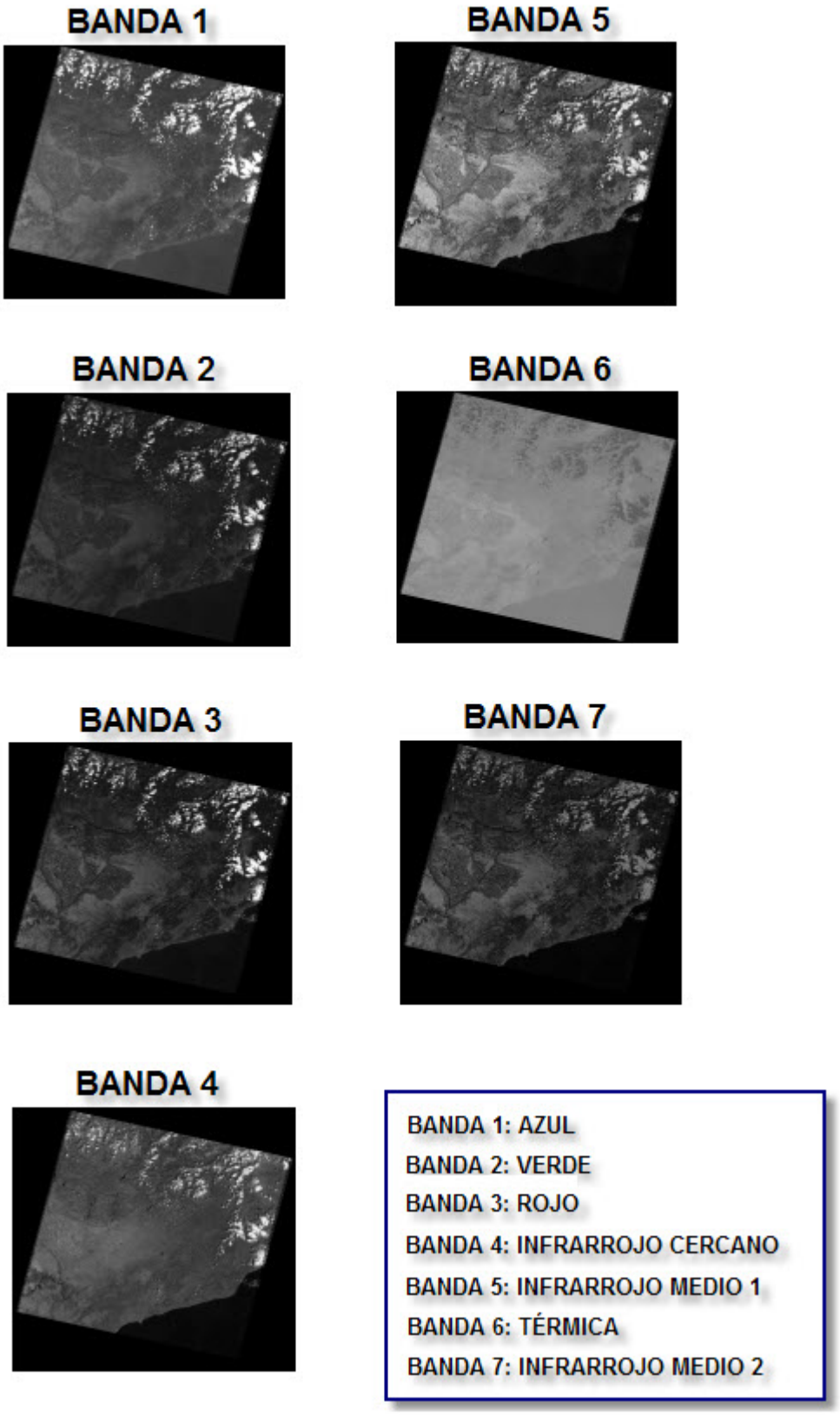

Figura 5.1: Imagen de Landsat 5 formada por siete bandas

- Máscara de nubes.

Para poder determinar el tipo de terreno que representa cada píxel, es necesario hacer un procesado del mismo en cada una de las bandas de trabajo, como se describe a continuación:

- Partir del nivel digital del píxel ("Unidades Digitales").
- Obtener la radiancia. La radiancia de una superficie emisora es el flujo radiante emitido (directamente o por reflexión o transmisión) por unidad de ángulo sólido y por unidad de área proyectada sobre un plano normal a la dirección en consideración. Se mide en vatios partidos por estereorradián por metro cuadrado ($\frac{W}{sr*m^2}$).
- Obtener la reflectancia. La reflectancia es la relación entre la cantidad de energía reflejada por un objeto sobre la incidente. Se mide en %
- Obtener ciertos índices de interés.

El procedimiento que se explica a continuación [16] debe ser aplicado a cada una de las bandas del satélite, con la peculiaridad de la banda térmica.

Inicialmente se parte de los niveles digitales de la imagen, es decir, del valor que tiene cada píxel una vez que es leída la imagen. Estos niveles están codificados como un número de 0 a 255, y están relacionados linealmente con la radiancia (o brillo) captada por el sensor, de acuerdo a la siguiente expresión [16]:

$$radiancia = nivel\_gris * G + B,$$

donde G y B representan la ganancia y el offset respectivamente [15]. Ambos valores dependen de la banda en la que estemos trabajando, como se puede ver en la Tabla 5.1.

Una vez hemos calculado la radiancia, pasamos a calcular la reflectancia. Si la radiancia representa la potencia (por unidad de superficie y ángulo sólido) captada por el sensor, la reflectancia representa el porcentaje de la potencia solar incidente reflejada por la superficie. Para calcular la reflectancia es necesario saber por lo tanto esta potencia solar incidente, que depende de la hora y fecha a la que se tomó la imagen. La expresión que relaciona ambas magnitudes es la siguiente [16]:

$$reflectancia = \frac{\pi * radiancia * d^2}{ESSUN * cos(\theta)},$$

donde d representa la distancia solar y θ el ángulo de elevación (ambos se obtienen del fichero de metadatos, y son constantes en todas las bandas), ESSUN es la constante de radiancia solar (depende de la banda de trabajo [15], como se puede ver en la Tabla 5.1).

Una vez calculada la reflectancia podemos distinguir el tipo de terreno mediante la firma espectral, como se muestra en la Figura 5.2.

Para ello, una primera aproximación podría ser ir comparando el valor de reflectancia medido con el valor de referencia para un determinado tipo de terreno en cada una de las bandas. Sin embargo, comparar el valor medido directamente con la firma espectral puede provocar errores, ya que la reflectancia también depende de otros factores indirectos, como las sombras o la inclinación del terreno.

Para solucionar este problema se definen una serie de índices o ratios [17] que minimizan el efecto de estos factores:

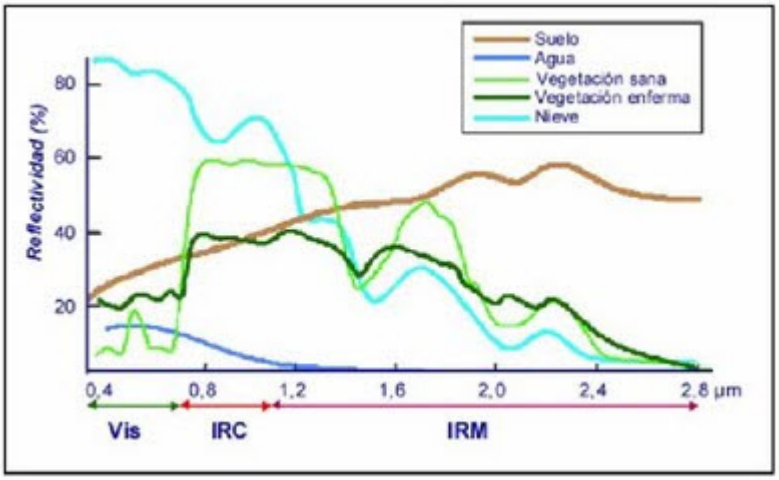

Figura 5.2: Firma espectral. (Fuente: [1])

- Índice de vegetación (ndvi): Representa la relación entre la banda del infrarrojo y la banda visible del rojo. Indica la proporción de vegetación que tiene un determinado píxel, permitiendo detectar su presencia.

$$ndvi = \frac{nir-red}{nir+red}$$

- Índice de humedad (ndwi): Representa la relación entre la banda del infrarrojo y la primera banda del infrarrojo medio. Indica la proporción de humedad que tiene un determinado píxel, permitiendo detectar la presencia de agua.

$$ndwi = \frac{nir-mir1}{nir+mir1}$$

El caso de la banda térmica es especial. En este caso, en vez de trabajar con la reflectancia, se trabaja con la temperatura física del terreno en grados Kelvin, que puede derivarse directamente de la medida de radiancia. Para ello, se utiliza la siguiente expresión [16]:

$$temperatura = \frac{K2}{ln(\frac{K1}{radiancia}+1)},$$

donde K1 y K2 son unas constantes, cuyo valor se puede obtener de la Tabla 5.1.

Además de todos los procedimientos descritos para el reconocimiento de terreno, se ha implementado un algoritmo para la detección de nubes.

Este algoritmo ha sido desarrollado por la *NASA* para detectar la presencia de nubes sobre imágenes capturadas por el satélite *Landsat* [18]. Consiste en ir calculando una serie de relaciones entre las reflectancias de las distintas bandas, y compararlas con los correspondientes umbrales. En función de si los valores obtenidos son mayores o menores que los umbrales, se puede establecer si un determinado píxel tiene o no presencia de nube. Mas concretamente, este algoritmo permite clasificar un píxel como nube, no nube, o ambiguo.

Aplicando todos los procedimientos descritos podemos obtener numerosa información sobre el tipo de terreno que representa la imagen, con sus características particulares.

Además tenemos la posibilidad de obtener series temporales de imágenes que permiten comparar los valores actuales con referencias históricas para evaluar cambios, impactos medioambientales, tendencias y anomalías en la vegetación, que afectan a la superficie terrestre. Las series temporales de datos tienen un gran valor para comprender mejor los cambios y procesos que acontecen en nuestros bosques, pastos y cultivos. Por lo tanto, mediante técnicas de reconocimiento de terreno podemos ser testigos del proceso de transformación y degradación que sufre nuestro planeta, pudiendo mejorar su conservación.

Entre los diferentes campos de aplicación de un sistema de clasificación del tipo de terreno podemos destacar:

- Agricultura de precisión: permite obtener información sobre las necesidades de riego, actividad fotosintética, contenido de biomasa, necesidad de fertilizantes...
- Observación del calentamiento global: permite hacer una evolución de la cobertura de nieve a lo largo de los años como un indicador de la variación de la cantidad de nieve y los efectos provocados por el calentamiento global.
- Detección de incendios: permite obtener información sobre las superficies mas propensas a sufrir un incendio, y en caso de que se produzca, permite obtener detalladamente la superficie de área quemada y así evaluar la severidad de los daños. Para ello se integra el índice de vegetación para calcular la masa vegetal muerta, que define el riesgo de incendio, junto con la temperatura, viento...

5.2. Aplicación desarrollada en C

La aplicación desarrollada se encarga de implementar todos los algoritmos descritos anteriormente, y ejecutarlos en función del tipo de información que queramos generar a la salida.

Como ya se comentó en el apartado anterior, esta aplicación puede generar cuatro tipos de imágenes a la salida a elección del usuario:

- Imagen del terreno en color natural.
- Imagen con reconocimiento de terreno.
- Imagen térmica del terreno.
- Máscara de nubes.

Además el usuario tiene la posibilidad de elegir si generar una sola de estas imágenes o generar las cuatro imágenes a la vez.

Inicialmente se indica la imagen de entrada sobre la que se va a realizar el reconocimiento de terreno, y se procede a leer la información de cada una de las bandas de la imagen. Toda la información leída es almacenada en una estructura de datos para su posterior procesado, para lo cual se hace uso de la correspondiente función de lectura de imágenes TIFF (explicada en su correspondiente capítulo).

Cada imagen de *Landsat* viene acompañada de un fichero de metadatos que contiene información como el instante en el que se capturó la imagen, la posición que tenía el satélite... Como se mencionó anteriormente, esta información es necesaria para realizar el procesado de la

imagen, por lo que se ha definido una función que permite acceder al fichero de metadatos y leer los parámetros necesarios.

Una vez que se ha leído toda la información asociada a la imagen en cada una de las bandas y se tienen los metadatos necesarios, se prosigue con el procesado de la imagen para generar la información de salida.

La imagen que se va a generar a la salida siempre va a ser una imagen en color, por lo que contendrá información de las tres bandas del visible (rojo, azul y verde). Por lo tanto, cada píxel de la imagen de salida tendrá asociado tres muestras, correspondientes a cada una de estas tres bandas.

Como ya se mencionó anteriormente, la banda térmica tiene menor resolución que el resto de bandas, por lo que es necesario realizar un *resampling* para poder llevar a cabo el procesado. Por ello se ha implementado un algoritmo que realiza el duplicado de cada uno de los píxeles de la imagen de entrada, tanto por filas como por columnas, como se puede ver en el siguiente fragmento de código:

```
double *termica;
termica = (double*)malloc(image[0]->filas*image[0]->columnas*
 sizeof(double));
if(termica == NULL)
{
        printf("Error al asignar memoria con malloc\n");
        return (-1);
}

uint32_t aux1=0, aux2=0;

for(i=0; i<image[0]->filas; i++)
{
        aux2 = 0;

        for(j=0; j<image[0]->columnas; j++)
        {
                termica[i*image[0]->columnas+j] = image[5]->pixeles[aux1
                 *image[5]->columnas + aux2];

                if(j%2 == 1)
                        aux2 += 1;
        }

        if(i%2 == 1)
                aux1 += 1;
}
```

Una vez tenemos la información de las siete bandas de la imagen de entrada, todas ellas con la misma resolución, pasamos a realizar el procesado en sí, el cual dependerá del tipo de imagen que queramos generar a la salida.

En todos los casos se debe realizar un bucle que vaya recorriendo cada uno de los píxeles de la imagen y determinando el tipo de terreno que le corresponde a partir de la información asociada en las diferentes bandas.

Para ello se han desarrollado distintas funciones que implementan cada uno de los algoritmos

descritos en el apartado anterior.

Para calcular la radiancia de un determinado píxel en una banda concreta se utiliza el siguiente algoritmo:

```
double Radiancia(double pixel, double ganancia, double offset)
{
        double rad = pixel*ganancia + offset;
        if(rad < 0)
                rad = 0;

        return (rad);
}
```

A partir del valor de radiancia calculado podemos obtener el valor de reflectancia por medio del siguiente algoritmo:

```
double Reflectancia(double rad, double d, double irrad, double ang)
{
        return ((PI*rad*d*d) / (irrad*cos(ang*PI/180)));
}
```

Si estamos trabajando con la banda térmica nos interesará conocer el valor de temperatura en grados Kelvins de un determinado píxel. Para ello usamos el siguiente algoritmo:

```
double Temperatura(double term, const double K1, const double K2)
{
        return (K2 / (log((K1/term)+1)));
}
```

Si lo que queremos es realizar un reconocimiento completo del terreno, es necesario calcular el índice de vegetación y el índice de humedad a partir de los valores de reflectancia. De esta forma minimizamos ciertos errores que puedan aparecer por efectos de sombras o inclinaciones del terreno. Para ello usamos los siguientes algoritmos:

```
double NDVI(double nir, double red)
{
        return ((nir-red) / (nir+red));
}
```

```
double NDWI(double nir, double mir1)
{
        return ((nir-mir1) / (nir+mir1));
}
```

Finalmente tenemos un algoritmo que permite detectar la presencia de nube en un determinado píxel a partir de los valores de reflectancia en cada una de las bandas. Este algoritmo aplica una serie de filtros a partir de las relaciones entre las reflectancias de las diferentes bandas, de forma que permite detectar si un píxel tiene nube, si no tiene nube, o si es ambiguo.

Toda esta funcionalidad se describe en el siguiente fragmento de código:

```
uint8_t DeteccionNubes1(double blue, double green, double red,
 double nir, double mir1, double term, double mir2)
{
        //Umbrales de cada filtro
        double brightness_threshold = 0.08;
        double NDSI_threshold = 0.7;
        uint32_t temperature_threshold = 300;
        uint32_t composite_threshold = 225;
        double filter5_threshold = 2.0;
        double filter6_threshold = 2.0;
        double filter7_threshold = 1.0;
        uint32_t filter8_threshold = 210;

        double pixel_out, NDSI;
        double composite, filter5, filter6, filter7, filter8;

        //Diferenciamos entre nube(2 o 3), no nube(0), y ambiguo(1)
        //Filtro 1: Deteccion de brillo. (Nubes son brillantes)
        if(red < brightness_threshold)
        {
                pixel_out = 0;    //No hay nube
                return pixel_out;
        }

        //Filtro 2: Eliminamos la nieve
        NDSI = (green - mir1)/(green + mir1);
        if(NDSI > NDSI_threshold)
        {
                pixel_out = 0; //No hay nube
                return pixel_out;
        }

        //Filtro 3: Deteccion de temperatura
        if(term > temperature_threshold)
        {
                pixel_out = 0;    //No hay nube
                return pixel_out;
        }

        //Filtro 4: Eliminamos superficies frias con baja reflectancia
        composite = (1 - mir1)*term;
        if(composite > composite_threshold)
        {
                pixel_out = 1;    //Ambiguo
                return pixel_out;
        }

        //Filtro 5: Eliminamos vegetacion con alta reflectancia
         en la banda del rojo
        filter5 = nir / red;
        if(filter5 > filter5_threshold)
        {
                pixel_out = 1;    //Ambiguo
                return pixel_out;
        }
```

```
        //Filtro 6: Eliminamos vegetacion con alta reflectancia
         en la banda del verde
        filter6 = nir / green;
        if(filter6 > filter6_threshold)
        {
                pixel_out = 1;    //Ambiguo
                return pixel_out;
        }

        //Filtro 7: Eliminamos rocas y superficies arenosas
         con alta reflectividad
        filter7 = nir / mir1;
        if(filter7 < filter7_threshold)
        {
                pixel_out = 1;    //Ambiguo
                return pixel_out;
        }

        //Filtro 8: Es nube. Distinguimos entre nube fria y caliente
        filter8 = mir1 / term;

        if(filter8 > filter8_threshold)
                pixel_out = 2;    //Warm cloud
        else
                pixel_out = 3;  //Cold cloud

        return pixel_out;
}
```

Todos estos algoritmos son aplicados en función del tipo de imagen que se vaya a generar a la salida:

- Imagen del terreno en color natural: Se combinan las bandas en el visible.
- Imagen con reconocimiento de terreno: Radiancia + reflectancia + temperatura + ndvi + ndwi + detección de nubes.
- Imagen térmica del terreno: Radiancia + temperatura.
- Máscara de nubes: Radiancia + reflectancia + temperatura + detección de nubes.

Para corregir posibles errores en el procesado de la imagen y evitar valores erróneos en la imagen de salida, se ha diseñado un filtro de mediana de dimensiones 3x3, que permite obtener el valor final de cada píxel a partir de sus píxeles vecinos. De esta forma se evitan incongruencias en píxeles aislados [19]. Este filtro de mediana utiliza el método de la burbuja para ordenar todos los píxeles de la máscara en orden ascendente.

Finalmente se almacena la información de salida en la correspondiente estructura, y se genera la imagen de salida en formato TIFF, para lo cual se llama al correspondiente método de escritura de imágenes TIFF (explicado en su correspondiente capítulo).

La imagen de salida correspondiente a la imagen del terreno en color natural almacena información de las tres bandas del visible (rojo, azul y verde) en sus correspondientes bandas.

La imagen de salida correspondiente a la imagen de reconocimiento de terreno almacena información sobre los siguientes tipos de terreno:

- Nube.
- Muy poca vegetación.
- Poca vegetación.
- Bastante vegetación.
- Mucha vegetación.
- Agua.
- Nieve.
- Suelo muy seco.
- Suelo bastante seco.
- Suelo seco.
- Suelo poco seco.
- Suelo un poco húmedo.

La imagen de salida correspondiente a la imagen térmica almacena información de los siguientes intervalos de temperatura.

- Temperatura inferior a -5 ºC.
- Temperatura comprendida entre -5 ºC y 0ºC.
- Temperatura comprendida entre 0 ºC y 5ºC.
- Temperatura comprendida entre 5 ºC y 10ºC.
- Temperatura comprendida entre 10 ºC y 15ºC.
- Temperatura comprendida entre 15 ºC y 20ºC.
- Temperatura comprendida entre 20 ºC y 25ºC.
- Temperatura comprendida entre 25 ºC y 30ºC.
- Temperatura superior a 30 ºC.

Finalmente, la imagen de salida correspondiente a la máscara de nubes almacena información sobre los siguientes casos:

- No hay nube.
- Sí hay nube.
- Ambiguo.

5.3. Aplicación desarrollada con CUDA

Para mejorar las prestaciones de la aplicación y reducir los tiempos de procesado se ha realizado una paralelización de la misma, de forma que los algoritmos sean ejecutados por varios hilos sobre píxeles distintos de forma simultánea.

Como ya se comentó en el capítulo dedicado a la tecnología *CUDA*, una de las limitaciones que un desarrollador de software tiene a la hora de trabajar con esta tecnología son las restricciones en cuanto al tamaño máximo de memoria.

En un sistema de teledetección generalmente se va a trabajar con imágenes de gran tamaño que además contendrán información en diferentes bandas espectrales. Este es el caso que nos ocupa, donde se trabaja con imágenes multiespectrales que contienen información de siete bandas de frecuencia, por lo que la cantidad de información a manejar es muy grande. Para evitar errores en la tarjeta gráfica por problemas de memoria se ha optado por realizar un procesado por bloques de los datos de entrada. De esta forma, en cada llamada a la función que realiza el procesado mediante paralelización se le pasa un fragmento de la imagen de entrada.

En el siguiente fragmento de código se puede ver la técnica descrita:

```
uint32_t max_memory = 1073020928;  //Memoria global total en bytes
 de la tarjeta grafica

//Trabajamos con datos de tipo double
max_memory = max_memory / sizeof(double);

//Trabajamos con 7 bandas --> Evitamos sobrepasar la memoria maxima
uint32_t pixels_nvidia = max_memory / (nBands+3);
uint32_t pixels_total = 0;

//Hacemos un bucle para procesar los datos mediante paralelizacion
while(pixels_total < image[0]->filas*image[0]->columnas)
{
        if(pixels_total + pixels_nvidia < image[0]->filas*image[0]->
         columnas)
        {
                //Pasamos un fragmento con un total de pixels_nvidia
                 pixels
                ProcesadoImagenesParalelizacion(image[0]->pixeles+
                 pixels_total, image[1]->pixeles+pixels_total,
                 image[2]->pixeles+pixels_total, image[3]->pixeles+
                 pixels_total, image[4]->pixeles+pixels_total,
                 termica+pixels_total, image[6]->pixeles+
                 pixels_total, valores+pixels_total, pixels_nvidia,
                 distancia, datos[2], resp);
        }
        else
        {
                //Pasamos el resto de pixeles
                ProcesadoImagenesParalelizacion(image[0]->pixeles+
                 pixels_total, image[1]->pixeles+pixels_total,
                 image[2]->pixeles+pixels_total, image[3]->
                 pixeles+pixels_total, image[4]->pixeles+
                 pixels_total, termica+pixels_total, image[6]->
                 pixeles+pixels_total, valores+pixels_total,
```

```
                (image[0]->filas*image[0]->columnas -
                pixels_total), distancia, datos[2], resp);
        }
        pixels_total += pixels_nvidia;
}
```

En la función que realiza el procesado mediante paralelización hay que indicar el número de hilos por bloque y el número de bloques que usaremos para procesar todos los datos. En este caso se ha optado por trabajar con 512 hilos por bloque, y un número de bloques proporcional al número de píxeles del fragmento a procesar.

```
//Definimos el numero de hilos
unsigned int num_threads = 512;

...

//Definimos el numero de hilos por bloque y el numero de bloques
dim3 dimBlock(num_threads);
dim3 dimGrid(pixels/num_threads + (pixels%num_threads == 0?0:1));
```

El procedimiento que se sigue para realizar la paralelización de la aplicación se divide en tres partes:

- Transferir todos los datos de entrada de la memoria del host a la memoria global de la tarjeta gráfica.
- Realizar todo el procesado de los datos.
- Transferir los datos de salida de la memoria global de la tarjeta gráfica a la memoria del host.

Para transferir los datos de entrada a la memoria global de la GPU es necesario reservar el correspondiente bloque de memoria y copiar en él los datos de entrada. Este proceso se hace con las correspondientes funciones de mantenimiento de memoria de *CUDA* vistas en el pertinente capítulo.

Una vez que tenemos los datos de entrada en la memoria de la tarjeta gráfica se lleva a cabo el procesado de los mismos sobre la GPU, es decir, mediante paralelización. Para ello se han definido una serie de kernels que implementan cada uno de los algoritmos descritos anteriormente. A continuación se muestra un fragmento de código de uno de los kernels utilizados, y posteriormente se detallan todos los kernels definidos.

```
//Kernel que calcula la radiancia de cada pixel de la imagen en una
 banda especifica
__global__ void Radiancia2(double *banda_d, uint32_t pixels, uint8_t
 banda)
{
        //Definimos los parametros caracteristicos de las imagenes
         capturadas por Landsat 5 TM

        __const__ double G[]={0.765827, 1.448189, 1.043976, 0.876024,
         0.120354, 0.055376, 0.065551};
```

```
        __const__ double B[]={-2.29, -4.29, -2.21, -2.39, -0.49, 1.18,
         -0.22};

        //Indice de bloque
        int bx = blockDim.x * blockIdx.x;

        //Indice de hilo
        int tx = threadIdx.x;

        if(bx+tx < pixels)
        {
                banda_d[bx+tx] = banda_d[bx+tx]*G[banda] + B[banda];
                if(banda_d[bx+tx] < 0.0)
                        banda_d[bx+tx] = 0.0;
        }
        __syncthreads();
}
```

Como se puede apreciar, la forma de acceder a un hilo en concreto es por medio de los índices de bloque y de hilo, los cuales están definidos por las variables:

- threadIdx.x : Identificador de un hilo dentro de un bloque.
- blockIdx.x : Identificador de un bloque de hilos.
- blockDim.x : Tamaño de un bloque de hilos.

También se puede observar que las constantes específicas utilizadas por esta función se han definido en la memoria constante de la tarjeta gráfica mediante el calificativo '__const__'.

El resto de kernels definidos son:

- __global__ void Reflectancia2(double *banda_d, uint32_t pixels, double d, double ang, uint8_t banda);
- __global__ void Temperatura(double *term_d, uint32_t pixels);
- __global__ void Nubes(double *blue_d, double *green_d, double *red_d, double *nir_d, double *mir1_d, double *term_d, double *mir2_d, uint8_t *nubes_d, uint3_t pixels);
- __global__ void Vecinos(uint8_t *aux_d, uint8_t *pixeles_d, uint32_t filas, uint32_t columnas, uint8_t inicio, uint8_t final);
- __global__ void NDVI(double *nir_d, double *red_d, double *ndvi_d, uint32_t pixels);
- __global__ void NDWI(double *nir_d, double *mir1_d, double *ndwi_d, uint32_t pixels);

Finalmente, y tras haber realizado el procesado de los datos, se debe copiar el resultado obtenido a la memoria de la CPU y liberar los bloques de memoria utilizados en la GPU.

De esta forma se consiguen unos resultados óptimos en cuanto a tiempos de ejecución, ya que los resultados obtenidos son los mismos que los proporcionados por la aplicación en C, pero en un tiempo mucho menor.

5.4. Resultados obtenidos y conclusiones

Mediante paralelización se obtienen notables mejoras en cuanto a tiempos de ejecución, como se puede ver en la Tabla 5.2 y en la Tabla 5.3, donde se muestran los tiempos obtenidos al generar distintas imágenes de salida para 2 imágenes diferentes de *Landsat* (se incluye el ahorro relativo obtenido entre el tiempo total sin paralelización y el tiempo total con paralelización, medido en %). Estos tiempos también han sido representados en diferentes gráficas, como se puede ver en la la Figura 5.3 y en la Figura 5.4, donde se pueden ver gráficamente las diferencias obtenidas. Nótese la escala logarítmica de las figuras.

Tipo de ejecución	**Reconocimiento**	**Térmica**	**Nubes**	**Todas**
Sin paralelización	66.557.177	7.223.686	63.802.480	92.367.944
Con paralelización	5.076.326	1.085.745	4.931.752	6.398.222
Manejo de memoria en la GPU	4.662.454	1.082.427	4.521.424	5.618.482
Procesar los datos en la GPU	713	245	618	853
Ahorro relativo (%)	92	85	92	93

Tabla 5.2: Tiempos de ejecución (μs) aplicación de reconocimiento de terreno imagen 1

Tipo de ejecución	**Reconocimiento**	**Térmica**	**Nubes**	**Todas**
Sin paralelización	66.495.431	7.246.260	64.775.408	92.153.128
Con paralelización	5.066.969	1.085.609	4.942.319	6.364.317
Manejo de memoria en la GPU	4.652.590	1.076.202	4.463.391	5.588.919
Procesar los datos en la GPU	724	212	626	862
Ahorro relativo (%)	92	85	92	93

Tabla 5.3: Tiempos de ejecución (μs) aplicación de reconocimiento de terreno imagen 2

Los tiempos mostrados son los obtenidos al ejecutar la aplicación con 2 imágenes de *Landsat* 5 diferentes. En ellos se puede apreciar que los tiempos de ejecución son similares para las dos imágenes, lo cual es razonable ya que ambas tienen un volumen de información similar.

Como se puede apreciar, se consigue un gran ahorro de tiempo de ejecución al aplicar una paralelización sobre la aplicación, ya que el tiempo total con paralelización siempre es menor que el obtenido sin paralelización.

Como ya se comentó en el capitulo dedicado a la tecnología *CUDA*, al realizar la paralelización de la aplicación el mayor tiempo es empleado para transferir los datos entre la memoria de la CPU y la memoria de la GPU. Una vez que los datos ya están en la tarjeta gráfica, el procesado de los mismos se realiza en un tiempo muchísimo menor que el empleado por la misma aplicación sin paralelización. Por lo tanto, si únicamente comparamos el tiempo de ejecución de la aplicación sin paralelización con el tiempo de procesado de los datos en la tarjeta gráfica, vemos que el ahorro obtenido es muy significativo.

Si observamos los tiempos obtenidos para las diferentes imágenes de salida, vemos que el mayor ahorro de tiempo se obtiene al generar todas las imágenes de salida de forma simultánea. Esto es debido a que la transferencia de datos a la GPU se hace una sola vez.

En definitiva, se puede concluir que en vista de los resultados obtenidos la paralelización introduce numerosas ventajas en la aplicación de reconocimiento de terreno para un sistema de teledetección, ya que se reducen muy considerablemente los tiempos de ejecución.

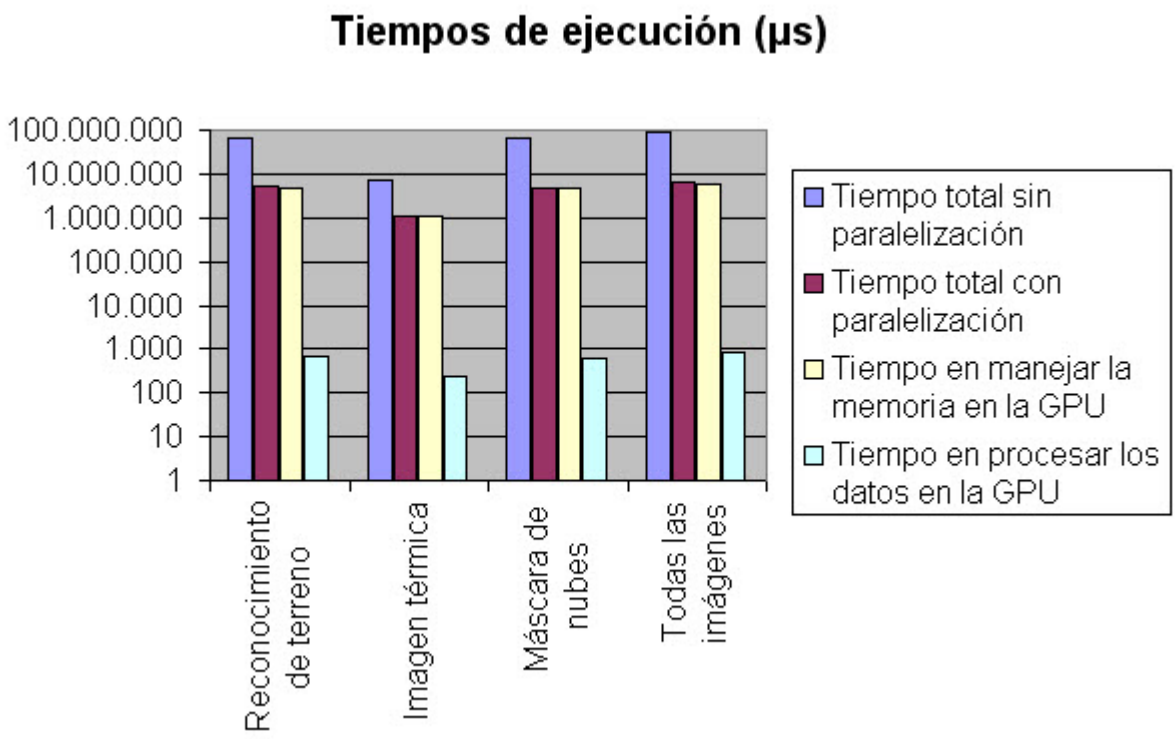

Figura 5.3: Tiempos de ejecución aplicación reconocimiento de terreno imagen1

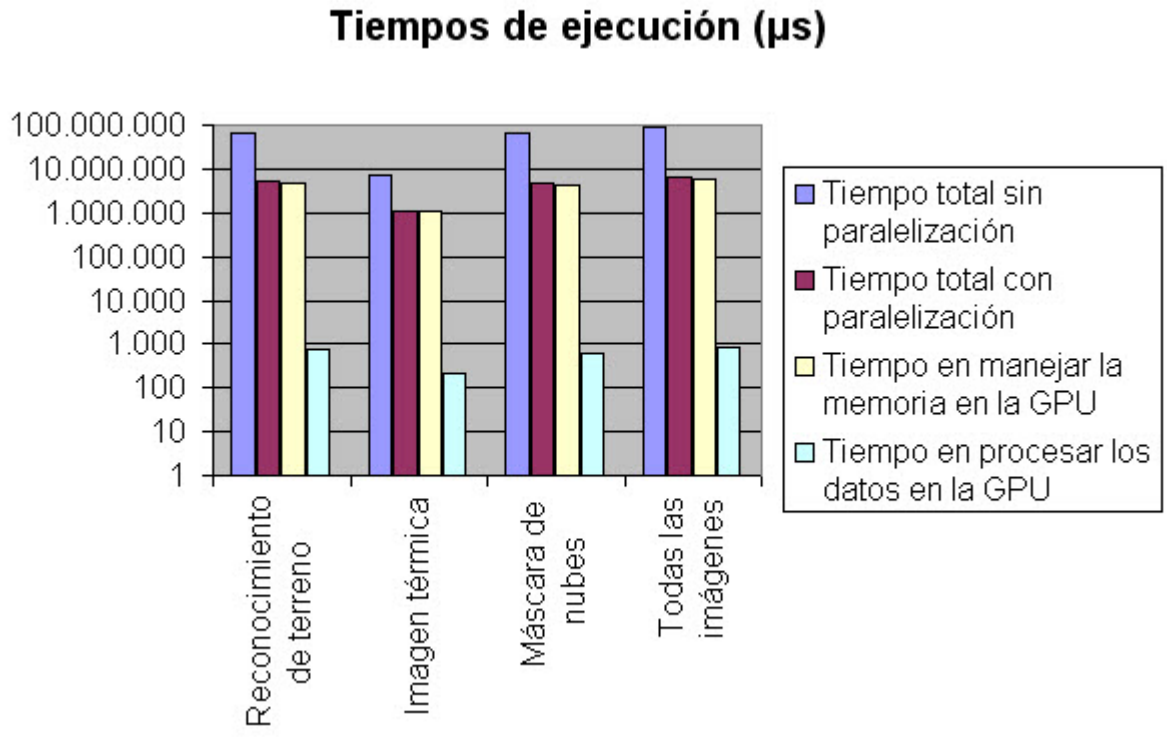

Figura 5.4: Tiempos de ejecución aplicación reconocimiento de terreno imagen2

En la Figura 5.5 y en la Figura 5.6 se muestran algunos ejemplos de los resultados obtenidos con la aplicación, mostrando las diferentes imágenes de salida obtenidas al ejecutar la aplicación sin paralelización y con paralelización.

Las imágenes mostradas se corresponden respectivamente con la imagen del terreno en color natural proporcionada por *Landsat* 5 (obtenida a partir de las tres bandas del espectro visible), la imagen con el reconocimiento de terreno, la imagen térmica (donde los colores fríos corresponden a temperaturas bajas y los colores cálidos a temperaturas altas), y la imagen con la máscara de nubes del terreno.

Como se puede apreciar, las imágenes generadas mediante paralelización y sin paralelización siempre son iguales. (Comprobado mediante comparación de los valores de los píxeles de ambas imágenes).

Finalmente detallamos como debe ser invocada la aplicación para ser ejecutada. Para ello es necesario pasar como argumento en la llamada a la función el nombre de la imagen a procesar. Esta aplicación esta diseñada para trabajar exclusivamente con imágenes de *Landsat* 5, las cuales se pueden descargar gratuitamente de la página web de Globis [2] , como ya se comentó anteriormente. Estas imágenes están compuestas por un total de siete subimágenes, donde cada una corresponde a cada una de las bandas de trabajo de *Landsat*.

Para ejecutar la aplicación es necesario pasar como argumento la imagen correspondiente a la primera banda de trabajo. La aplicación automáticamente se encarga de leer el resto de las bandas.

Un ejemplo de como se realizaría la llamada a la aplicación sería:

```
./procesadoTiff imagen2/L519803103120070813B10.TIF
```

Durante la ejecución el usuario puede elegir si generar una sola imagen de salida o todas a la vez, y si ejecutar la aplicación sin paralelización o con paralelización.

[2] `http://glovis.usgs.gov/`

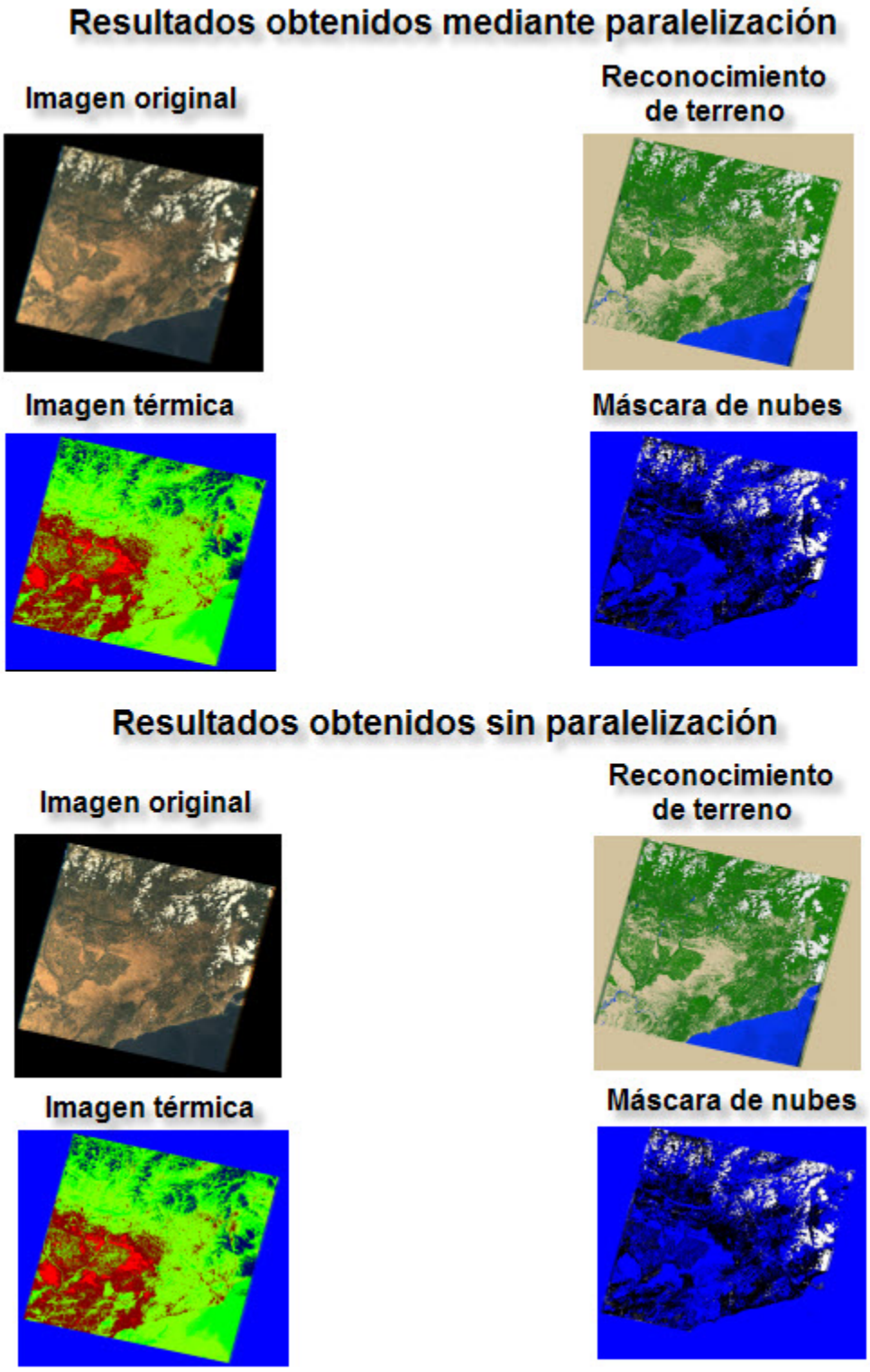

Figura 5.5: Resultados aplicación reconocimiento de terreno imagen1

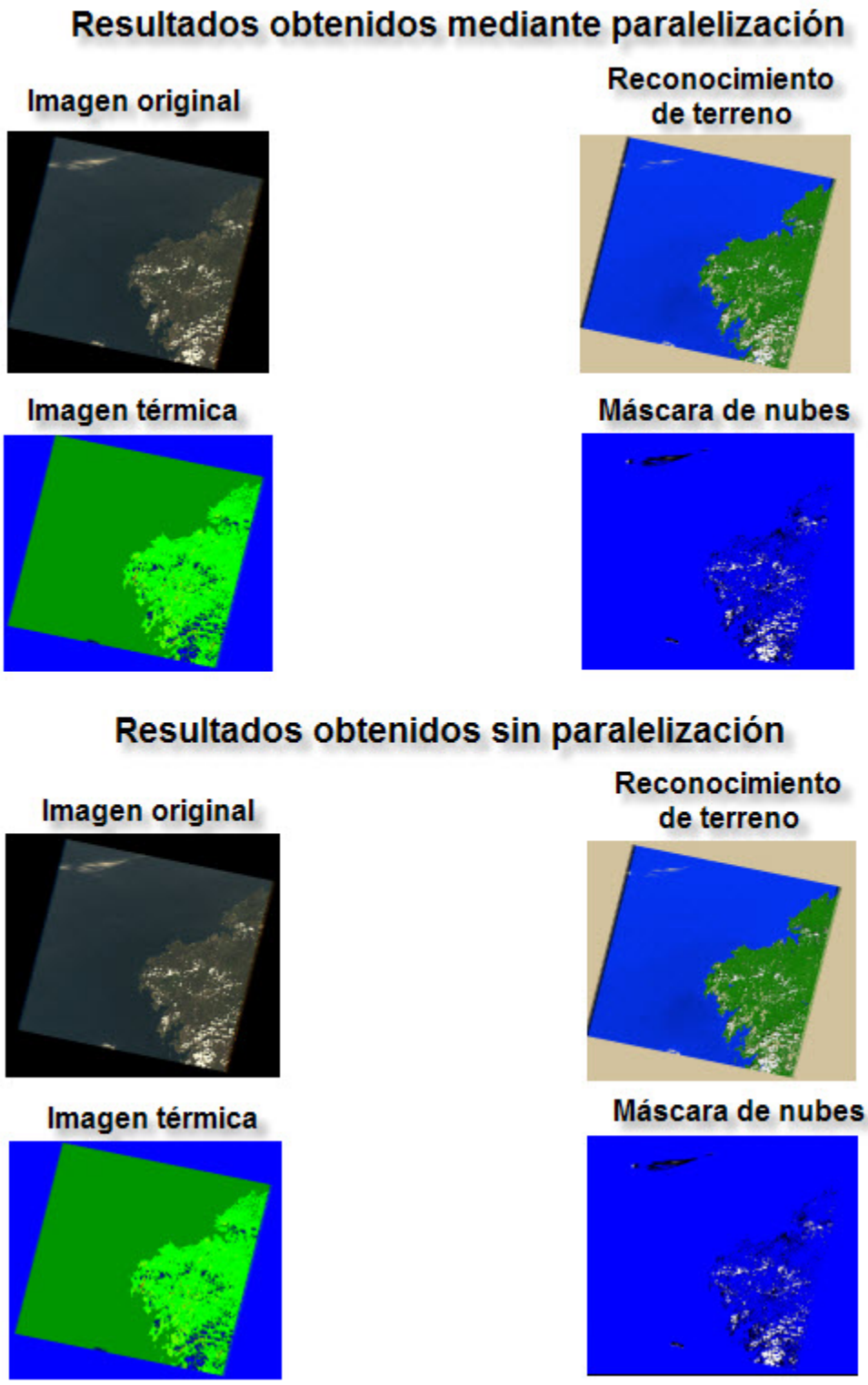

Figura 5.6: Resultados aplicación reconocimiento de terreno imagen2

Capítulo 6

Calibración absoluta

6.1. Introducción

Este módulo tiene como principal objetivo convertir los píxeles de la imagen de unidades digitales a unidades físicas [8]. Este proceso es equivalente a aplicar una ganancia sobre la imagen. Para ello se establece un determinado nivel de ganancia, por el que se va multiplicando cada uno de los píxeles de la imagen, y un nivel de offset, el cual se sumará.

La operación que se realiza es la siguiente:

$$pixel\_out = pixel\_in * ganancia + offset$$

Por lo tanto, a partir de los niveles digitales de la imagen se pueden obtener otras unidades físicas, como por ejemplo radiancias.

Finalmente, y en caso de que sea un requisito de la aplicación, para evitar que un determinado píxel supere el valor máximo establecido por la resolución radiométrica de la imagen, se realiza una saturación de la misma, de forma que el valor de cada píxel quede establecido en un determinado margen.

6.2. Aplicación desarrollada en C

La aplicación desarrollada se encarga de implementar toda la funcionalidad descrita, generando una imagen calibrada a la salida.

Inicialmente la aplicación solicita al usuario los dos parámetros característicos de este módulo:

- Ganancia (Valores comprendidos entre 1 y 10)
- Offset (Valores comprendidos entre 0 y 10)

Posteriormente se procede a la lectura de la imagen de entrada mediante el correspondiente método para lectura de imágenes TIFF (explicado en su correspondiente capítulo). El nombre de la imagen de entrada se debe pasar como argumento en la propia llamada a la aplicación.

Una vez que se tiene toda la información asociada a la imagen se lleva a cabo el procesamiento de la misma. Para ello se genera un bucle que vaya recorriendo cada uno de los píxeles de la imagen, teniendo en cuenta que la imagen de entrada puede ser una imagen multiespectral formada por varias bandas separadas, o por varias muestras consecutivas por píxel.

De esta forma, sobre cada píxel se realiza la calibración y posteriormente una saturación, tal y como se puede ver en el siguiente fragmento de código:

```
if(image->bandas != 1)  //Imagen con varias bandas
{
        //Realizamos la calibracion de la imagen. Tambien se evita la
         saturacion
        for(i=0; i<image->filas*image->columnas*image->bandas; i++)
        {
                image->pixeles[i] = image->pixeles[i]*gain + offset;
                image->pixeles[i] = Saturate(image->pixeles[i],
                 max_value);
        }
```

```
}

else //Imagen con una banda y tal vez con varias muestras por pixel
{
        //Realizamos la calibracion de la imagen. Tambien se evita
         la saturacion
        for(i=0; i<image->filas*image->columnas*image->inf.
         samplesperpixel; i++)
        {
                image->pixeles[i] = image->pixeles[i] * gain + offset;
                image->pixeles[i] = Saturate(image->pixeles[i],
                 max_value);
        }
}
```

El procedimiento para realizar la saturación es el siguiente:

```
double Saturate (double pixel, double max_value)
{
        if (pixel < 0.0)
                pixel = 0.0;
        else if (pixel > max_value)
                pixel = max_value;

        return pixel;
}
```

Como se puede observar se ha optado por realizar una saturación de los píxeles de la imagen de salida una vez que han sido calibrados. En este caso el valor máximo que puede tomar un píxel depende del número de bits con el que se codifica. Por ejemplo, si se utilizan 8 bits para codificar cada píxel, es decir, cada píxel de la imagen de salida será de tipo char, entonces su valor máximo será 255 ($2^8 - 1$). Este parámetro está caracterizado por la variable 'max_value'.

Una vez que han sido procesados todos los píxeles de la imagen se genera la correspondiente imagen de salida. Para ello se usa el método de escritura de imágenes TIFF (explicado en su correspondiente capítulo).

6.3. Aplicación desarrollada con CUDA

Para mejorar las prestaciones de la aplicación se ha llevado a cabo una paralelización de la misma, de forma que todo el procesado se realice en los multiprocesadores de la tarjeta gráfica.

Para realizar las diferentes pruebas se ha utilizado como imagen de entrada la generada por la aplicación de reconocimiento de terreno correspondiente a la imagen del terreno en color natural, obtenida a partir de las bandas del visible. Por lo tanto se va a trabajar con una imagen formada por tres bandas espectrales, lo que supone manejar una gran cantidad de información.

Por esta razón, y tal y como se comentó en el capítulo anterior, se ha optado por realizar el procesado de la imagen por bloques, para evitar así que se generen errores en la tarjeta gráfica por problemas de memoria.

Como la operación a realizar es independiente para cada píxel, no es necesario distinguir entre las diferentes bandas de la imagen, por lo que el procesado de un fragmento de imagen

puede involucrar píxeles de dos bandas diferentes.

El siguiente fragmento de código muestra como se lleva a cabo este procesado de la imagen por bloques en el caso de una imagen formada por diferentes bandas:

```
uint32_t max_memory = 1073020928;  //Memoria global total en bytes
 de la tarjeta grafica

//Trabajamos con datos de tipo double
max_memory = max_memory / sizeof(double);

if(image->bandas != 1)  //Imagen con varias bandas
{
        //Trabajamos con 3 bandas --> Evitamos sobrepasar la memoria
         maxima
        uint32_t pixels_nvidia = max_memory / (image->bandas+3);
        uint32_t pixels_total = 0;

        //Hacemos un bucle para procesar los datos mediante
         paralelizacion
        while(pixels_total < image->filas*image->columnas*image->bandas)
        {
                if(pixels_total + pixels_nvidia < image->filas*
                 image->columnas*image->bandas)
                {
                        //Pasamos un fragmento con un total de
                         pixels_nvidia pixels
                        Calibracion_absoluta(gain, offset,pixels_nvidia,
                         image->pixeles+pixels_total, max_value);
                }
                else
                {
                        //Pasamos el resto de pixeles
                        Calibracion_absoluta(gain, offset, image->filas*
                         image->columnas*image->bandas-pixels_total,
                         image->pixeles+pixels_total, max_value);
                }
                pixels_total += pixels_nvidia;
        }
}
```

Para realizar la paralelización es necesario indicar el número de hilos por bloque y el número de bloques que se utilizarán. En este caso se ha optado por trabajar con 512 hilos por bloque, y un número de bloques proporcional al número de píxeles del fragmento a procesar.

```
//Definimos el numero de hilos
unsigned int num_threads = 512;

...

//Definimos el numero de hilos por bloque y el numero de bloques
dim3 dimBlock(num_threads);
dim3 dimGrid(size/num_threads + (size%num_threads == 0?0:1));
```

Como toda aplicación en *CUDA*, para realizar la paralelización es necesario transferir los

datos de entrada de la memoria del host a la memoria global de la GPU. Para ello es necesario reservar el correspondiente bloque de memoria y copiar en él los datos de entrada.

Una vez que tenemos los datos en la memoria de la tarjeta gráfica ya podemos realizar el procesado de los mismos mediante paralelización. Para ello se ha definido el siguiente kernel:

```
__global__ void Calibracion(double gain, double offset, int size,
 double* imagen_d, double maxvalue)
{
        //Indice de bloque
        int bx = blockDim.x * blockIdx.x;

        //Indice de hilo
        int tx = threadIdx.x;

        if(bx+tx < size)
        {
                //Calibracion
                imagen_d[bx+tx] = imagen_d[bx+tx] * gain + offset;

                //Saturacion
                if (imagen_d[bx+tx] < 0.0)
                        imagen_d[bx+tx] = 0.0;
                else if (imagen_d[bx+tx] > maxvalue)
                        imagen_d[bx+tx] = maxvalue;
        }
}
```

Esta función se encarga de realizar la calibración y saturación de todos los píxeles de forma simultánea, con el consiguiente ahorro en tiempo de procesado.

Como ya se explicó en el capítulo anterior, para acceder a un píxel concreto se utilizan los índices de hilo y de bloque.

Una vez que todos los píxeles del fragmento de imagen han sido procesados, se copia el resultado obtenido en la memoria de la CPU, y se libera el correspondiente bloque de memoria de la GPU.

6.4. Resultados obtenidos y conclusiones

En el caso del código sin paralelizar, la operación es realizada de forma secuencial por la propia CPU, lo que implica que se realice píxel a píxel. Es decir, para realizar la calibración del píxel n previamente se debe haber realizado la del píxel $n-1$.

Por el contrario, en el código paralelizado la operación es realizada de forma simultánea en varios píxeles, lo que conlleva un ahorro de tiempo.

Esta operación tan sencilla es un claro ejemplo del concepto de paralelización, donde en vez de realizarse los cálculos de forma secuencial, estos pasan a ser realizados de forma paralela por diferentes hilos, con el consiguiente ahorro de tiempo durante el procesado.

Los tiempos de ejecución obtenidos en ambos casos se pueden ver en la Tabla 6.1. En la Figura 6.1 se han representado gráficamente.

Como vemos en los tiempos de ejecución obtenidos, el tiempo total con paralelización es

Tiempo total sin paralelización	4.524.435
Tiempo total con paralelización	1.783.729
Tiempo en manejar la memoria en la GPU	1.781.594
Tiempo en procesar los datos en la GPU	293
Ahorro relativo (%)	61

Tabla 6.1: Tiempos de ejecución (μs) módulo de Calibración absoluta

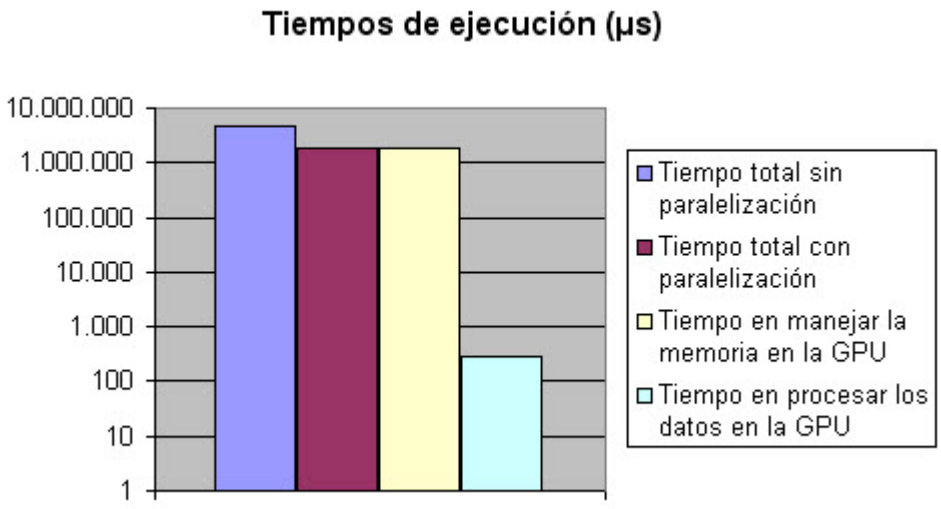

Figura 6.1: Tiempos de ejecución módulo de Calibración Absoluta

menor que el obtenido sin paralelización.

La mayor parte del tiempo de ejecución con paralelización es empleada para transferir los datos entre la memoria del PC y la memoria de la GPU. Si únicamente nos fijamos en el tiempo empleado para procesar estos datos, vemos que el tiempo obtenido es mucho menor que el empleado por la misma aplicación sin paralelización.

Por lo tanto, concluimos que los resultados obtenidos son muy buenos, ya que se obtiene una reducción de tiempo considerable, mas aún si nos fijamos únicamente en los tiempos de procesado.

Las imágenes de salida obtenidas con paralelización y sin paralelización se pueden ver en la Figura 6.2, donde se puede comprobar que ambas son iguales.

Para ejecutar la aplicación es necesario pasar como argumento el nombre de la imagen a procesar, como se puede ver en el siguiente ejemplo:

```
./CalibracionAbsoluta original.tiff
```

Durante la ejecución, y tal y como se comentó anteriormente, la aplicación solicitará una serie de parámetros al usuario:

- Ganancia (Valores comprendidos entre 1 y 10).

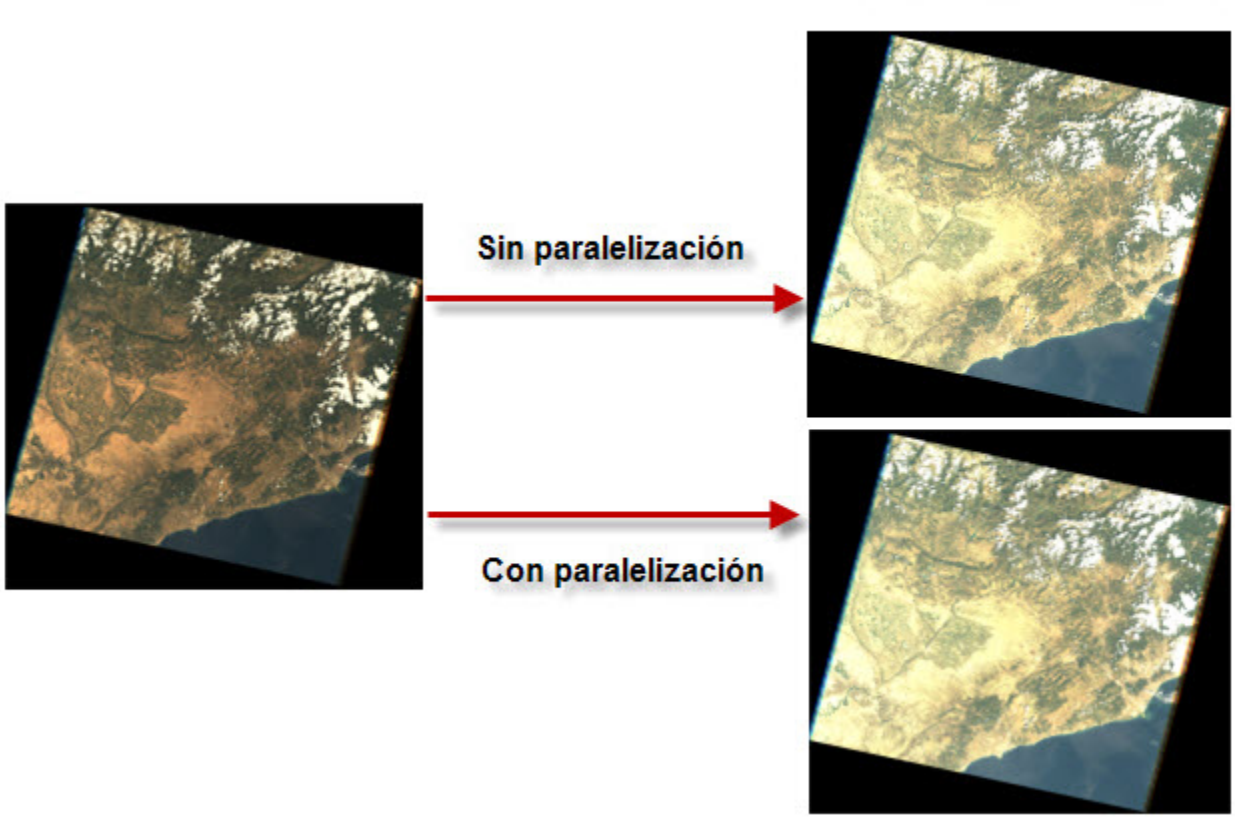

Figura 6.2: Resultados módulo de Calibración Absoluta

- Offset (Valores comprendidos entre 0 y 10).

Finalmente el usuario puede escoger si ejecutar la aplicación mediante paralelización o sin paralelización.

Capítulo 7

Detección de nubes

7.1. Introducción

Este módulo se encarga de detectar la presencia de nubes sobre la imagen capturada por el sensor, y generar la correspondiente máscara de salida para minimizar sus efectos [8] .

La presencia de nubes puede provocar errores en los datos capturados por el satélite, y por lo tanto generar una información de salida errónea.

El algoritmo utilizado por este módulo para detectar las nubes es diferente al que se describió anteriormente en el capítulo de reconocimiento de terreno. En ese caso se describió un algoritmo desarrollado por la NASA, y específico para imágenes de *Landsat*. Ahora vamos a describir el algoritmo utilizado por la cadena de procesado genérica AIPC de Deimos Space.

El procedimiento que se sigue ahora consiste en calcular la reflectancia de un determinado píxel a partir de su radiancia, según la siguiente expresión:

$$reflectancia = \frac{radiancia}{Krad},$$

donde Krad es una constante de radiancia solar que depende de la banda de trabajo.

Una vez calculada la reflectancia se compara con un determinado umbral (que dependerá de la banda de trabajo), de forma que si el valor de reflectancia obtenido está por encima de este umbral es porque hay presencia de nube. En este caso el correspondiente píxel de la máscara de salida se marcaría como tal.

La imagen de entrada es una imagen multiespectral, por lo que tendremos información del terreno en diferentes bandas del espectro electromagnético. Esta imagen debe haber sido calibrada previamente.

Por lo tanto este algoritmo debe ser aplicado sobre cada uno de los píxeles de la imagen, y en cada una de las bandas de trabajo. Es decir, para detectar la presencia de nube en un determinado píxel se deben analizar cada una de las bandas espectrales, y ver si se cumple una cierta condición en cada una de ellas.

7.2. Aplicación desarrollada en C

La aplicación desarrollada se encarga de implementar el algoritmo descrito para generar una máscara de nubes a la salida.

En este caso la aplicación recibe dos parámetros de entrada, correspondientes a la imagen sobre la que se va a detectar la presencia de nubes, y a la máscara de nubes de entrada sobre la que se generará la máscara de salida.

Ambas imágenes son multiespectrales con un total de cuatro bandas, por lo que van a almacenar información en cuatro bandas de frecuencia diferentes.

Inicialmente se definen los parámetros característicos de este módulo. En este caso se corresponden con los umbrales y los valores de radiancia solar de cada una de las bandas de trabajo.

Posteriormente se procede a la lectura de la información almacenada en la imagen y la máscara de entrada. Primero se lee el contenido de la imagen de entrada, y se almacena en su correspondiente estructura de datos para su posterior procesado. Después se procede a la lectura de la máscara de entrada, la cual se modificará en función de las nubes que detecte la aplicación para generar la máscara de salida. En ambos casos la lectura de los datos asociados a cada una de las imágenes se realiza mediante el correspondiente método de lectura de imágenes TIFF.

Una vez se tiene toda la información asociada tanto a la imagen como a la máscara de entrada, se lleva a cabo el procesado de las mismas. Para ello se ha definido un algoritmo que se encarga de ir recorriendo cada uno de los píxeles de la imagen de entrada para detectar la presencia de nube.

Este algoritmo se puede ver en el siguiente fragmento de código:

```
//Detectamos la presencia de nube en cada uno de los pixeles
 de la imagen
for (p=0; p<pixels; p++)
{
        isCloud = true;

        for (l=0; l<nLayers; l++)
        {
                // r(l) = Lo*PI*d^2/Eo*cos(to)
                // solarConstRad is en W/m^2 -> radiances must be in
                 the same units
                double r = (image->pixeles[p + l*image->filas*
                 image->columnas])/solarConstRad[l];

                if ((( r < Thresholds[l]) && ThrIsMinimum[l]) ||
                 (( r > Thresholds[l]) && !ThrIsMinimum[l]))
                {
                        isCloud = isCloud && false;
                        break;
                }
        }

        if (isCloud)
        {
                for(i=0; i<pixelMaskLayers; i++)
                {
                        mascara->pixeles[p + i*mascara->filas*
                         mascara->columnas] = 255;
                        //La nube la marcamos como blanco
                }
        }
}
```

Como se puede observar, el algoritmo consiste en ir recorriendo cada uno de los píxeles de la imagen y procesándolos en las diferentes bandas para detectar la presencia de nube. En el caso de que se supere el umbral establecido en cada una de las bandas, el correspondiente píxel de la máscara de salida se marca como tal. Para ello se establece con el valor 255, es decir, quedará reflejado como un píxel en blanco en la máscara de salida, simulando así la presencia de nube.

Finalmente se genera la correspondiente máscara de salida, para lo cual se hace uso del correspondiente método de escritura de imágenes TIFF.

7.3. Aplicación desarrollada con CUDA

Para mejorar las prestaciones de la aplicación se ha llevado a cabo una paralelización de la misma, de forma que todo el algoritmo pueda ser ejecutado en paralelo por varios hilos, pudiendo

detectar la presencia de nubes en varios píxeles de forma simultánea.

Las imágenes de entrada se corresponden con dos imágenes multiespectrales que contienen información en cuatro bandas de frecuencia. En este caso el tamaño de las imágenes no es excesivamente grande, por lo que todo el procesado en la tarjeta gráfica se puede hacer en una sola vez sin que se generen problemas de memoria. Por lo tanto no es necesario realizar una fragmentación de la imagen de entrada como se hacía en las aplicaciones desarrolladas en los capítulos anteriores.

Para realizar la paralelización es necesario indicar el número de hilos por bloque y el número de bloques que se utilizarán. En este caso se ha optado por trabajar con 512 hilos por bloque, y un número de bloques proporcional al número de píxeles de la imagen de entrada.

```
//Definimos el numero de hilos
unsigned int num_threads = 512;

...

//Definimos el numero de hilos por bloque y el numero de bloques
dim3 dimBlock(num_threads);
dim3 dimGrid(pixels/num_threads + (pixels%num_threads == 0?0:1));
```

Como ya se comentó en capítulos anteriores, para poder realizar el procesado de los datos de entrada en la tarjeta gráfica es necesario transferirlos a la memoria global de la GPU.

Para ello se deben reservar dos bloques de memoria. Uno de ellos que permita almacenar los datos de la imagen de entrada, y otro que permita almacenar los datos de la máscara de entrada, teniendo en cuenta que ambos bloques tienen que ser proporcionales al número de píxeles y al número de bandas, para así poder almacenar toda la información. Una vez se han reservado los correspondientes bloques de memoria se transfieren los datos de entrada de la memoria de la CPU a la memoria global de la GPU.

A continuación se realiza el procesado de estos datos mediante paralelización. Para ello se ha definido un kernel que implementa el algoritmo de detección de nubes descrito anteriormente, como se puede ver en el siguiente fragmento de código:

```
__global__ void Nubes(double *imagen_d, double *mascara_d,int32_t bands,
 int64_t pixels, double* Thresholds, bool* ThrIsMinimum, double*
 solarConstRad)
{
        //Indice de bloque
        int bx = blockDim.x * blockIdx.x;

        //Indice de hilo
        int tx = threadIdx.x;

        //r(l) = Lo*PI*d^2/Eo*cos(to)
        //solarConstRad is en W/m^2 -> reflectances must be in the
         same units

        double r=0;
        bool isCloud = true;

        for (unsigned int i=0; i<bands; i++)
        {
```

```
                if(bx+tx < pixels)
                {
                        r = (imagen_d[bx+tx+pixels*i])/solarConstRad[i];

                        if(((r<Thresholds[i]) && ThrIsMinimum[i]==true)
                         || ((r>Thresholds[i]) && ThrIsMinimum[i]==
                         false))
                        {
                                isCloud = false;
                                break;
                        }
                }
        }

        if (isCloud == true)
        {
                for (unsigned int i=0; i<bands; i++)
                {
                        if(bx+tx < pixels)
                        {
                                mascara_d[bx+tx+pixels*i] = 255;
                                // paint pixel with 255
                        }
                }
        }
}
```

Como se puede observar, el algoritmo implementado es el mismo que en el caso de la aplicación desarrollada en C, con la diferencia de que ahora el procesado de los diferentes píxeles de la imagen se realiza de forma simultánea, con el consiguiente ahorro en el tiempo de ejecución de la aplicación.

Una vez se han procesado todos los píxeles de la imagen, se transfiere el resultado de la memoria global de la GPU a la memoria de la CPU, para así generar la máscara de salida. Finalmente se liberan los bloques de memoria usados en la tarjeta gráfica.

7.4. Resultados obtenidos y conclusiones

En la Tabla 7.1 se muestran los tiempos obtenidos durante la ejecución de la aplicación con y sin paralelización. También se muestran gráficamente en la Figura 7.1.

Tiempo total sin paralelización	507.897
Tiempo total con paralelización	631.214
Tiempo en manejar la memoria en la GPU	629.404
Tiempo en procesar los datos en la GPU	87
Ahorro relativo (%)	-24

Tabla 7.1: Tiempos de ejecución (μs) módulo de Detección de nubes

Como vemos en los tiempos de ejecución obtenidos, el tiempo total con paralelización es mayor que el obtenido sin paralelización, lo que produce que el ahorro relativo sea negativo. Esto es debido a que la cantidad de información a procesar no es excesivamente grande.

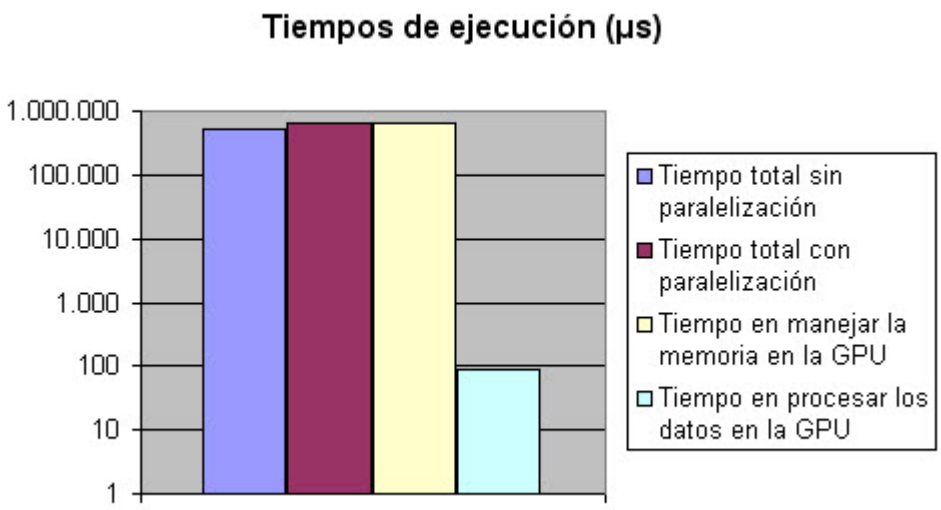

Figura 7.1: Tiempos de ejecución módulo de Detección de nubes

La mayor parte del tiempo de ejecución con paralelización es empleada para transferir los datos entre la memoria del PC y la memoria de la GPU. Si únicamente nos fijamos en el tiempo empleado para procesar estos datos, vemos que el tiempo obtenido es mucho menor que el empleado por la misma aplicación sin paralelización.

En este caso, los tiempos de ejecución obtenidos tanto con paralelización como sin paralelización son menores que en el módulo de Calibración Absoluta, debido principalmente a que se trabaja con imágenes de menor tamaño, lo que implica manejar una menor cantidad de información. Esto provoca que el ahorro de tiempo obtenido en cuanto a tiempos de procesado sea menor que en ese módulo, ya que el tiempo que se tarda en ejecutar la aplicación sin paralelización es bastante menor que en el caso anterior.

En definitiva, concluimos que en cuanto a tiempos de procesado de los datos, los resultados obtenidos son buenos.

En la Figura 7.2 se pueden ver las imágenes de salida obtenidas para cada una de las bandas, tanto con paralelización como sin paralelización. En ambos casos las imágenes generadas son iguales.

Para ejecutar este módulo se deben pasar dos argumentos en la llamada a la aplicación. Primero se debe indicar cual es la imagen a procesar, y segundo cual es la máscara de entrada, como vemos en el siguiente ejemplo:

```
./DeteccionNubes S2TESTIMGABSCAL20081217T1646070000001.tiff mask.tiff
```

Tanto la imagen como la máscara de entrada deben tener el mismo número de bandas y el mismo tamaño, ya que sino se generará un error.

El usuario puede elegir si quiere ejecutar la aplicación con paralelización o sin paralelización.

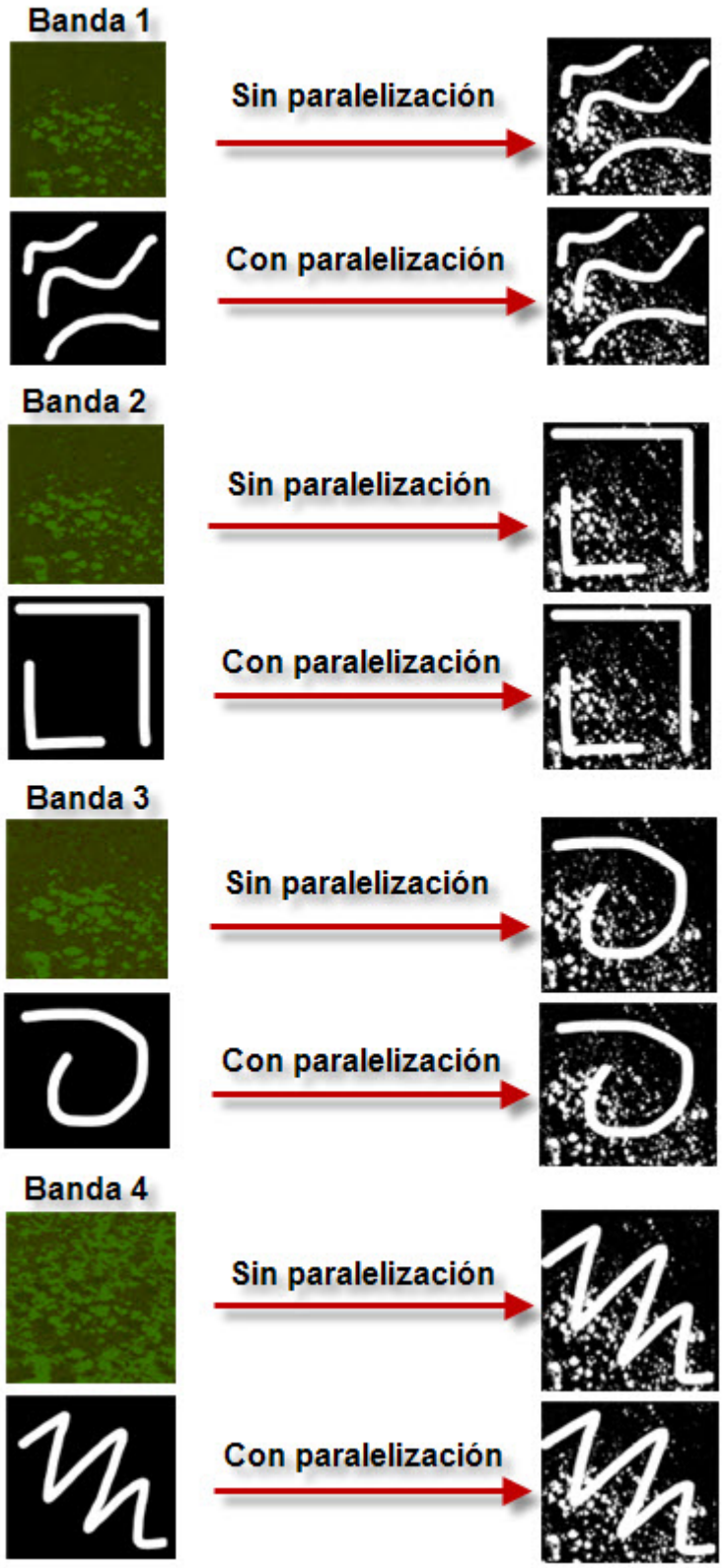

Figura 7.2: Resultados módulo de Detección de nubes

Capítulo 8

Deconvolución

8.1. Introducción

Este módulo permite mejorar la función de transferencia de modulación (MTF) de la imagen.

Para ello se encarga de realizar una convolución espacial en 2 dimensiones entre la imagen de entrada y un filtro. De esta forma se consigue realzar la imagen, obteniéndose una mayor nitidez [8], ya que se eliminan efectos no deseados.

El filtro utilizado para realizar la convolución es un filtro de Wiener, de dimensiones 11x11, cuyos coeficientes han sido preestablecidos. Se ha optado por elegir este tipo de filtro porque es el que produce una mayor relación señal a ruido (SNR) en la señal de entrada. Su función de transferencia es:

$$H[f] = \frac{S[f]^2}{S[f]^2+N[f]^2}$$

La operación que se realiza en este módulo es:

$$y[m,n] = x[m,n] * h[m,n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[i,j]h[m-i,n-j],$$

donde y[m,n] es la imagen de salida, x[m,n] la imagen de entrada, y h[m,n] el filtro de Wiener.

Este módulo es más complejo que los vistos en los dos capítulos anteriores, ya que la operación de convolución requiere una mayor carga computacional.

Para obtener el valor final de cada píxel de la imagen se debe operar con una ventana del tamaño del filtro que englobe a los píxeles vecinos. Por lo tanto, para obtener el valor de salida de un determinado píxel se debe conocer el valor de los píxeles vecinos, por lo que ahora el procedimiento a realizar no es independiente para cada píxel.

8.2. Aplicación desarrollada en C

Para implementar toda esta funcionalidad se ha desarrollado una aplicación en C que sigue los pasos descritos a continuación.

En este módulo el usuario no tiene que definir ningún parámetro característico, ya que los parámetros necesarios para el procesamiento de la imagen de entrada son definidos implícitamente.

Para ello se define un filtro de dimensiones 11x11, cuyos coeficientes corresponden a un filtro de Wiener.

Una vez se ha definido el filtro se procede a la lectura de la imagen de entrada, para lo cual se usa el correspondiente método de lectura de imágenes TIFF. De esta forma se almacena toda la información de la imagen en una estructura de datos. El nombre de la imagen de entrada debe ser pasado como argumento al ejecutar la aplicación.

Cuando se han leído los datos tanto de la imagen de entrada como del filtro se llama al correspondiente método que realizará la convolución espacial entre ambos.

El primer paso para realizar la convolución es comprobar que el número de filas y columnas del filtro sea impar, ya que al ser una convolución espacial en dos dimensiones, para poder calcular la convolución de un determinado píxel es necesario que el filtro este centrado en ese píxel, para lo cual el número de filas y columnas debe ser impar.

Esta aplicación realiza la convolución tanto para imágenes con mas de una banda espectral como con mas de una muestra por píxel. Por lo tanto se han desarrollado dos algoritmos para el manejo de cada tipo de imagen.

Como ya se comentó anteriormente, la convolución de un determinado píxel depende de sus píxeles vecinos. Por ello se ha optado por utilizar la técnica de doble búfer, de forma que no se vayan sobrescribiendo el valor de los píxeles originales una vez se vaya calculando la convolución de cada uno de ellos.

Por lo tanto se van a manejar dos vectores:

- image->pixeles: Contiene los valores de los píxeles de la imagen de salida una vez se ha realizado la convolución.
- buffer: Contiene los valores de los píxeles de la imagen de entrada.

El algoritmo utilizado para realizar la convolución se puede ver en el siguiente fragmento de código:

```
for (m=0; m<image->filas; m++)
{
        for (n=0; n<image->columnas; n++)
        {
                // Perform pixel multiplication and addition
                int64_t firstrow = (m-frows/2) >= 0 ? m-frows/2 : 0;
                int64_t lastrow = (m+frows/2) <= image->filas-1 ?
                 m+frows/2 : image->filas-1;
                int64_t firstcol = (n-fcols/2) >= 0 ? n-fcols/2 : 0;
                int64_t lastcol = (n+fcols/2) <= image->columnas-1 ?
                 n+fcols/2 : image->columnas-1;
                int64_t Lrowdiscard = frows/2-(m-firstrow);
                int64_t Lcoldiscard = fcols/2-(n-firstcol);

                double pixelvalue = 0.0;

                for (i=firstrow; i<=lastrow; i++)
                {
                        for (j=firstcol; j<=lastcol; j++)
                        {
                                pixelvalue += buffer[i*image->columnas+j
                                 + k*image->filas*image->columnas] *
                                 filtro[(i-firstrow+Lrowdiscard)*fcols
                                 + (j-firstcol+Lcoldiscard)];
                        }
                }

                //Evitamos la saturacion
                if (pixelvalue < 0.0)
                        pixelvalue = 0.0;
                else if (pixelvalue > maxvalue)
                        pixelvalue = maxvalue;

                image->pixeles[m*image->columnas+n+k*image->filas*
                 image->columnas] = pixelvalue;
        }
```

```
}
```

Como se puede observar, el algoritmo consiste en ir recorriendo cada uno de los píxeles de la imagen y colocando el filtro centrado en el correspondiente píxel. Para ello se debe calcular el número de filas y columnas a procesar, ya que puede ocurrir que parte del filtro quede fuera de la imagen, por lo que esa parte no debe ser procesada.

Una vez se conocen las filas y columnas a procesar para calcular la convolución de un determinado píxel, se debe ir multiplicando cada píxel de la imagen de entrada con su correspondiente del filtro, y sumándolos.

Finalmente se realiza una saturación para evitar que el valor del píxel de salida supere el valor máximo establecido por la resolución radiométrica.

Este algoritmo debe ser aplicado a cada una de las bandas de la imagen.

En el caso de que la imagen de entrada contenga mas de una muestra por píxel, de forma que las muestras de un mismo píxel sean consecutivas, se ha optado por ordenar todos los píxeles por bandas antes de realizar la convolución. Una vez realizada la convolución, los píxeles de salida se vuelven a colocar de forma que las muestras de un mismo píxel sean consecutivas.

Para ello se usan los siguientes algoritmos respectivamente:

```
//Colocamos todos los pixeles de la imagen por bandas
for(k=0; k<image->inf.samplesperpixel; k++)
{
        for(i=0; i<image->filas*image->columnas; i++)
        {
                buffer[j] = image->pixeles[k+i*image->inf.
                 samplesperpixel];
                j++;
        }
}
```

```
//Ordenamos los pixeles de forma que todas las muestras sean
 consecutivas
for(i=0; i<image->filas*image->columnas; i++)
{
        for(k=0; k<image->inf.samplesperpixel; k++)
        {
                image->pixeles[j]=aux[i+k*image->filas*image->columnas];
                j++;
        }
}
```

Finalmente se genera la imagen de salida. Para ello se hace uso del correspondiente método de escritura de imágenes TIFF.

8.3. Aplicación desarrollada con CUDA

Para mejorar las prestaciones de la aplicación se ha realizado una paralelización de la misma que permite calcular la convolución de cada píxel de la imagen de forma simultánea por medio

de diferentes hilos, con el consiguiente ahorro en tiempo de procesado, ya que la convolución consume bastantes recursos.

Al igual que en el módulo de Calibración absoluta, las diferentes pruebas se han realizado con la imagen del terreno en color natural generada por la aplicación de reconocimiento de terreno, lo que implica manejar una gran cantidad de información. Esta imagen está formada por tres bandas espectrales, lo que implica que se debe calcular la convolución de forma independiente para cada una de ellas.

Como cada banda contiene una gran cantidad de píxeles es necesario realizar un procesado por bloques para evitar errores por problemas de memoria en la tarjeta gráfica.

Para ello se transfiere el primer bloque de información correspondiente a una banda junto con el filtro a la memoria global de la GPU, para llevar a cabo el proceso de convolución. Posteriormente se procesa el segundo bloque de la imagen, a continuación el tercero, y así sucesivamente hasta procesar el bloque final. Este procedimiento se repite para cada una de las bandas de la imagen de entrada.

Hay que tener en cuenta que para procesar un bloque de la imagen que no sea el inicial se deben pasar a la tarjeta gráfica un cierto número de filas procesadas anteriormente, pero que son necesarias para poder procesar las primeras filas del bloque en cuestión. Es decir, en el segundo bloque de imagen, para calcular la convolución de la fila n de la imagen global (1ª de este fragmento) se necesitan las 5 filas anteriores para el caso de un filtro con 11 filas. De forma similar ocurre con las últimas filas del bloque, ya que se necesitan las 5 primeras del bloque siguiente. Este proceso se puede ver gráficamente en la Figura 8.1, y su implementación en el siguiente fragmento de código:

```
uint32_t filas_nvidia = floor((max_memory/image->columnas)/4);

for(i=0; i<image->bandas; i++)
{
        uint32_t filas_nvidia2 = 0;
        uint32_t filas_total = 0;
        uint8_t inicio = 1;
        uint8_t final = 0;

        while(filas_total < image->filas)
        {
                if((filas_total + filas_nvidia < image->filas) &&
                 (inicio == 1))
                {
                        //Primera porcion de imagen
                        ConvolucionCUDA(image->pixeles+i*image->filas*
                         image->columnas, buffer+i*image->filas*
                         image->columnas, filtro, filas_nvidia, image->
                         columnas, frows, fcols, maxvalue, inicio,
                         final, init, finish);

                        inicio = 0;
                        init = 0;
                        filas_nvidia2 = filas_nvidia - frows/2;
                }

                else if((filas_total + filas_nvidia < image->filas)
                 && (inicio == 0))
```

```
                {
                        //Porcion intermedia de imagen
                        ConvolucionCUDA(image->pixeles+filas_total*
                         image->columnas+i*image->filas*image->columnas,
                         buffer+(filas_total-frows/2)*image->columnas+
                         i*image->filas*image->columnas, filtro,
                         filas_nvidia, image->columnas, frows, fcols,
                         maxvalue, inicio, final, init, finish);

                        filas_nvidia2 = filas_nvidia - frows/2 -frows/2;
                }

                else
                {
                        final = 1;

                        if(i == image->bandas-1)
                                finish = 1;

                        if(inicio == 1)
                        {
                                //Imagen que se procesa en una sola vez
                                ConvolucionCUDA(image->pixeles+i*
                                 image->filas*image->columnas, buffer+
                                 i*image->filas*image->columnas, filtro,
                                 image->filas, image->columnas, frows,
                                 fcols, maxvalue, inicio, final, init,
                                 finish);
                        }

                        else
                        {
                                //Porcion final de imagen
                                ConvolucionCUDA(image->pixeles+
                                 filas_total*image->columnas+i*image->
                                 filas*image->columnas, buffer+(filas_
                                 total-frows/2)*image->columnas+i*
                                 image->filas*image->columnas, filtro,
                                 image->filas-filas_total+frows/2,
                                 image->columnas, frows, fcols,maxvalue,
                                 inicio, final, init, finish);
                        }

                        filas_nvidia2 = image->filas - filas_total;
                }

                filas_total += filas_nvidia2;
        }
}
```

Para realizar la paralelización es necesario indicar el número de hilos por bloque y el número de bloques que se utilizarán. En este caso se ha optado por trabajar con 512 hilos por bloque, y un número de bloques proporcional al número de píxeles del fragmento a procesar.

```
//Definimos el numero de hilos
```

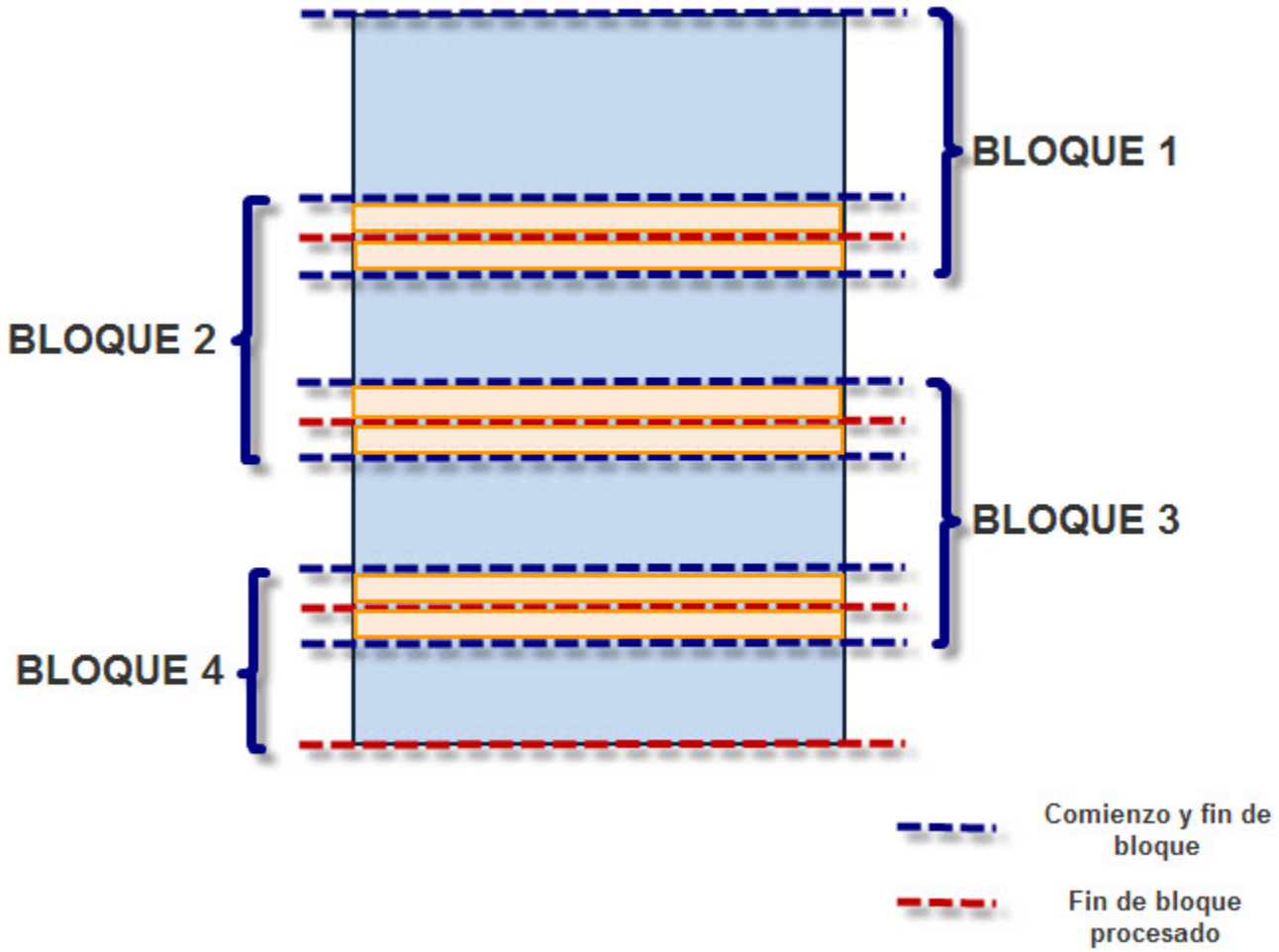

Figura 8.1: Esquema procesado por bloques convolución

```
unsigned int num_threads = 512;

...

//Definimos el numero de hilos por bloque y el numero de bloques
dim3 dimBlock(num_threads);
dim3 dimGrid((nRows*nCols)/num_threads + ((nRows*nCols)%num_threads
 == 0?0:1));
```

Como toda aplicación en *CUDA*, el primer paso es transferir los datos de entrada de la memoria del host a la memoria global de la GPU.

Una vez se han transferido los datos correctamente se ejecuta el correspondiente kernel que realizará la convolución.

```
__global__ void Conv(double *imagen_d, double *imagen_d_final, double *
 filtro_d, int32_t nRows, int32_t nCols, int32_t frows, int32_t fcols,
 double maxvalue, uint8_t inicio)
{
        //Indice de bloque
        int bx = blockDim.x * blockIdx.x;
        //Indice de hilo
        int tx = threadIdx.x;

        if(bx+tx < nRows*nCols)
        {
                int32_t fila = (bx+tx)/nCols;
                int32_t columna = (bx+tx)%nCols;

                int32_t firstrow = (fila-frows/2)>=0 ? fila-frows/2 : 0;
                int32_t lastrow = (fila+frows/2) <= nRows-1 ? fila+
                 frows/2 : nRows-1;
                int32_t firstcol = (columna-fcols/2) >= 0 ? columna-
                 fcols/2 : 0;
                int32_t lastcol = (columna+fcols/2) <= nCols-1 ?
                 columna+fcols/2 : nCols-1;
                int32_t Lrowdiscard = frows/2-(fila-firstrow);
                int32_t Lcoldiscard = fcols/2-(columna-firstcol);

                double pixelvalue=0.0;

                uint32_t i=0;
                uint32_t j=0;
                uint32_t x=0;
                uint32_t y=0;

                for (i=firstrow; i<=lastrow; i++)
                {
                for(j=firstcol; j<=lastcol; j++)
                {
                        x = i*nCols + j;
                        y = (i-firstrow+Lrowdiscard)*fcols + (j-firstcol
                         +Lcoldiscard);

                        if((x < nCols*nRows) && (y < frows*fcols))
                        {
```

```
                                pixelvalue += imagen_d[x] * filtro_d[y];
                        }
                }
                }

                imagen_d_final[fila*nCols + columna] = pixelvalue;

                //Saturacion
                if (imagen_d_final[fila*nCols + columna]<0.0)
                        imagen_d_final[fila*nCols + columna]=0.0;
                else if (imagen_d_final[fila*nCols + columna]>maxvalue)
                        imagen_d_final[fila*nCols + columna]=maxvalue;
        }
}
```

Como se puede observar, el algoritmo es exactamente igual que el desarrollado para la aplicación sin paralelización, con la salvedad de que ahora el algoritmo será ejecutado concurrentemente por varios hilos, de forma que se calculará la convolución de varios píxeles de forma simultánea. Esto supone un gran ahorro en tiempo de procesado.

Una vez calculada la convolución, y en función de la porción de imagen que sea, se copia una determinada cantidad de bytes del resultado obtenido a la memoria de la CPU, y en una determinada posición del puntero que apunta al vector que contiene el resultado.

8.4. Resultados obtenidos y conclusiones

En la Tabla 8.1 se muestran los tiempos obtenidos durante la ejecución de la aplicación con y sin paralelización. También se pueden ver gráficamente en la Figura 8.2.

Tiempo total sin paralelización	293.332.423
Tiempo total con paralelización	13.738.437
Tiempo en manejar la memoria en la GPU	5.795.807
Tiempo en procesar los datos en la GPU	241
Ahorro relativo (%)	95

Tabla 8.1: Tiempos de ejecución (μs) módulo de Deconvolución

Como se puede observar, los resultados obtenidos con paralelización son muy buenos, ya que se obtiene un gran ahorro de tiempo.

En este caso el tiempo total con paralelización es mucho menor que el obtenido sin paralelización. Esto es debido fundamentalmente a que las operaciones que se realizan en este módulo requieren una carga computacional elevada, además de que se trabaja con un gran volumen de datos.

Si además únicamente nos fijamos en los tiempos de procesado, el ahorro que se consigue es excepcional, obteniendo un factor de reducción superior a 1.000.000. Esta comparación (obviando el tiempo de copia de datos) está justificada, porque en una aplicación operacional la copia de datos se realiza una sola vez, al principio y al final de la aplicación, y es independiente del número de algoritmos que encadenemos en el medio.

En definitiva, los resultados obtenidos en este módulo son mucho mejores que en los módulos

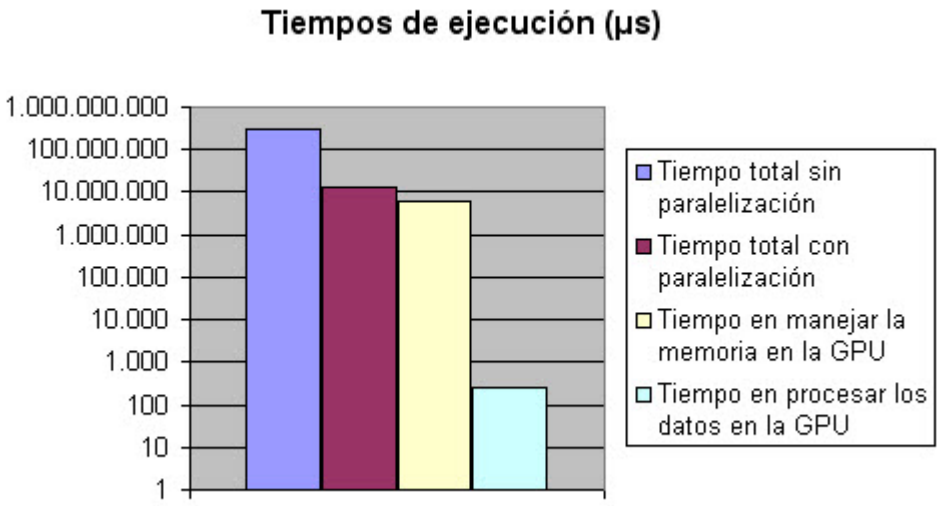

Figura 8.2: Tiempos de ejecución módulo de Deconvolución

anteriores, debido principalmente a que se trabaja con un volumen de datos elevado, y sobre todo a que las operaciones que se realizan requieren una gran carga computacional.

En la Figura 8.3 se pueden ver las imágenes de salida obtenidas.

Para ejecutar esta aplicación se debe indicar cual es la imagen de entrada que se va a procesar. El nombre de la imagen debe ser pasado como argumento en la llamada a la aplicación. A continuación se muestra un ejemplo:

```
./Convolucion original.tiff
```

El usuario no tiene que fijar ningún parámetro, ya que la convolución se realiza con un filtro de Wiener preestablecido de dimensiones 11x11.

Durante la ejecución el usuario puede elegir si ejecutar la aplicación sin paralelización o con paralelización.

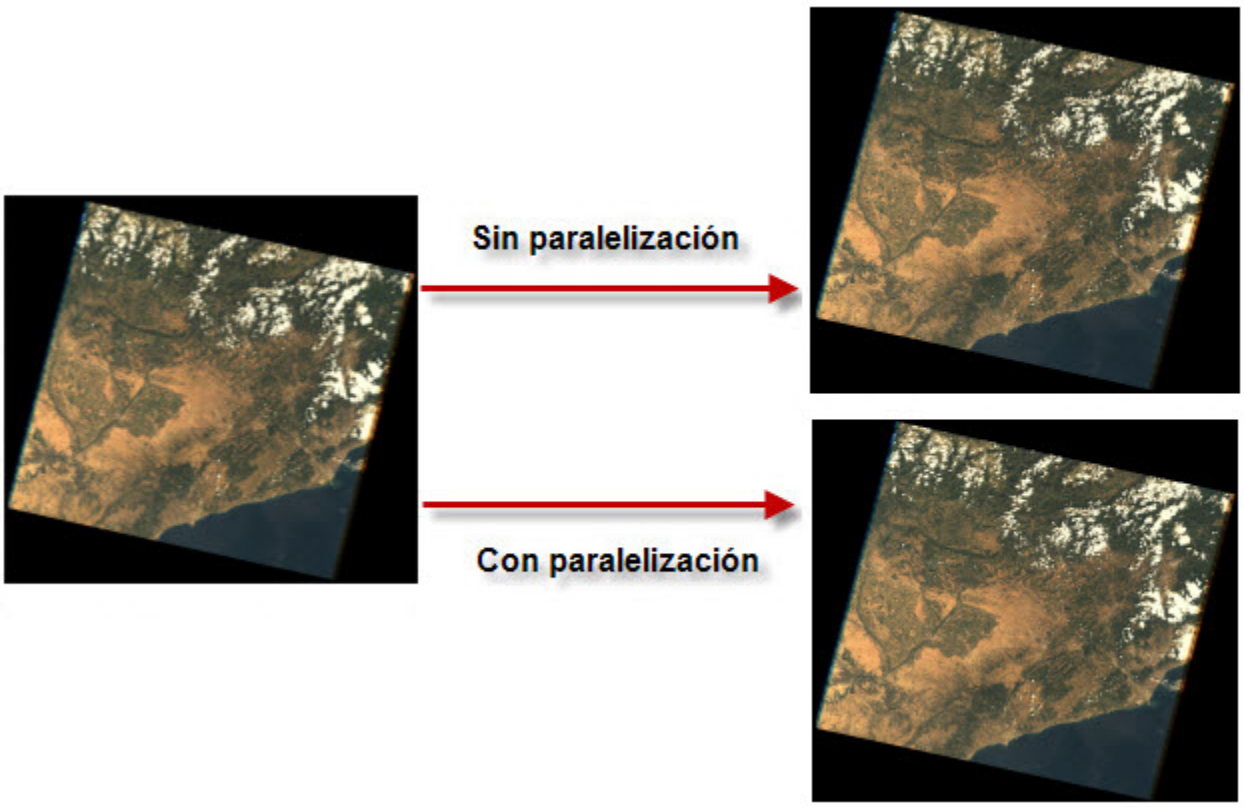

Figura 8.3: Resultados módulo de Deconvolución

Capítulo 9

Reducción de ruido

9.1. Introducción

Este módulo permite reducir el nivel de ruido en la imagen cuando su relación señal a ruido está por debajo de 150.

Para realizar el filtrado de ruido se utiliza la transformada *Wavelet* discreta en dos dimensiones [4], como se puede ver en la Figura 9.1, donde se muestra el esquema seguido.

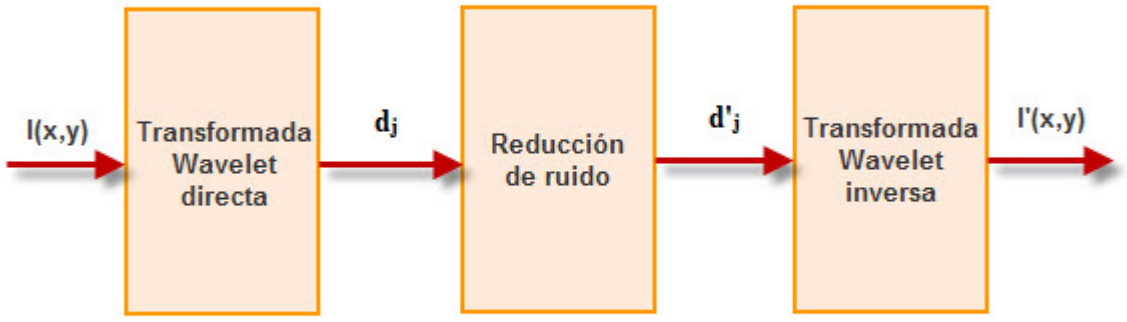

Figura 9.1: Esquema módulo reducción de ruido

Como se puede observar, todo el procesado de la imagen se divide en tres partes:

- Aplicación de la transformada *Wavelet* directa sobre la imagen de entrada.
- Aplicación de un umbral de ruido sobre los coeficientes *Wavelet*.
- Aplicación de la transformada *Wavelet* inversa sobre los coeficientes *Wavelet* modificados.

El primer paso que se realiza consiste en aplicar la transformada *Wavelet* discreta sobre la imagen de entrada.

Esta transformada descompone una señal en una matriz de coeficientes. En el caso de aplicarla sobre una imagen, ésta se descompone en 4 tipos de coeficientes, conocidos como aproximaciones, detalles horizontales, detalles verticales y detalles diagonales, como se puede ver en la Figura 9.2.

La aproximación contiene la mayor parte de la energía de la imagen, es decir, la información más importante, mientras que los detalles tienen valores próximos a cero.

Al aplicar una transformada *Wavelet* se establece un determinado nivel de descomposición, que consiste en ir aplicando la transformada sucesivas veces sobre el coeficiente de aproximación del nivel anterior.

Los niveles más bajos de compresión se corresponden con las bandas de alta frecuencia. En particular, el primer nivel representa la banda de más alta frecuencia y el nivel más fino de

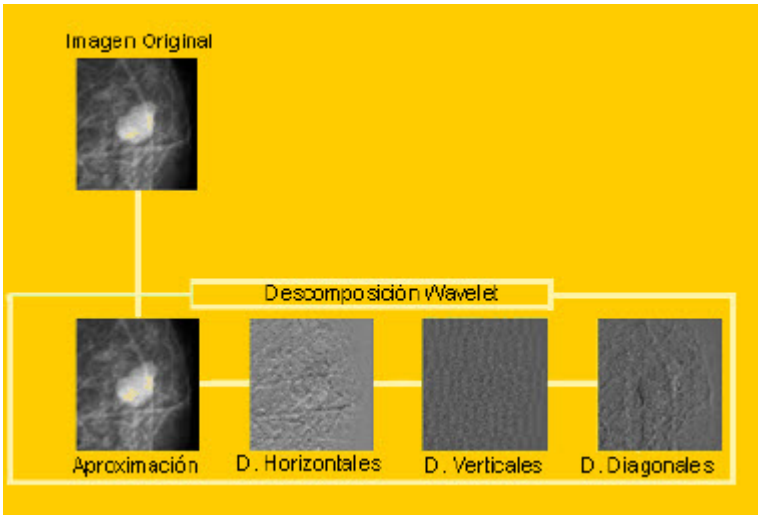

Figura 9.2: Descomposición wavelet (Fuente: [4])

resolución. A la inversa, el último nivel (n) de descomposición corresponde con la banda de frecuencia más baja y la resolución más tosca. Así, al desplazarse de los niveles más altos a los más bajos, o sea, de baja resolución a alta resolución, se observa una disminución de la energía contenida en las subbandas recorridas.

Después de aplicar los diferentes niveles de descomposición, queda una estructura de coeficientes *Wavelet* como la mostrada en la Figura 9.3, donde se ha realizado una descomposición de nivel 3.

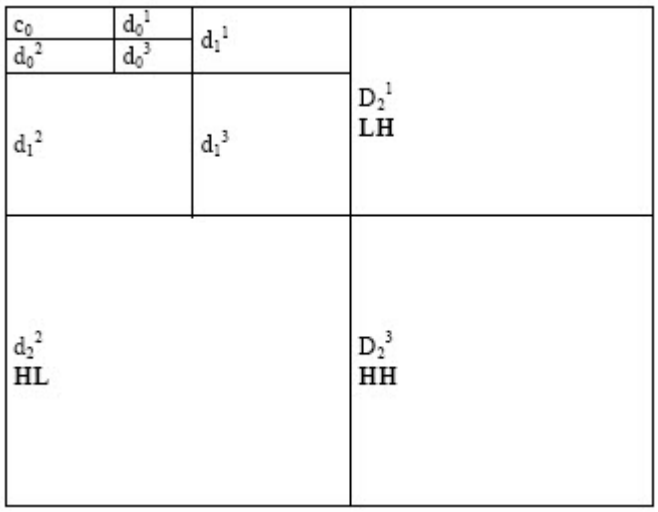

Figura 9.3: Descomposición wavelet de nivel 3

Los diferentes coeficientes *Wavelet* se obtienen mediante el filtrado de la señal de entrada con filtros paso alto y paso bajo. Este filtrado debe ser aplicado primero sobre las filas, y luego sobre las columnas de la imagen. Una vez que la señal ha sido filtrada se realiza un diezmado de orden 2.

Este esquema se puede ver en la Figura 9.4

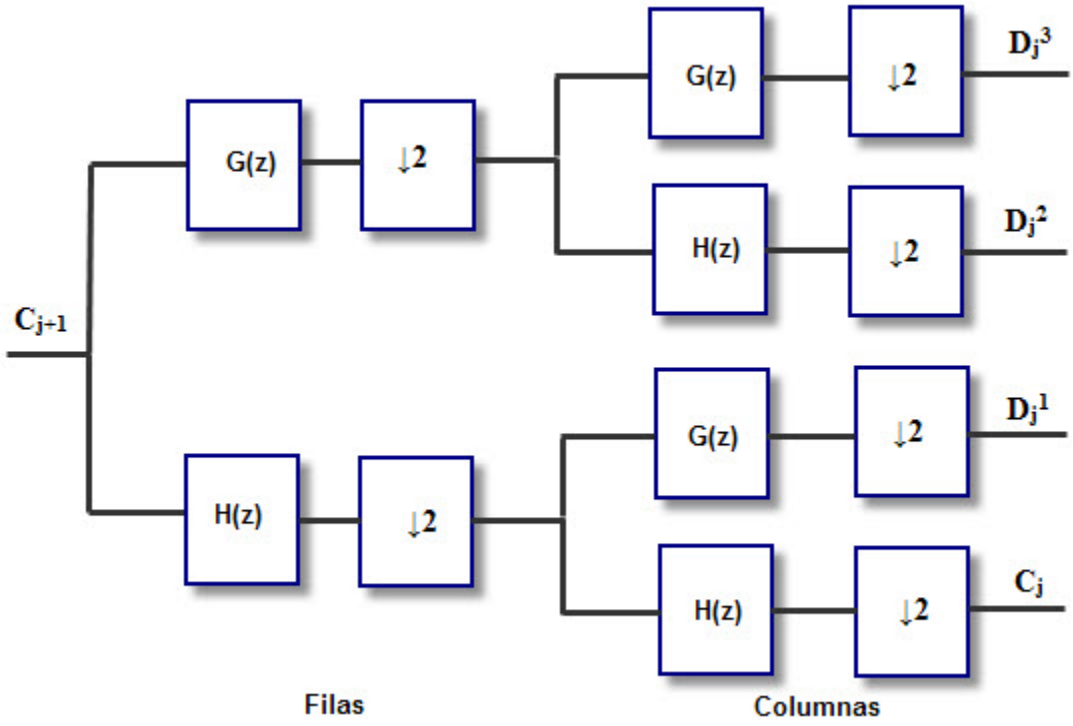

Figura 9.4: Transformada wavelet directa

En este caso G(z) representa un filtro paso alto y H(z) un filtro paso bajo. Cada vez que la señal pasa por un filtro se debe realizar a continuación un diezmado de orden 2. En el caso de haber filtrado por filas, el diezmado se realiza sobre las columnas, mientras que si el filtrado se ha llevado a cabo por columnas el diezmado se realiza por filas.

Como se observa, a partir del coeficiente de aproximación Cj+1 se obtienen los coeficientes del siguiente nivel (los tres coeficientes de detalles y el coeficiente de aproximación).

Una vez se ha obtenido la transformada *Wavelet* de la imagen de entrada y tenemos los coeficientes de los distintos niveles, se aplica la siguiente expresión para reducir el nivel de ruido.

$D'j = 0 \quad , \qquad si \quad |Dj| <= Umbral$

$D'j = Dj - Umbral \quad , \qquad si \quad Dj > Umbral$

$D'j = Dj + Umbral \quad , \qquad si \quad Dj < -Umbral$

, donde Umbral es un valor establecido por el usuario que puede tomar valores comprendidos entre 0 y 100.

Esta expresión se aplica sobre cada uno de los coeficientes *Wavelet*, salvo el coeficiente de aproximación.

Finalmente se debe aplicar la transformada *Wavelet* inversa para reconstruir la imagen de entrada a partir de los nuevos coeficientes *Wavelet*, y obtener así la imagen de salida. El proceso a aplicar es el inverso al realizado para calcular la transformada *Wavelet* directa. Es decir, ahora se utilizan filtros paso alto y paso bajo inversos, y antes de realizar el filtrado se debe realizar una interpolación de orden 2 sobre la señal de entrada. Este esquema se puede ver en la Figura 9.5

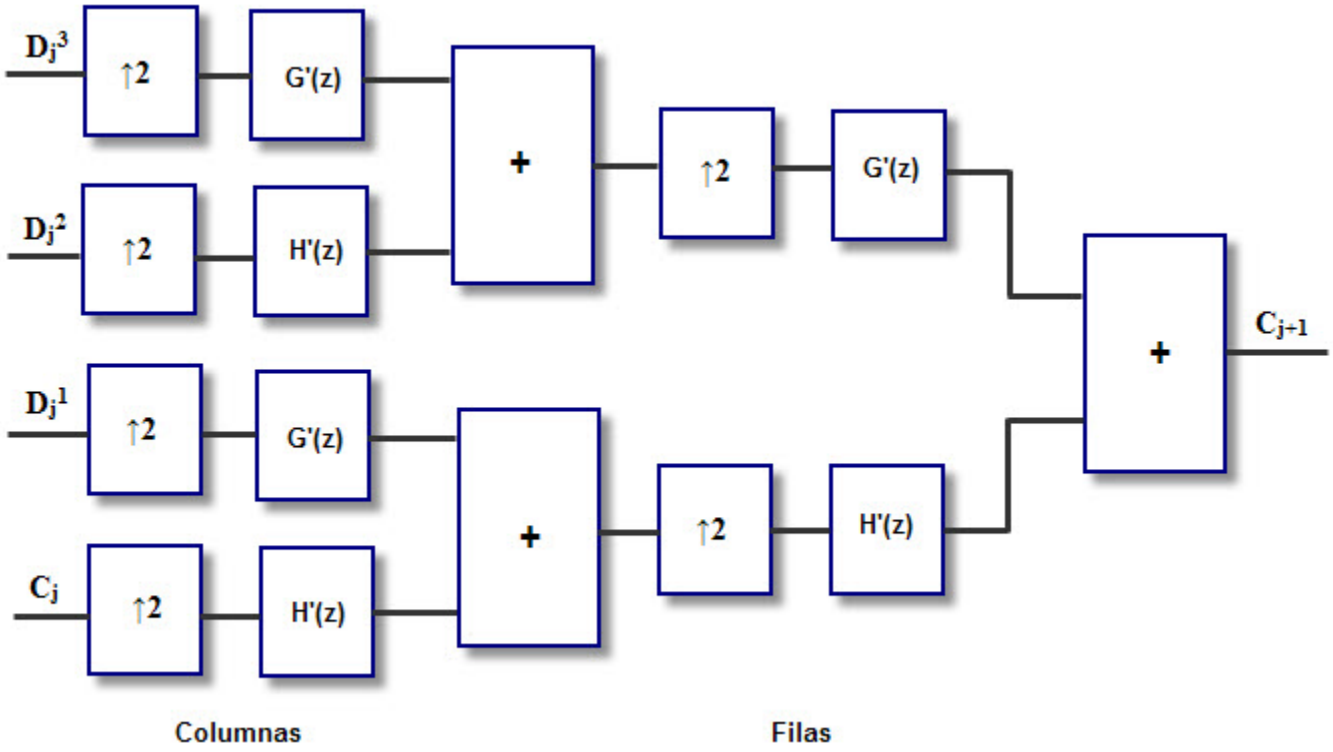

Figura 9.5: Transformada wavelet inversa

9.2. Aplicación desarrollada en C

La aplicación desarrollada se encarga de implementar todos los procedimientos descritos para reducir el nivel de ruido en la imagen de entrada mediante una transformada *Wavelet*.

Inicialmente la aplicación solicita al usuario el valor de tres parámetros característicos de este módulo:

- Tipo de filtro: El usuario puede escoger el tipo de filtro con el que realizar las convoluciones. Puede elegir entre un filtro de tipo Haar, Daubechies, Symmlet y Coiflet.
- Nivel de descomposición: Indica el nivel de descomposición de la transformada *Wavelet*, es decir, el número de veces que se aplicará la tranformada *Wavelet* sobre el coeficiente de aproximación. Puede tomar valores comprendidos entre 1 y 250.
- Umbral de ruido: Indica el umbral de ruido a aplicar sobre los coeficientes *Wavelet*. Puede tomar valores comprendidos entre 0 y 100.

En función del tipo de filtro seleccionado por el usuario, el valor de los coeficientes de los filtros directos e inversos son establecidos implicitamente.

Una vez definidos todos los parámetros característicos de este módulo se procede a la lectura de la imagen de entrada. Para ello se hace uso del correspondiente método de lectura de imágenes TIFF.

Posteriormente se lleva a cabo el procesado de la información de la imagen de entrada para reducir su nivel de ruido.

Para ello es necesario definir una matriz de vectores donde se irán almacenando cada uno de los cuadrantes obtenidos al aplicar la transformada *Wavelet*.

Generalmente la imagen de entrada será una imagen multiespectral, por lo que todo el proceso que se va a explicar debe ser aplicado de forma independiente a cada una de las bandas de la imagen.

El primer paso es calcular la transformada *Wavelet* de la imagen de entrada para cada uno de los niveles de descomposición establecidos por el usuario. De esta forma obtenemos cuatro cuadrantes para cada uno de los niveles.

Para obtener un determinado cuadrante el proceso que se sigue consiste en realizar cuatro pasos:

- Aplicar una convolución por filas sobre el coeficiente de aproximación del nivel anterior, o sobre la imagen de entrada si es el primer nivel.
- Aplicar un diezmado de orden 2 sobre las columnas.
- Aplicar una convolución por columnas.
- Aplicar un diezmado de orden 2 sobre las filas.

Cada cuadrante se obtiene en función del tipo de filtro que se aplique en cada convolución:

- Cuadrante HH: filtro paso alto + filtro paso alto.
- Cuadrante HL: filtro paso alto + filtro paso bajo.
- Cuadrante LH: filtro paso bajo + filtro paso alto.
- Cuadrante LL: filtro paso bajo + filtro paso bajo.

Una vez obtenidos los diferentes cuadrantes se lleva a cabo la reducción de ruido sobre cada uno de ellos, salvo sobre el correspondiente al coeficiente de aproximación, como se puede ver en el siguiente fragmento de código:

```
for(i=nivel_descomposicion; i>0; i--)
{
        cuadrantes[(i-1)*4] = ReduccionRuido(cuadrantes[(i-1)*4],
         filas_cuadrante[i-1], columnas_cuadrante[i-1], threshold);

        cuadrantes[(i-1)*4 + 1] = ReduccionRuido(cuadrantes[(i-1)*4+1],
        filas_cuadrante[i-1], columnas_cuadrante[i-1], threshold);

        cuadrantes[(i-1)*4 + 2] = ReduccionRuido(cuadrantes[(i-1)*4+2],
        filas_cuadrante[i-1], columnas_cuadrante[i-1], threshold);
}
```

Finalmente se calcula la transformada *Wavelet* inversa para reconstruir la imagen. El proceso es el inverso al explicado anteriormente para calcular la transformada *Wavelet* directa.

Ahora se debe ir reconstruyendo el coeficiente de aproximación de un determinado nivel a partir de los cuadrantes del nivel inferior. Para ello se deben ir procesando los cuadrantes de dos en dos, como se describe a continuación:

- Aplicar una interpolación por filas sobre el cuadrante
- Aplicar una convolución por columnas
- Sumar los resultados obtenidos en dos cuadrantes
- Aplicar una interpolación por columnas
- Aplicar una convolución por filas
- Sumar los resultados obtenidos para reconstruir el cuadrante del nivel superior o generar la imagen final

Ahora los filtros que se utilizan para realizar la convolución son los inversos a los utilizados para calcular la transformada *Wavelet* directa.

A continuación mostramos a modo de ejemplo uno de los algoritmos utilizados para realizar todo el proceso descrito. En concreto se muestra el algoritmo usado para obtener un cuadrante al aplicar la transformada *Wavelet*.

```
double* ConstruirCuadrante(double *image, double *FilterCols, double *
 FilterRows, uint32_t rowsImage, uint32_t colsImage, uint32_t
 rowsFilter1, uint32_t colsFilter1, uint32_t rowsFilter2, uint32_t
 colsFilter2, uint32_t *rows, uint32_t *cols)
{
        int64_t i=0, j=0, k=0;

        double *filtro1;
        filtro1 = (double *)malloc(rowsFilter1*colsFilter1*
         sizeof(double));
        if(filtro1 == NULL)
        {
                printf("Error al asignar memoria con malloc\n");
                return (NULL);
        }

        //Invertimos los coeficientes del filtro1 para aplicar la
         convolucion
        for(i=0; i<rowsFilter1*colsFilter1; i++)
                filtro1[i] = FilterCols[rowsFilter1*colsFilter1 -1 - i];

        double *filtro2;
        filtro2 = (double *)malloc(rowsFilter2*colsFilter2*
         sizeof(double));
        if(filtro2 == NULL)
        {
                printf("Error al asignar memoria con malloc\n");
                return (NULL);
        }

        //Invertimos los coeficientes del filtro2 para aplicar la
         convolucion
        for(i=0; i<rowsFilter2*colsFilter2; i++)
                filtro2[i] = FilterRows[rowsFilter2*colsFilter2 -1 - i];

        uint32_t colsResult = floor((colsImage+colsFilter1-1)/2);
```

```
        uint32_t rowsResult = floor((rowsImage+colsFilter1-1)/2);

        //Reservamos memoria para el vector auxiliar
        double* aux = (double *)malloc(rowsImage * colsResult
         * sizeof(double));
        if(aux == NULL)
        {
                printf("Error al asignar memoria con malloc\n");
                return (NULL);
        }

        double* result = (double *)malloc(rowsResult * colsResult
         * sizeof(double));
        if(result == NULL)
        {
                printf("Error al asignar memoria con malloc\n");
                return (NULL);
        }

        // Set the image pointer to the beginning of image
        double *iter = image;
        double *iter2;
        double *iter3;
        double *fiter;
        int rdiscard;
        int ldiscard;

        double pixel_value;

        // Iterate in rows
        for (i=0; i<rowsImage; i++)
        {
                iter2 = iter;

                //Iterate in columns (downsampling by 2)
                for (j=0; j<colsResult; j++)
                {
                        rdiscard = colsFilter1-2*(colsResult-j)+
                         colsImage%2;
                        if (rdiscard < 0)
                                rdiscard=0;

                        ldiscard = colsFilter1-2-2*j;
                        if (ldiscard < 0)
                                ldiscard=0;

                        fiter = filtro1;
                        if (ldiscard > 0)
                                fiter += ldiscard;

                        //Initialize the target pixel value at the
                         convolution beginning
                        pixel_value = 0.0;

                        //Perform the discrete convolution of image and
                         filter
```

```
                        iter3 = iter2;

                        for (k=0; k<colsFilter1-rdiscard-ldiscard; k++)
                        {
                                pixel_value += (*fiter)*(*iter3);
                                fiter++;
                                iter3++;
                        }

                        aux[i*colsResult + j] = pixel_value;

                        //Move forward the image pointer 2 positions to
                         implement downsampling
                        if (ldiscard <= 1)
                                iter2 += 2;
                }

                //Set the pointer to the next column of the first row
                iter += colsImage;
        }

        // Re-apply the convolution filter going now through rows
        // Iterate on columns

        iter = aux;

        for (j=0; j<colsResult; j++)
        {
                iter2 = iter;

                //Iterate on rows (downsampling by 2)
                for (i=0; i<rowsResult; i++)
                {
                        rdiscard = colsFilter2-2*(rowsResult-i)
                         +rowsImage%2;
                        if (rdiscard < 0)
                                rdiscard=0;

                        ldiscard = colsFilter2-2-2*i;
                        if (ldiscard < 0)
                                ldiscard=0;

                        fiter = filtro2;
                        if (ldiscard > 0)
                                fiter += ldiscard;

                        //Initialize the target pixel value at the
                         convolution beginning
                        pixel_value = 0.0;

                        //Perform the discrete convolution of image and
                         filter
                        iter3 = iter2;

                        for (k=0; k<colsFilter2-rdiscard-ldiscard; k++)
                        {
```

```
                                pixel_value += (*fiter)*(*iter3);
                                fiter++;
                                iter3 += colsResult;
                        }

                        result[i*colsResult + j] = pixel_value;

                        //In order to downsample rows, the image pointer
                         is moved down 2 rows per loop
                        if (ldiscard <= 1)
                                iter2 += 2*colsResult;
                }

                // Set the pointer to the next column of the first row
                iter++;
        }

        free(filtro1);
        free(filtro2);
        free(aux);

        *rows = rowsResult;
        *cols = colsResult;

        return result;
}
```

Una vez realizado todo el procesado de la imagen se genera la imagen final. Para ello se usa el correspondiente método de escritura de imágenes TIFF.

9.3. Aplicación desarrollada con CUDA

Al igual que en todos los módulos explicados anteriormente se ha realizado una paralelización de la aplicación que permite mejorar considerablemente las prestaciones de la misma, reduciendo los tiempos de ejecución.

En este caso utilizamos como imagen de entrada la generada por la aplicación de reconocimiento de terreno correspondiente a la imagen del terreno en color natural. Se trata de una imagen multiespectral que contiene una gran cantidad de información, por lo que es necesario realizar un procesado por bloques en la tarjeta gráfica para evitar problemas de memoria.

Para ello se ha optado por realizar la paralelización de la aplicación en dos pasos. Primero se calcula la transformada *Wavelet* directa y se aplica la corrección de ruido sobre cada uno de los coeficientes *Wavelet* obtenidos. Una vez se han procesado todos los píxeles pertenecientes a una banda espectral se calcula la transformada *Wavelet* inversa para reconstruir la imagen original.

En ambos casos el procesado de la información se realiza por bloques, de forma que se le van pasando a la GPU fragmentos de información compuestos por un determinado número de filas. A modo de ejemplo mostramos el proceso seguido para calcular la transformada *Wavelet* y aplicar reducción de ruido.

```
while(filas_total < image->filas)
{
```

```
        if((filas_total + filas_nvidia < image->filas) && (inicio == 1))
        {
                //Primera porcion de imagen
                WaveletParalelizacion(image->pixeles+z*image->filas*
                 image->columnas, cuadrantes[i*4], cuadrantes[i*4+1],
                 cuadrantes[i*4+2], cuadrantes[i*4+3], FilterLowPass,
                 FilterHighPass, filas_nvidia, image->columnas,
                 rowsFilter1, colsFilter1, rowsFilter2, colsFilter2,
                 floor((filas_nvidia+colsFilter1-1)/2), columnas_
                 cuadrante[i], threshold, inicio, final, init, finish);

                inicio = 0;
                init = 0;
                filas_nvidia2 = filas_nvidia - 2;
        }

        else if((filas_total + filas_nvidia < image->filas) &&
         (inicio == 0))
        {
                //Porcion intermedia de imagen
                WaveletParalelizacion(image->pixeles+(filas_total-
                 (colsFilter1-2))*image->columnas+z*image->filas*
                 image->columnas, cuadrantes[i*4]+(filas_total/2)*
                 columnas_cuadrante[i], cuadrantes[i*4+1]+(filas_
                 total/2)*columnas_cuadrante[i], cuadrantes[i*4+2]+
                 (filas_total/2)*columnas_cuadrante[i], cuadrantes
                 [i*4+3]+(filas_total/2)*columnas_cuadrante[i],
                 FilterLowPass, FilterHighPass, filas_nvidia, image->
                 columnas, rowsFilter1, colsFilter1, rowsFilter2,
                 colsFilter2, floor((filas_nvidia+colsFilter1-1)/2),
                 columnas_cuadrante[i], threshold, inicio, final, init,
                 finish);

                filas_nvidia2 = filas_nvidia - 2 - colsFilter1/2;
        }

        else
        {
                //Porcion final de imagen

                final = 1;

                if(i == nivel_descomposicion-1)
                        finish = 1;

                if(inicio == 1) //Imagen que se procesa en una sola vez
                {
                        WaveletParalelizacion(image->pixeles+z*image->
                         filas*image->columnas, cuadrantes[i*4],
                         cuadrantes[i*4+1], cuadrantes[i*4+2],
                         cuadrantes[i*4+3], FilterLowPass,
                         FilterHighPass, image->filas, image->columnas,
                         rowsFilter1, colsFilter1, rowsFilter2,
                         colsFilter2, filas_cuadrante[i], columnas_
                         cuadrante[i], threshold, inicio, final, init,
                         finish);
```

```
                }

                else
                {
                        WaveletParalelizacion(image->pixeles+(filas_
                         total-(colsFilter1-2))*image->columnas+z*
                         image->filas*image->columnas, cuadrantes[i*4]
                         +(filas_total/2)*columnas_cuadrante[i],
                         cuadrantes[i*4+1]+(filas_total/2)*columnas_
                         cuadrante[i], cuadrantes[i*4+2]+(filas_total/
                         2)*columnas_cuadrante[i], cuadrantes[i*4+3]+
                         (filas_total/2)*columnas_cuadrante[i],
                         FilterLowPass, FilterHighPass, image->filas-
                         filas_total+(colsFilter1-2), image->columnas,
                         rowsFilter1, colsFilter1, rowsFilter2,
                         colsFilter2, floor((image->filas-filas_total+
                         (colsFilter1-2)+colsFilter1-1)/2), columnas_
                         cuadrante[i], threshold, inicio, final, init,
                         finish);
                }

                filas_nvidia2 = image->filas - filas_total;
        }

        filas_total += filas_nvidia2;
}
```

Para poder ejecutar los diferentes kernels es necesario definir el número de hilos por bloque y el número de bloques. En este caso se ha optado por trabajar con 512 hilos por bloque, y un número de bloques proporcional al número de píxeles del fragmento a procesar.

Como en toda aplicación con CUDA el primer paso es transferir toda la información de la memoria de la CPU a la memoria global de la GPU. En este caso se debe transferir los datos de la imagen de entrada y de los filtros paso alto y paso bajo a utilizar al realizar las convoluciones.

Para llevar a cabo todo el procesado se han definido una serie de kernels que realizan las mismas operaciones que en la aplicación desarrollada en C, pero mediante paralelización

- __global__ void ReduccionRuido(double *image, uint32_t rowsImage, uint32_t colsImage, double threshold);
- __global__ void SumarImagenes(double *image1, double *image2, double *result, uint32_t rowsImage, uint32_t colsImage);
- __global__ void ConstruirPrimeraMitadCuadranteCUDA(double *image, double *result, double *filtro, uint32_t rowsImage, uint32_t colsImage, uint32_t rowsFilter, uint32_t colsFilter, uint32_t colsResult);
- __global__ void ConstruirSegundaMitadCuadranteCUDA(double *image, double *result, double *filtro, uint32_t rowsImage, uint32_t colsImage, uint32_t rowsFilter, uint32_t colsFilter, uint32_t rowsResult);
- __global__ void ReconstruirPrimeraMitadCuadrante(double *image, double *result, double *filtro, uint32_t rowsImage, uint32_t colsImage, uint32_t rowsFilter, uint32_t colsFilter, uint32_t rowsResult);

- __global__ void ReconstruirSegundaMitadCuadrante(double *image, double *result, double *filtro, uint32_t rowsImage, uint32_t colsImage, uint32_t rowsFilter, uint32_t colsFilter, uint32_t colsResult);

9.4. Resultados obtenidos y conclusiones

Los resultados que se muestran a continuación han sido obtenidos al aplicar la transformada *Wavelet* con un filtro de Daubechies y con un nivel de descomposición igual a 10.

Los tiempos de ejecución obtenidos al ejecutar la aplicación con paralelización y sin paralelización se pueden ver en la Tabla 9.1. En la Figura 9.6 se han representado gráficamente.

Tiempo total sin paralelización	115.906.683
Tiempo total con paralelización	29.451.241
Tiempo en manejar la memoria en la GPU	23.996.112
Tiempo en procesar los datos en la GPU	2.414
Ahorro relativo (%)	75

Tabla 9.1: Tiempos de ejecución (μs) módulo de Reducción de ruido

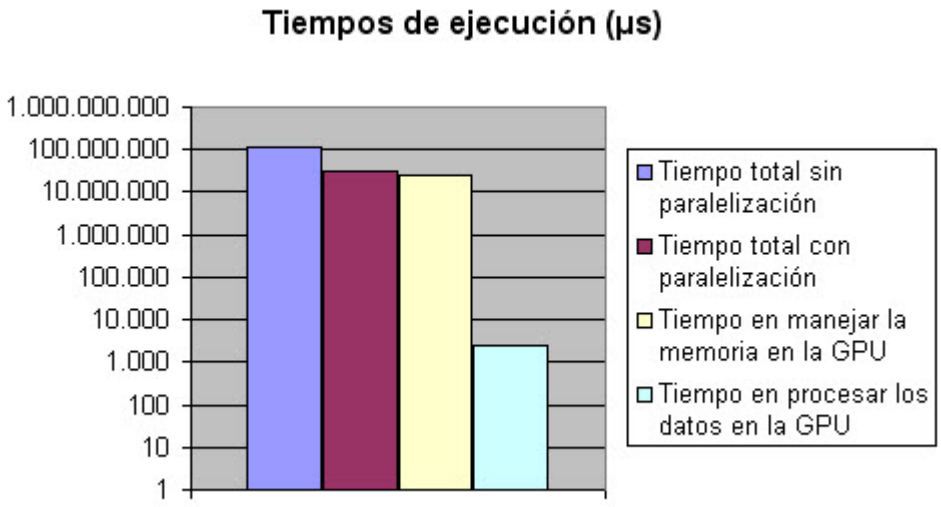

Figura 9.6: Tiempos de ejecución módulo de Reducción de ruido

Como vemos en los tiempos de ejecución obtenidos, el tiempo total con paralelización es bastante menor que el obtenido sin paralelización, lo que demuestra los beneficios que introduce la tecnología CUDA en un sistema de teledetección.

Si únicamente nos fijamos en los tiempos de procesado en la GPU vemos que el ahorro de tiempo es muy considerable.

Las imágenes de salida obtenidas con paralelización y sin paralelización se pueden ver en la Figura 9.7, donde se puede comprobar que ambas son iguales.

Para ejecutar la aplicación es necesario pasar como argumento el nombre de la imagen a procesar, como se puede ver en el siguiente ejemplo:

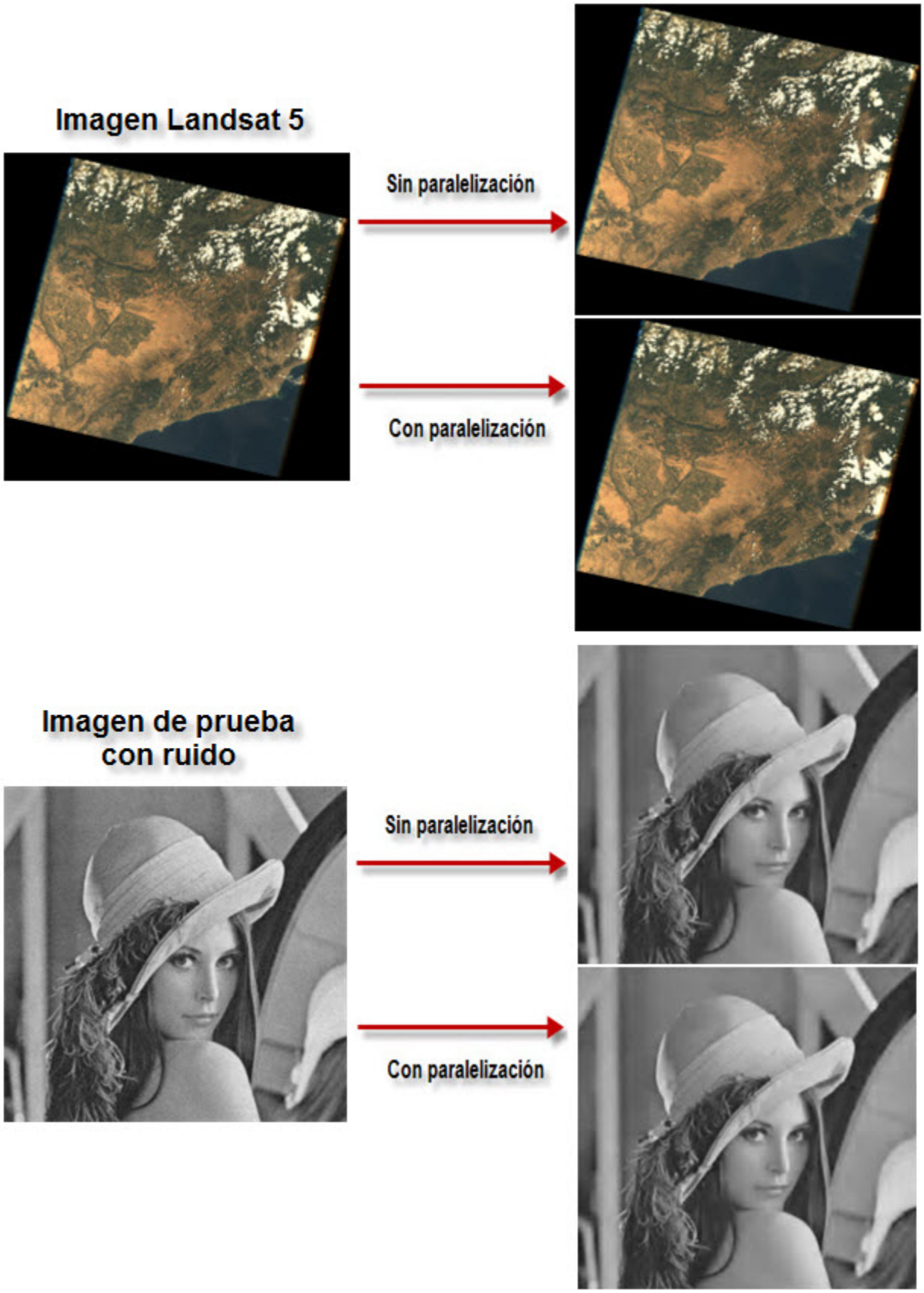

Figura 9.7: Resultados módulo de Reducción de ruido

```
./Wavelet original.tiff
```

Durante la ejecución, y tal y como se comentó anteriormente, la aplicación solicitará una serie de parámetros al usuario:

- Tipo de filtro (Haar(1), Daubechies(2), Symmlet(3), Coiflet(4)).
- Nivel de descomposición (Valores comprendidos entre 1 y 250).
- Umbral de ruido (Valores comprendidos entre 0 y 100).

Finalmente el usuario puede escoger si ejecutar la aplicación mediante paralelización o sin paralelización.

Capítulo 10

Corregistración

10.1. Introducción

El objetivo principal de este módulo es alinear todas las bandas de una imagen multiespectral respecto a una de las bandas de la imagen, que se toma como referencia. De esta forma se consigue que todas las bandas estén en la misma geometría.

Para ello se debe calcular el desplazamiento que tiene cada una de las bandas de la imagen respecto a la de referencia, para así generar un polinomio de salida cuyos coeficientes determinen la ecuación de desplazamiento. Para simplificar los cálculos vamos a trabajar con polinomios de primer grado.

A priori se conoce el desplazamiento teórico que debería haber entre las imágenes de las diferentes bandas espectrales respecto a la de referencia. Este desplazamiento teórico es debido a la diferente posición de cada uno de los sensores en el satélite, y nos va a permitir conocer la posición que debería tener un píxel en la imagen de búsqueda a partir de su posición en la de referencia.

El proceso que se debe seguir para calcular el desplazamiento real a partir del teórico consiste en dividir ambas imágenes en subimágenes, y calcular la correlación cruzada normalizada entre cada par de ellas, ya que de esta forma obtendremos el grado de similitud entre ambas. La correspondencia en la posición entre cada par de subimágenes está dada por el desplazamiento teórico.

Una vez calculada la correlación cruzada, se busca el valor máximo para comprobar si está por encima de un determinado umbral, de forma que si el valor obtenido es superior entonces hay suficiente similitud entre ambas subimágenes, por lo que existe una correspondencia entre ellas. En este caso se calcula el desplazamiento real entre ambas.

Este proceso se repite para todas las subimágenes, de forma que tendremos el desplazamiento real entre cada par de subimágenes que cumplan la condición explicada anteriormente.

Con todos los desplazamientos obtenidos se forma un sistema de ecuaciones lineales, que al resolverlo nos proporcionará la ecuación de salida con los coeficientes del polinomio de primer grado que representa el desplazamiento real entre la imagen de referencia y la de búsqueda.

Por lo tanto la salida de este módulo será de la forma:

$$X' = a_{00} + a_{01} * Y + a_{10} * X$$

$$Y' = b_{00} + b_{01} * Y + b_{10} * X \ ,$$

donde:

X: Coordenada x de la banda de referencia.

Y: Coordenada y de la banda de referencia.

X': Coordenada x de la banda de búsqueda.

Y': Coordenada y de la banda de búsqueda.

Un esquema de este procedimiento se puede ver en la Figura 10.1.

En definitiva, este módulo calcula el desplazamiento de una banda espectral respecto a una banda de referencia tanto en la coordenada X como en la coordenada Y. Como vemos, el desplazamiento de una coordenada puede depender de la otra coordenada espacial, y está caracterizado por un polinomio de primer grado.

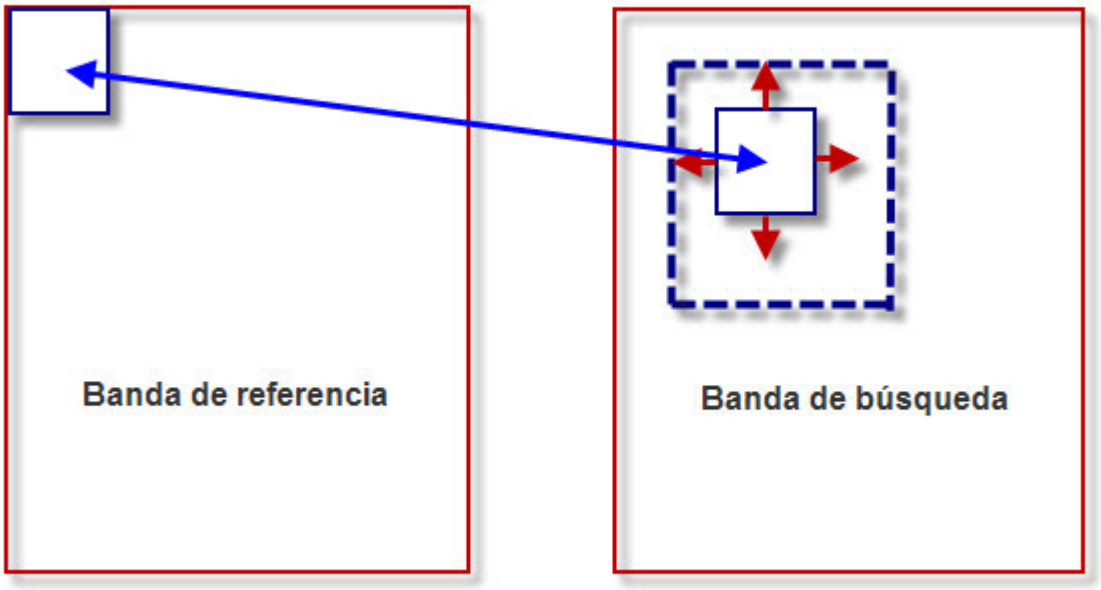

Figura 10.1: Proceso de corregistración

10.2. Aplicación desarrollada en C

Para implementar todo el procedimiento descrito se ha desarrollado una aplicación en C que ejecuta una serie de algoritmos para realizar la corregistración de una imagen multiespectral.

Inicialmente la aplicación solicita al usuario el umbral de correlación, el cual puede tomar valores comprendidos entre 0 y 1.

El error teórico entre la banda de referencia y la de búsqueda es definido implícitamente por la aplicación en función de cual sea la imagen de entrada a procesar.

Una vez definidos los parámetros característicos de este módulo se procede a la lectura de la imagen de entrada. Para ello se hace uso del correspondiente método de lectura de imágenes TIFF.

La imagen de entrada debe ser una imagen multiespectral formada por al menos dos bandas, para así poder realizar la corregistración entre ambas.

Posteriormente se lleva a cabo el procesado de la imagen de entrada. Para realizar este procesado es necesario definir una serie de parámetros, entre los que destacan el tamaño de la subimagen de referencia y de búsqueda. Se ha optado por definir un tamaño para la subimagen de búsqueda el doble que el de la de referencia, para así ir desplazando la de referencia sobre la de búsqueda y ver donde está el máximo de correlación.

El procedimiento que se va a describir se aplica sobre cada una de las bandas de la imagen, tomando como referencia de ellas la primera. Para ello se define un bucle for que vaya recorriendo las diferentes bandas espectrales de la imagen de entrada.

Lo primero que se hace es calcular el número de tiePoints, es decir, el número de subimágenes que vamos a poder obtener tanto en la imagen de búsqueda como en la de referencia, y que dependerá del tamaño de cada subimagen y de la separación entre ellas. Cada tiePoint contendrá las coordenadas del píxel central de la subimagen.

A partir del número de tiePoints definimos las coordenadas de los tiePoints candidatos, es decir, obtenemos las coordenadas de las subimágenes de referencia, y mediante el error teórico calculamos las coordenadas teóricas de las subimágenes de búsqueda. Este proceso se puede ver en los siguientes fragmentos de código.

```
//Asignamos las coordenadas a cada uno de los tiePoints candidatos
for(x=distanceToBorders; x<filas-distanceToBorders; x+=GridCellSize)
{
        for(y=distanceToBorders; y<columnas-distanceToBorders;
         y += GridCellSize)
        {
                //grid point at position (x, y). Add it to vector
                 of candidates
                tiePoints_candidatos[j].x_ref = x;
                tiePoints_candidatos[j].y_ref = y;

                j++;
        }
}
```

```
//A partir de los tiePoints de la imagen de referencia calculamos los
 de la imagen de busqueda mediante el desplazamiento teorico

for(i=0; i<NumberOfTiePoints; i++)
{
        if((x_teor.a00 + x_teor.a01*tiePoints_candidatos[i].y_ref +
         x_teor.a10*tiePoints_candidatos[i].x_ref <= (filas-
         searchWindowSizeX/2)) && (x_teor.a00 + x_teor.a01*
         tiePoints_candidatos[i].y_ref + x_teor.a10*
         tiePoints_candidatos[i].x_ref >= searchWindowSizeX/2))
                tiePoints_candidatos[i].x = x_teor.a00 + x_teor.a01*
                 tiePoints_candidatos[i].y_ref + x_teor.a10*
                 tiePoints_candidatos[i].x_ref;

        else
                tiePoints_candidatos[i].x=tiePoints_candidatos[i].x_ref;

        if((y_teor.a00 + y_teor.a01*tiePoints_candidatos[i].y_ref +
         y_teor.a10*tiePoints_candidatos[i].x_ref <= (columnas-
         searchWindowSizeY/2)) && (y_teor.a00 + y_teor.a01*
         tiePoints_candidatos[i].y_ref + y_teor.a10*tiePoints_
         candidatos[i].x_ref >= searchWindowSizeY/2))
                tiePoints_candidatos[i].y = y_teor.a00 + y_teor.a01*
                 tiePoints_candidatos[i].y_ref + y_teor.a10*
                 tiePoints_candidatos[i].x_ref;
        else
                tiePoints_candidatos[i].y=tiePoints_candidatos[i].y_ref;
}
```

Para cada uno de los tiePoints candidatos encontrados calculamos la subimagen de referencia y de búsqueda, para realizar una correlación cruzada normalizada entre ambas. El algoritmo usado para realizar la correlación cruzada entre dos subimágenes se puede ver en el siguiente fragmento de código.

```
for (shiftX=-kerDeltaX; shiftX<=kerDeltaX; shiftX++)
{
        int64_t B_initX = shiftX + halfDiffSizeX;
        int64_t B_endX  = B_initX + aDimX - 1;

        for (shiftY=-kerDeltaY; shiftY<=kerDeltaY; shiftY++)
        {
                double ccNumerator = 0;
                double ccDenominator = 0;

                int64_t B_initY = shiftY + halfDiffSizeY;
                int32_t A_initY = 0;

                if (B_initY < 0)
                        A_initY = 0 - B_initY;

                int32_t A_endY = aDimY;

                if (A_endY > bDimY - B_initY)
                        A_endY = bDimY - B_initY;

                // get search window subimage mean
                double mean = getSubimageMean (imageB, filasB,columnasB,
                 B_initX, B_endX, B_initY, (shiftY + offsetBY));

                for (x = 0; x < aDimX;)
                {
                        int64_t _x = x + B_initX;

                        if (_x >= bDimX) //no more overlap between
                         windows, no more contributions to CC value
                                break;

                        if (_x < 0)
                        {
                                //no overlap yet ,skip these first rows
                                x = (0 - _x); //rows to skip
                                continue;
                        }

                        for (y = A_initY; y < A_endY; y++)
                        {
                                float valueB = imageB[(_x)*columnasB +
                                 (y+B_initY)];
                                float valueA = offsetA[x*columnasA + y];
                                ccNumerator   += valueA * (valueB-mean);
                                ccDenominator += (valueB-mean) * (valueB
                                 -mean);
                        }

                        x++;
```

```
                }

                double ccValue; //cross-correlation value

                if (fabs(ccDenominator) <= PRECISION_MARGIN)
                        //denominator is zero, set CC to 0
                        ccValue = 0;
                else
                {
                        ccDenominator = sqrt (ccDenominator);
                        ccDenominator *= ccDenominatorFactor1;

                        ccValue = ccNumerator / ccDenominator;
                }

                //set pixel value in correlation surface
                ccSurface[i] = ccValue;

                i++;
        }
}
```

Como podemos observar, este algoritmo necesita una serie de parámetros estadísticos de las subimágenes de referencia y de búsqueda. Para ello hace uso de las siguientes funciones:

- getStatistics: Función que permite obtener la media, desviación típica, valor máximo y valor mínimo de la imagen que se le pasa como argumento.
- getMean: Función que calcula la media de la imagen que se le pasa como argumento.
- getSubimageMean: Función que calcula la media de un fragmento de la imagen que se le pasa como argumento.

Una vez calculada la correlación entre el par de subimágenes de referencia y de búsqueda asociadas a un tiePoint, se busca su valor máximo. Si este valor es mayor que el umbral de correlación establecido previamente por el usuario, entonces el tiePoint es correcto, ya que existe suficiente similitud entre ambas subimágenes, por lo que se calculan las coordenadas finales de la imagen de referencia y de búsqueda.

Las coordenadas finales de la subimagen de referencia se corresponden con las iniciales del tiePoint candidato, mientras que las de la subimagen de búsqueda se obtienen a partir de las del tiePoint candidato y el píxel central de la subimagen.

```
if(max >= threshold)
{
        //El tiePoint es correcto
        //Calculamos la separacion real entre la imagen de referencia
         y la de busqueda y la almacenamos en la estructura de salida

        uint32_t shiftX = x_max - searchHalfX;
        uint32_t shiftY = y_max - searchHalfY;

        tiePoints[total_tiePoints].x_ref=tiePoints_candidatos[i].x_ref;
        tiePoints[total_tiePoints].y_ref=tiePoints_candidatos[i].y_ref;
```

```
        tiePoints[total_tiePoints].x = tiePoints_candidatos[i].x +
         shiftX;
        tiePoints[total_tiePoints].y = tiePoints_candidatos[i].y +
         shiftY;

        total_tiePoints++;
}
```

Una vez realizado todo este procesado sobre cada uno de los tiePoints candidatos, obtenemos el número de tiePoints finales (los que cumplen la condición), y las coordenadas finales de cada uno de ellos. A partir de estos datos generamos un sistema de ecuaciones lineales, el cual resolveremos mediante la librería 'lapack', que provee un método para ello.

La ecuación a resolver es de la forma:

$$A * C = B,$$

donde A es una matriz que contiene las coordenadas de cada una de las subimágenes de referencia asociadas a cada tiePoint encontrado, B es un vector que contiene la coordenada de la imagen de búsqueda asociada a cada tiePoint, y C es un vector que contendrá los coeficientes del polinomio buscado.

Tenemos que obtener el polinomio de desplazamiento tanto para la coordenada X como para la coordenada Y. En función de la coordenada con la que se trabaje, el vector B contendrá los datos asociados a esa coordenada.

Para resolver el sistema de ecuaciones lineales se ha optado por usar la función 'dgels_'. Esta función proporciona la solución mediante una descomposición por mínimos cuadrados, para lo cual realiza una triangulación de la matriz de coeficientes.

Una vez resuelto el sistema de ecuaciones lineales se muestra por pantalla el resultado obtenido. Este resultado estará formado por los coeficientes del polinomio de primer grado que representa la ecuación de desplazamiento tanto para la coordenada X como para la coordenada Y.

10.3. Aplicación desarrollada con CUDA

Para reducir los tiempos de ejecución de la aplicación se ha realizado una paralelización de la misma, de forma que todo el procesado sea realizado por los multiprocesadores incorporados en la tarjeta gráfica.

En este módulo las pruebas se han realizado con imágenes multiespectrales cuyo tamaño no es demasiado grande, por lo que no hay problemas de memoria en la tarjeta gráfica. Además el procesado se va haciendo por bandas, por lo que solo es necesario transferir la información correspondiente a la banda de referencia y de búsqueda a la memoria de la GPU.

Por lo tanto el primer paso es transferir toda la información asociada a la banda de referencia y a la de búsqueda a la memoria global de la GPU. Para llevar a cabo todo el procesado también es necesario transferir las coordenadas de los diferentes tiePoints candidatos a la memoria global de la GPU.

Para ejecutar los diferentes kernels es necesario definir el número de hilos por bloque y el número de bloques.

En este caso se ha optado por trabajar con 256 hilos por bloque, debido a que el kernel que realiza la correlación requiere una gran cantidad de registros para realizar los cálculos, ya que este algoritmo lleva asociado una carga computacional elevada. En toda aplicación desarrollada con CUDA siempre existe una relación inversa entre el número de hilos por bloque y el número de registros disponibles. Si se hubiera escogido un total de 512 hilos por bloque se habrían producido errores durante la ejecución del kernel, ya que no habría suficientes registros disponibles para todos los hilos.

El número de bloques de hilos es proporcional al número de tiePoints encontrados en las imágenes a procesar, como podemos ver en el siguiente fragmento de código:

```
//Usamos 256 hilos para que no haya problemas con el numero de
 registros del kernel
unsigned int num_threads = 256;

//Definimos el numero de hilos por bloque y el numero de bloques
dim3 dimBlock(num_threads);
dim3 dimGrid(NumberOfTiePoints/num_threads + (NumberOfTiePoints
 %num_threads == 0?0:1));
```

Todo el procesado se ha realizado por partes, de forma que se han definido tres kernels, donde cada uno de ellos se encarga de realizar una función específica.

- CalcularSubimagenes: Kernel que permite calcular las diferentes subimágenes asociadas a la banda de referencia y a la de búsqueda.
- CrossCorrelationCUDA: Kernel que realiza la correlación entre cada par de subimágenes.
- BuscarMaximo: Kernel que obtiene el valor máximo de correlación asociado a cada par de subimágenes.

En los tres casos todo el procesado se va realizando en paralelo, de forma que se realizan las correspondientes operaciones sobre cada tiePoint de forma simultánea, con el consiguiente ahorro de tiempo.

A modo de ejemplo se muestra el código asociado al kernel que calcula las subimágenes para cada uno de los tiePoints:

```
__global__ void CalcularSubimagenes(double *referencia,double *busqueda,
 double *refW, double *searchW, uint32_t filas, uint32_t columnas,
 uint16_t refHalfX, uint16_t refHalfY, uint16_t searchHalfX, uint16_t
 searchHalfY, struct coordenadas *tiePoints_candidatos, uint32_t
 NumberOfTiePoints)
{

//Indice de bloque
int bx = blockDim.x * blockIdx.x;

//Indice de hilo
int tx = threadIdx.x;

if(bx+tx < NumberOfTiePoints)
{
```

```
        uint64_t j1, j2;

        //Subimagen de referencia
        for(j1=tiePoints_candidatos[bx+tx].x_ref-refHalfX; j1<=
         tiePoints_candidatos[bx+tx].x_ref+refHalfX; j1++)
        {
                for(j2=tiePoints_candidatos[bx+tx].y_ref-refHalfY;
                 j2<=tiePoints_candidatos[bx+tx].y_ref+refHalfY; j2++)
                {
                        refW[((bx+tx)*(2*refHalfX+1)*(2*refHalfY+1))
                         +(j1-(tiePoints_candidatos[bx+tx].x_ref-
                         refHalfX))*(2*refHalfY+1)+ (j2-(tiePoints_
                         candidatos[bx+tx].y_ref-refHalfY))] =
                         referencia[j1*columnas+j2];
                }
        }

        //Subimagen de busqueda
        for(j1=tiePoints_candidatos[bx+tx].x-searchHalfX; j1<=
         tiePoints_candidatos[bx+tx].x+searchHalfX; j1++)
        {
                for(j2=tiePoints_candidatos[bx+tx].y-searchHalfY;
                 j2<=tiePoints_candidatos[bx+tx].y+searchHalfY; j2++)
                {
                        searchW[((bx+tx)*(2*searchHalfX+1)*(2*
                         searchHalfY+1))+(j1-(tiePoints_candidatos
                         [bx+tx].x-searchHalfX))*(2*searchHalfY+1)
                         + (j2-(tiePoints_candidatos[bx+tx].y-
                         searchHalfY))] = busqueda[j1*columnas+j2];
                }
        }
}

}
```

Como vemos, cada hilo se encarga de obtener la subimagen de referencia y de búsqueda asociado a un determinado tiePoint. Para ello hace uso del identificador de hilo y del identificador de bloque.

De la misma forma, en el segundo kernel cada hilo se encarga de realizar la correlación entre cada par de subimágenes y almacenar el resultado en el correspondiente vector.

En el tercer kernel cada hilo se encarga de obtener el máximo de correlación asociado a cada tiePoint y comprobar si está por encima de un determinado umbral. En ese caso obtiene las coordenadas finales del tiePoint.

Por lo tanto, cada hilo se va a encargar de realizar todas las operaciones asociadas a un determinado tiePoint, por lo que todos los cálculos correspondientes a los diferentes tiePoints se van a realizar de forma simultánea.

Finalmente se transfiere el resultado a la memoria de la CPU, con el que se formará un sistema de ecuaciones lineales exactamente igual al obtenido en la aplicación desarrollada en C. Para resolverlo y obtener el resultado final se utiliza la misma función que en caso de la aplicación secuencial.

10.4. Resultados obtenidos y conclusiones

Mediante la paralelización de la aplicación se han obtenido muy buenos resultados, como se puede ver en la Tabla 10.1. También se han representado gráficamente en la Figura 10.2.

Tiempo total sin paralelización	12.551.986
Tiempo total con paralelización	1.549.546
Tiempo en manejar la memoria en la GPU	382.510
Tiempo en procesar los datos en la GPU	1.159.451
Ahorro relativo (%)	88

Tabla 10.1: Tiempos de ejecución (μs) módulo de Corregistración

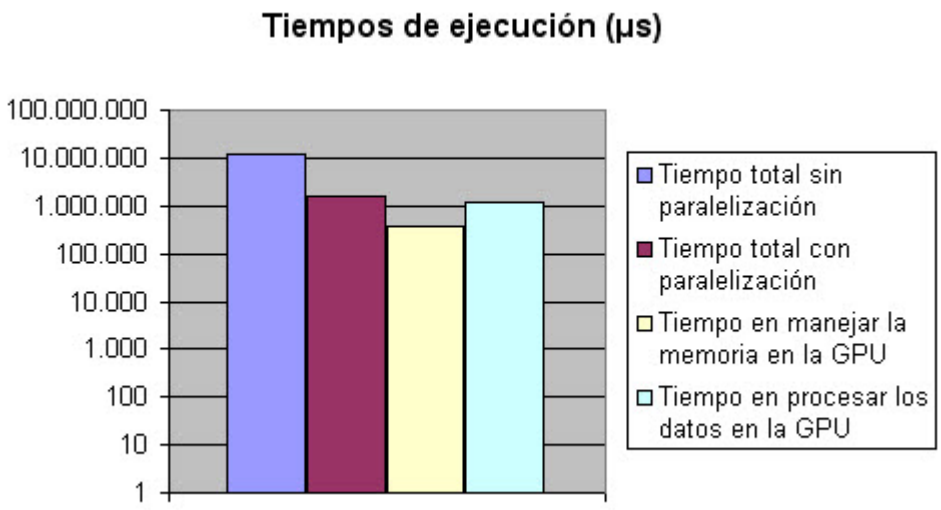

Figura 10.2: Tiempos de ejecución módulo de Corregistración

Como vemos, el tiempo total con paralelización es bastante menor que el obtenido sin paralelización, obteniéndose un factor de reducción próximo a 8.

En este caso, el tiempo de procesado mediante paralelización incluye además del empleado por la propia GPU para realizar los cálculos, el tiempo empleado para realizar ciertas operaciones de forma secuencial que no se han podido paralelizar. Debido a esta razón, y a que el proceso de correlación consume muchos recursos, el tiempo empleado por la GPU para procesar los datos es bastante mayor que el obtenido en otros módulos de la cadena AIPC.

Este módulo genera buenos resultados en cuanto a tiempos de ejecución debido a la alta carga computacional que tiene asociada, lo que hace que al ejecutar la aplicación de forma secuencial el tiempo empleado sea bastante mayor que el utilizado por la aplicación paralelizada. Por lo tanto, mediante paralelización se consiguen optimizar considerablemente los tiempos de ejecución de la aplicación.

Los resultados obtenidos para dos imágenes de entrada diferentes con paralelización y sin paralelización se pueden ver en la Figura 10.3, Figura 10.4, Figura 10.5, Figura 10.6, donde se puede comprobar que en ambos casos son iguales con paralelización y sin paralelización.

Como podemos apreciar, la primera imagen de entrada tiene un desplazamiento de 100 píxeles

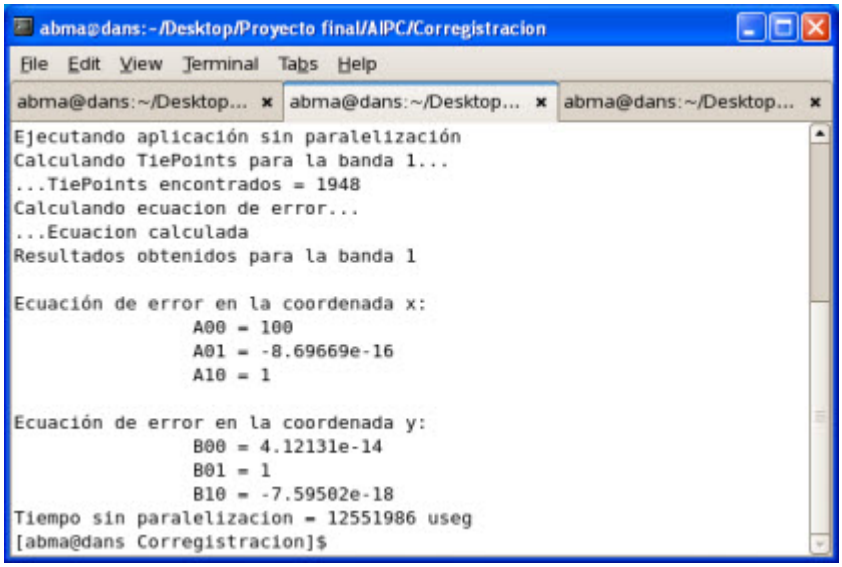

Figura 10.3: Resultados módulo de Corregistración imagen1 sin paralelización

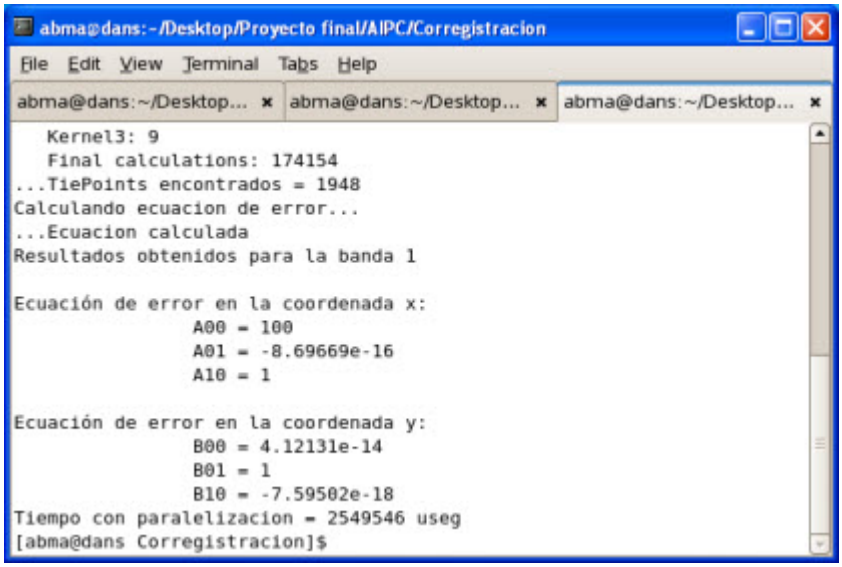

Figura 10.4: Resultados módulo de Corregistración imagen1 con paralelización

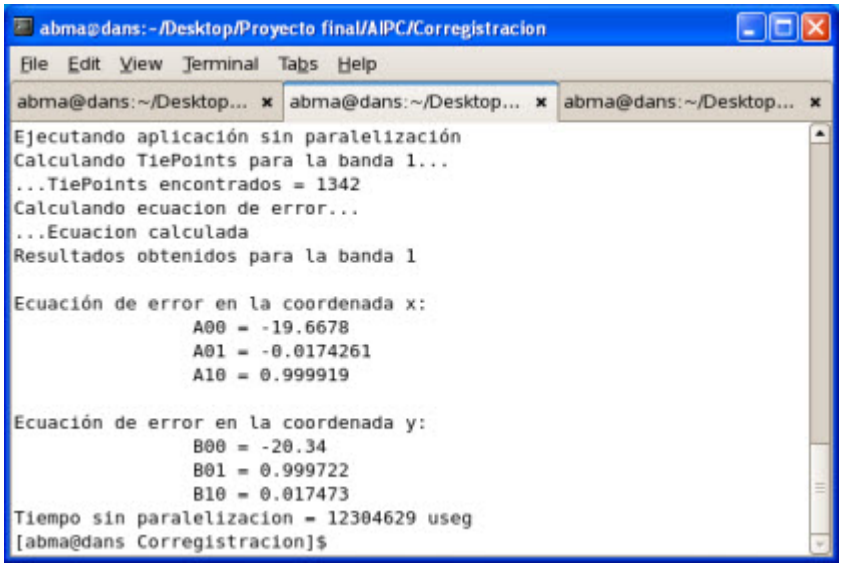

Figura 10.5: Resultados módulo de Corregistración imagen2 sin paralelización

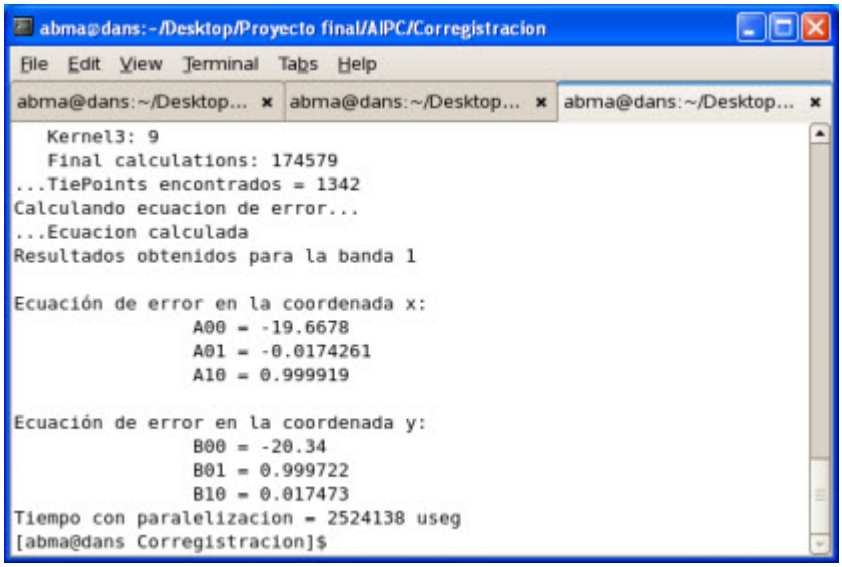

Figura 10.6: Resultados módulo de Corregistración imagen2 con paralelización

en la coordenada X, es decir, la imagen de búsqueda se ha desplazado verticalmente respecto a la de referencia. La segunda imagen tiene un desplazamiento de aproximadamente 20 píxeles tanto en la coordenada X como en la coordenada Y.

Por lo tanto, la salida de este módulo no es una imagen. En este caso la salida muestra el valor de los coeficientes del polinomio que representa el desplazamiento entre la banda de referencia y la de búsqueda. Las pruebas se han realizado con una imagen de entrada formada únicamente por dos bandas espectrales, por lo que los resultados muestran el desplazamiento sufrido por la coordenada X y la coordenada Y en la banda 2 respecto a la banda 1.

Para ejecutar la aplicación es necesario pasar como argumento el nombre de la imagen a procesar, como se puede ver en el siguiente ejemplo:

```
./Corregistracion chicago_x+100.tif
```

Durante la ejecución, y tal y como se comentó anteriormente, la aplicación solicitará el umbral de correlación al usuario, que podrá tomar valores comprendidos entre 0 y 1.

Finalmente el usuario puede escoger si ejecutar la aplicación mediante paralelización o sin paralelización.

Capítulo 11

Conclusiones

En este proyecto se ha tratado de presentar la tecnología *CUDA* de *Nvidia*. Para ello se ha realizado un análisis de las principales características que presenta, describiendo los conceptos mas importantes asociados a esta tecnología, y analizando las numerosas ventajas que puede introducir en un sistema de teledetección.

Mediante el desarrollo de diferentes aplicaciones se ha podido comprobar las mejoras que introduce. Para ello se ha realizado un estudio sistemático, comparando los resultados obtenidos al ejecutar la aplicación mediante paralelización y de forma secuencial, y comprobando que hay una reducción del tiempo requerido por la aplicación para realizar el procesado de todos los datos.

La línea de desarrollo seguida ha consistido en la paralelización de diferentes aplicaciones basadas en un sistema de teledetección.

Por un lado se ha desarrollado una aplicación que permite realizar un reconocimiento del terreno, y en la que se han obtenido muy buenos resultados mediante paralelización.

Por otro lado se han paralelizado cinco de los módulos pertenecientes a la cadena de procesado genérica *AIPC* de Deimos Space. Esta paralelización se ha realizado de forma totalmente independiente entre módulos, obteniendo buenos resultados.

En base a los resultados obtenidos, una posible línea de desarrollo futura podría consistir en la paralelización de todos los módulos que componen la cadena de procesado genérica AIPC, ya que permitiría acoplarlos todos en una única aplicación. De esta forma solo habría que hacer dos transferencias de datos entre memorias, una en el primer módulo de la cadena para transferir los datos de la memoria de la CPU a la memoria global de la GPU, y otra en el último módulo para transferir el resultado obtenido de la memoria de la GPU a la memoria de la CPU. Así se obtendría un gran ahorro de tiempo y el beneficio sería mucho mayor que el obtenido en cada módulo de forma independiente, ya que los módulos intermedios únicamente realizarían el procesado de la información, evitando tener que hacer transferencias de memoria.

De esta forma el sistema de teledetección sería capaz de generar las imágenes en un tiempo reducido, y aproximarse cada vez mas a un sistema de tiempo real.

En base a la experimentación y realización de las diversas pruebas se han extraído las siguientes conclusiones sobre la tecnología *CUDA*:

- El mayor tiempo de procesado en las aplicaciones paralelizadas estudiadas es el requerido para transferir datos entre la memoria del PC y de la tarjeta gráfica. Por lo tanto conviene hacer el mínimo número de transferencias entre memorias, o lo que es lo mismo, transferir los datos al principio y al final de la aplicación para minimizar el tiempo empleado por la misma.

- El tiempo empleado para realizar las operaciones una vez que se tienen los datos en la tarjeta gráfica es muy pequeño en comparación con el utilizado por la CPU.

- En toda aplicación siempre va a haber parte del código que se tenga que ejecutar de forma secuencial, no pudiéndose llevar a cabo una paralelización. Según la ley de Amdahl, el beneficio máximo que se puede obtener mediante paralelización es:

$$\frac{1}{1-P+\frac{P}{N}}$$

  siendo P la proporción de tiempo que lleva a cabo la paralelización, y N el número de procesadores disponibles.

- Hay que paralelizar únicamente aquella parte del código que permita realizar las operaciones en paralelo, ya que es donde realmente se va a obtener un beneficio.
- Cuanto mayor sea el volumen de datos a procesar mayor es el beneficio que se obtiene al paralelizar la aplicación, ya que mayor es la cantidad de datos que se pueden procesar en paralelo. Esto se puede comprobar con el módulo de Deconvolución (donde el volumen de datos es muy grande) y el módulo de Detección de nubes (donde el volumen de datos a manejar es menor).
- Cuanto mayor sea el número de operaciones a realizar y mas complejas sean éstas, mayor es el beneficio obtenido mediante paralelización. Esto se puede comprobar con los módulos de Calibración absoluta (pocas operaciones y sencillas) y el de Deconvolución (muchas operaciones y complejas).

Como resumen se detallan las principales características que presenta esta tecnología:

- Lenguaje C estándar para el desarrollo de aplicaciones de procesamiento paralelo en la GPU.
- Librerías numéricas estándar para FFT (Fast Fourier Transform) y BLAS (Basic Linear Algebra Subroutines) [20].
- Controlador *CUDA* dedicado a cálculo con comunicación de datos de alta velocidad entre la GPU y la CPU.
- El controlador de *CUDA* interacciona con los controladores de gráficos OpenGL y DirectX.
- Compatibilidad con sistemas operativos Linux de 32/64 bits, Windows XP de 32/64 bits, y Mac.
- Están disponibles herramientas gratuitas de desarrollo.

Por lo tanto la tecnología *CUDA* proporciona una forma óptima y de coste reducido de mejorar los algoritmos empleados en un sistema de teledetección.

Se puede concluir que es muy aconsejable y beneficioso la utilización de esta tecnología en sistemas de teledetección, ya que permite reducir considerablemente el tiempo transcurrido desde que el sensor captura los datos hasta que se genera la imagen final, optimizando de esta forma todos los algoritmos utilizados durante la cadena de procesado.

Capítulo 12

Pliego de condiciones

El presente proyecto se ha llevado a cabo haciendo uso del siguiente material:

12.1. Recursos hardware

- Ordenador personal con procesador Intel Core 2 Duo a 2.8 GHz y 2 GB de memoria RAM.
- Servidor de la empresa Deimos Space donde se encuentra instalada la tarjeta gráfica.
- Tarjeta gráfica NVidia, modelo GeForce GTX 280 que incorpora 30 multiprocesadores y 240 núcleos.
- Conexión a Internet de banda ancha.

12.2. Recursos software

- Sistema operativo Windows XP en ordenador personal.
- Sistema operativo Linux distribución Fedora en servidor.
- Aplicación Xmanager para conexión por acceso remoto.
- Drivers y software característico de la tecnología CUDA.
- Librerías de C.
- Librería Geotiff para manejo de imágenes GeoTiff.
- Imágenes multiespectrales capturadas por el satélite Landsat 5 TM.
- Aplicación IrfanView para visualización de imágenes multiespectrales.
- Aplicación TexnicCenter para desarrollo de la documentación en Latex.

Capítulo 13

Presupuesto

En el presente capítulo se recoge el cálculo estimado del presupuesto empleado en la realización del proyecto. Para ello se realiza un desglose en dos partes bien diferenciadas:

- Costes materiales
- Costes personales

13.1. Costes materiales

Bajo este concepto se incluyen todos los costes asociados al uso de los diferentes componentes hardware y software durante el desarrollo del proyecto.

13.1.1. Costes por materiales hardware

Equipo	Precio	Amortización	Uso	Subtotal
Ordenador personal	1000 €	5 años	9 meses	150 €
Servidor	3000 €	5 años	9 meses	450 €
Tarjeta gráfica	300 €	5 años	9 meses	45 €
Conexión a Internet	39.90 €/mes	-	9 meses	359.10 €
Total	**1004.10 €**			

Tabla 13.1: Costes materiales debidos a los componentes hardware

13.1.2. Costes por materiales software

Software	Precio unitario	Cantidad	Subtotal
Microsoft Windows XP	100 €	1	100 €
Licencia para Xmanager	83 €	1	83 €
Licencia para IrfanView	10 €	1	10 €
Microsoft Office	300 €	1	300 €
Total	**493 €**		

Tabla 13.2: Costes materiales debidos a los componentes software

13.2. Costes personales

Bajo este concepto se incluyen todos los costes asociados a la mano de obra necesaria para el desarrollo del proyecto.

Función	Tiempo	Coste	Subtotal
Desarrollo	900 horas	30 €/hora	27.000 €
Documentación	140 horas	12 €/hora	1.680 €
	Total	**28.680 €**	

Tabla 13.3: Costes materiales debidos a la mano de obra

13.3. Coste total

El importe total asociado a los diferentes costes es:

Tipo de coste	Subtotal
Costes por material hardware	1004.10 €
Costes por material software	493 €
Costes por mano de obra	28.680 €
Total	**30.177,10 €**

Tabla 13.4: Coste total

Al importe total obtenido se le debe sumar el 16 % de I.V.A.

Descripción	Subtotal
Coste total	30.177,10 €
16 % I.V.A.	4.828,34 €
Total	**35.005,44 €**

Tabla 13.5: Coste total (I.V.A. incluido)

El Importe Total del proyecto asciende a la cifra de **TREINTA Y CINCO MIL CINCO EUROS CON CUARENTA Y CUATRO CÉNTIMOS**

Capítulo 14

Bibliografía

Bibliografía

[1] **Curso de teledetección, 2008. Universidad Politécnica de Madrid.**

[2] **NVidia. *CUDA Programming Guide*, 2008.**

[3] **Johan Seland. Cuda programming. Technical report, Geilo Winter School, 2008.**

[4] **Ruth Montes Fraile Borja José García Menéndez, Eva Mancilla Ambrona. Optimización de la transformada wavelet discreta (dwt). Technical report, Universidad Complutense de Madrid, 2005.**

[5] ***CUDA Technology* [online]. 2009. Disponible en:** `http://www.nvidia.com/object/cuda_home.html`.

[6] **Estudio del Estado del Arte en Sistemas Grid Específicos para Algoritmos Espaciales. Entregable Proyecto CENIT España Virtual, 2008. Deimos Space y Grid Systems.**

[7] ***Agencia Espacial Europea* [online]. 2009. Disponible en:** `http://www.esa.int/esaCP/index.html`.

[8] **System design document. Technical report, Deimos Space, 2008. Internal Reference: AIPC-DMS-TEC-ADD01.**

[9] ***Imágenes GeoTiff* [online]. 2009. Disponible en:** `http://trac.osgeo.org/geotiff/`.

[10] **Algorithmic models: Modules e-g. Technical report, Deimos Space, 2008. Internal Reference: AIPC-DMS-TEC-TNO04.**

[11] **Requirements baseline: co-registration, ortho-rectification and mosaicking. Technical report, Deimos Space, 2008. Internal Reference: AIPC-DMS-TEC-TNO02.**

[12] ***Definición de taxonomía de Flynn según la wikipedia* [online]. 2009. Disponible en:** `http://es.wikipedia.org/wiki/Taxonomía_de_Flynn`.

[13] **NVidia. *CUDA Getting Started*, 2008.**

[14] **NVidia. *CUDA Reference manual*, 2008.**

[15] **Dennis L. Helder Gyanesh Chander, Brian L. Markham. Summary of current radiometric calibration coefficients for landsat mss, tm, etm+, and eo-1 ali sensors. Technical report, Elsevier Inc., 2009.**

[16] ***Landsat*** **[online]. 2009. Disponible en:** `http://landsathandbook.gsfc.nasa.gov/handbook.html`**.**

[17] **W. Takeuchi y Y. Yasuoka. Development of normalized vegetation, soil and water indices derived from satellite remote sensing data. Technical report, IIS/UT, Japan, 2004.**

[18] **Richard R. Irish. Landsat 7 automatic cloud cover assessment. Technical report, NASA.**

[19] **Rafael C. Gonzalez y Richard E. Woods. *Digital Image Processing*. Addison-Wesley, 2nd edición, 2003.**

[20] **NVidia. *CUBLAS Library*, 2008.**