UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

DEPARTAMENTO DE ARQUITECTURA DE COMPUTADORES Y AUTOMÁTICA

PROCESAMIENTO DE IMÁGENES HIPERESPECTRALES EN GPUs

TESIS DOCTORAL DE:

JAVIER SETOAIN RODRIGO

DIRIGIDA POR:

MANUEL PRIETO MATÍAS

CHRISTIAN TENLLADO VAN DER REIJDEN

Madrid, 2013

Procesamiento de Imágenes Hiperespectrales en GPUs

Departamento de Arquitectura de Computadores y Automática

Universidad Complutense de Madrid

TESIS DOCTORAL

Javier Setoain Rodrigo

Madrid, septiembre de 2013

Procesamiento de Imágenes Hiperespectrales en GPUs

Memoria presentada por Javier Setoain para optar al grado de Doctor por la Universidad Complutense de Madrid, realizada bajo la dirección de Manuel Prieto Matías y Christian Tenllado van der Reijden.

Madrid, 2 de septiembre de 2013.

Agradecimientos

Quiero empezar por agradecer su apoyo y su paciencia a Manuel y Christian, mis directores, que son las personas con las que más de cerca he trabajado durante el desarrollo de esta tesis. Dentro del mismo ámbito, también merecen mi más sincero agradecimiento toda la gente que nos da soporte en los laboratorios, que durante estos años han sido muchos: Enrique, César, Luis, Jorge, Eze, Adri, Rober, Dani, Fede, Jose Manuel y Raúl, que no solo se han encargado de que todo funcionase como tiene que funcionar, además me han dejado hacer y deshacer a mi antojo cuando así lo he necesitado. ¡Qué majos! Un agradecimiento especial se merece Antonio Plaza, que es quien plantó la semilla de este trabajo y alguien a quien siempre hemos podido acudir cuando lo hemos necesitado; ha sido todo un placer conocer a una persona tan afable y todo un privilegio trabajar con alguien de tanto talento.

Pero han sido muchos años y no solo he aprendido de la gente con la que he trabajado directamente, también hay muchos compañeros que me han enseñado mucho y a quienes estaré siempre agradecido por haber estado ahí. Muchas gracias Luis y muchas gracias Nacho.

Si hay algo que puedo destacar de todo este tiempo que he pasado trabajando en la Facultad de Ciencias Físicas es sin duda la gente que me ha rodeado. No importa si el trabajo iba bien o iba mal, con los compañeros que he tenido siempre he estado a gusto aquí y eso es lo que más tengo que agradecer. Manel, Fernado, Conchi Pineda y Conchi Díaz, mis compañeros de despacho. Dani, secretario del departamento y persona excepcional. Carlos García y Juan Fran,

compañeros inmejorables a los que ha sido un lujo tener por los alrededores, Guillermo, Joaquín, David Sánchez, los que he mencionado y mencionaré y los que seguro que me estoy dejando. Reservo un agradecimiento especial a una de las mejores personas que he tenido la suerte de conocer, Silvia, que por desgracia nos dejó en 2011. Recuerdo con cariño las conversaciones durante la comida y tu risa contagiosa. A todos mis compañeros, muchas gracias por estar ahí, gracias a gente tan estupenda se encara con alegría el día a día, y el trabajo ya no parece trabajo.

Por supuesto, no puedo dejar de agradecer a mis compañeros de aventuras: los doctorandos. Muchos han sido: Conchi, Rubén, Carolina, Hugo, Fermín, Rodri, Carlos, Pablo, Juan Carlos, Edgardo, Dani y el silencioso Ricardo. Gracias por ser tan majetes.

También quiero agradecer a gente con la que he trabajado puntualmente o con la que he coincidido en algún momento de todo este proceso que es convertirse en doctor. Empiezo por Fran Igual, que me ha echado una mano al final del todo. Gracias a Manuel Arenaz de la Universidad de A Coruña y David Valencia de la Universidad de Extremadura. Un recuerdo especial para la gente del *Barcelona Supercomputing Center*: Xavi, Rosa y Eduard, que me enseñaron mucho y me hicieron sentir bienvenido; Jordi, Xavi, Roger, Jairo, Javi y Juanjo, que me brindaron su amistad y me hicieron sentir en casa. ¡Moltes gràcies!

Y ahora, casi terminando, un rinconcito especial para todos aquellos compañeros que, además de compañeros, me han honrado con su amistad en algún momento. Enrique, Rubén, Rodri, Carol, Hugo, Fermín, Jorge, Dani y Nacho; César, Fede, Eze, Rober, Héctor, Luis, Adri, Carlos y Darío. ¡Gracias por

molar tanto!

Llega el turno de los agradecimientos a las personas que siempre han estado, están y siempre estarán ahí: ¡mi numerosísima familia! !Muchas gracias! Un agradecimiento especial para mi tío Mikel, que ya no está con nosotros. A él le debo (junto a Jesus y a Alfredo) mi sentido del humor, que es la parte de mí que más me gusta y más me reconforta; no habría sobrevivido a esto sin él.

Muchas gracias también a mi chica, Paola, y a sus padres, Pepe y Nati, que me han ayudado y apoyado muchísimo en la última fase de esta tesis. Habéis jugado un papel imprescindible en esto.

Para cerrar la sección de agradecimientos personales, muchas gracias a mis viejos amigos, a los que siempre tengo presente aunque casi no nos veamos. Los del barrio: Abraham, Omar, Jesus, Oscar y Raúl. Los de Calatayud: Isaac, Pepe, Alberto y las Malo (Laura y Marta). Soy quien soy porque crecí con vosotros. No me quiero olvidar tampoco de los amigos de la carrera, a los que tampoco veo ni la mitad de lo que quisiera pero que han estado ahí: David y Teresa, Tomás, Raúl, Fran, Pablo, Lucas, Adri y demás peña.

Por último quiero agradecer su apoyo y su confianza a Paco Tirado, director del grupo de investigación *ArTeCS*, que me hizo un hueco y sin cuyo trabajo no habría tenido a mi disposición los recursos que he necesitado para mi doctorado. Y con él también agradezco el soporte y financiación de este trabajo a los proyectos TIN2005-5619, TIN2008-00508, TIN2012-32180 y a la red de Excelencia Europea HIPEAC.

Índice general

1. Introducción **1**

1.1. Evolución de las Arquitecturas Paralelas 2

1.2. Arquitectura de las GPUs 10

1.3. OpenGL 20

1.4. CUDA 22

1.5. Imágenes Hiperespectrales 25

1.6. Motivación 26

1.7. Objetivos 28

2. Procesamiento de Imágenes Hiperespectrales **31**

2.1. El Algoritmo *AMEE* 34

2.1.1. *AMEE*: Cálculo multi-pasada del mapa *MEI* 39

2.1.2. La función de cálculo de distancia 41

2.1.3. Precisión 43

2.2. Conclusión 45

3. Procesamiento de Imágenes Hiperespectrales con *OpenGL* **47**

3.1. El Modelo de Procesamiento de Flujos 48

3.2. GPGPU sobre OpenGL 51
3.3. *AMEE* en OpenGL 60
3.3.1. *SID AMEE* 65
3.3.2. *SAM AMEE* en OpenGL 69
3.4. Resultados 74
3.4.1. Entorno experimental 74
3.4.2. Resultados experimentales 76
3.5. Conclusiones 81

4. Procesamiento de Imágenes Hiperespectrales con *CUDA* 85

4.1. Introducción a *CUDA* 86
4.2. Análisis del rendimiento de *CUDA* en *GPUs* 95
4.2.1. Factores de rendimiento 95
4.2.2. Análisis del rendimiento 101
4.2.2.1. Explotación de la ocupación 103
4.2.2.2. Explotación de la jerarquía de memoria . . . 109
4.2.2.3. Conclusiones del análisis 113
4.3. *AMEE* sobre *CUDA* 114
4.3.1. De *OpenGL* a *CUDA* 116
4.3.2. Minimización del número de *kernels* 118
4.3.3. Mejora del acceso a la memoria 120
4.3.4. Aprovechamiento de la *cache* bidimensional 121
4.3.5. Aprovechamiento de las texturas tridimensionales . . . 122
4.3.6. Comparación de versiones 123
4.4. Conclusiones 125

5. Principales Aportaciones **131**

5.1. Trabajo futuro 133

A. Hyperspectral Imaging on GPUs: A Summary in English **I**

A.1. Introduction II

A.2. Hyperspectral Imaging XVI

A.2.1. The *AMEE* algorithm XVIII

A.2.2. *AMEE*: Multi-pass *MEI* map XXIII

A.2.3. Distance function XXV

A.2.4. Accuracy XXVII

A.3. Hyperspectral Imaging on *OpenGL* XXIX

A.4. Hyperspectral Imaging on *CUDA* XLVII

A.4.1. Performance analysis XLVII

A.4.2. *AMEE* on *CUDA* LVII

A.4.3. Conclusions LXV

A.5. Main Contributions LXX

A.6. Future work LXXII

B. Introducción a OpenGL **LXXV**

C. Biblioteca de desarrollo para GPGPU **LXXXIII**

Bibliografía **XCIII**

Índice de figuras **XCIX**

Índice de tablas **CXIII**

Capítulo 1

Introducción

En este primer capítulo introducimos los conceptos con los que trabajamos a lo largo de esta tesis, poniendo en contexto nuestro trabajo y explicando los objetivos perseguidos durante nuestra investigación. En la sección 1.1 hacemos un resumen de la historia de las arquitecturas paralelas y cómo han evolucionado hasta convertirse en las actuales arquitecturas *CMP* (*Chip Multi-Procesador*), donde se enmarca nuestro trabajo. En la sección 1.2 explicamos cómo están diseñadas nuestras plataformas de estudio: las unidades de procesamiento gráfico o *GPUs* – de sus siglas en inglés, *Graphics Processing Units* – con qué objetivo fueron diseñadas y cómo han evolucionado hasta convertirse en procesadores masivamente paralelos. En la sección 1.3 se hace una pequeña introducción a una de las bibliotecas de acceso al *hardware* de una *GPU*, *OpenGL*, diseñada con el objetivo de utilizar la *GPU* para sintetizar gráficos. En la sección 1.4 comentamos *CUDA*, el entorno de desarrollo propuesto por *NVIDIA* como solución para utilizar las *GPUs* como procesadores de propósito general. En la sección 1.5 hacemos una introducción al ámbito de aplicación

con el que trabajamos: las imágenes hiperespectrales. Finalizamos presentando nuestras motivaciones y una declaración de los objetivos perseguidos durante el desarrollo de este trabajo en las secciones 1.6 y 1.7 respectivamente.

1.1. Evolución de las Arquitecturas Paralelas

En 1965, Gordon Moore, co-fundador de Intel, señaló que, con los avances que se iban produciendo en la tecnología de integración, eran capaces de doblar el número de transistores integrados en un chip cada 18 meses. A esta afirmación se la ha conoce como la "Ley de Moore" [Moo65], y ha venido cumpliéndose con casi total exactitud hasta el día de hoy, como se puede ver en la figura 1.1.

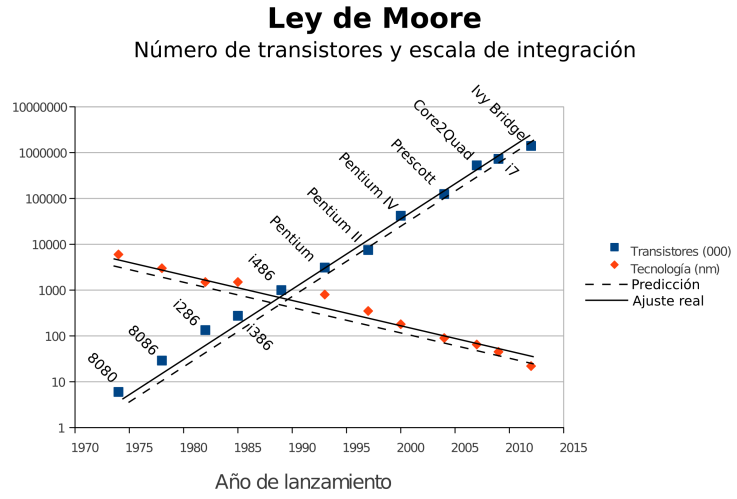


FIGURA 1.1: Predicción y ajuste a la Ley de Moore del número de transistores y la escala de integración.

Esta "ley" ha ido influyendo en el rendimiento de los procesadores de dos maneras fundamentalmente:

1. Conforme mejora el proceso de integración, los transistores se vuelven más rápidos, permitiendo aumentar la frecuencia de funcionamiento de los procesadores construidos con dichos transistores (lo que se conoce como el "Escalado de Dennard").

2. Los diseñadores de procesadores pueden utilizar el creciente número de transistores para extraer más paralelismo de los programas de forma transparente al programador.

Desde el punto de vista del programador generalista, un computador consiste en un único procesador que ejecuta un flujo secuencial de instrucciones y que está conectado a una memoria monolítica que contiene todos los datos e instrucciones, lo que se conoce como una arquitectura Von Neumann (ver figura 1.2).

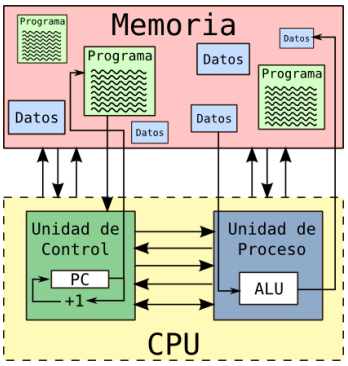


FIGURA 1.2: En la arquitectura propuesta por John Von Neumann, un computador consiste en una memoria que contiene tanto los datos como los programas, y en una CPU que es la encargada de ejecutar los programas que procesan dichos datos. Una CPU está compuesta por una unidad de control, que gestiona el flujo de ejecución de los programas, y una unidad de proceso, que gestiona el procesamiento de los datos. En la figura, *PC* (*Program Counter*) es el contador de programa, encargado de seguir el flujo de ejecución de las instrucciones de un programa. *ALU* (*Arithmetic Logic Unit*) es la unidad aritmético-lógica, que realiza operaciones aritméticas y lógicas sobre datos.

Los diseñadores de hardware, para evitar la necesidad de volver a formar

a los programadores y de rehacer los programas, se han centrado sobre todo en mejoras que respetasen la abstracción propuesta por Von Neumann.

Mejoras en la memoria

Por un lado han utilizado los transistores adicionales para aumentar el tamaño de las memorias *cache*, memorias mucho más rápidas y cercanas al procesador en las que guardamos los datos accedidos con más frecuencia. También han aumentado el número de registros arquitectónicos, permitiendo mantener más datos en memoria de acceso inmediato, así como el tamaño de los bancos de registros a nivel físico, lo que permite que las técnicas de renombramiento de registros aumenten la eficiencia del procesador.

Mejoras en el procesador

Para mantener la abstracción de arquitectura de Von Neumann, las mejoras en el diseño del procesador se hacen buscando dos objetivos principalmente:

1. Aumentar la frecuencia de reloj por encima de lo que lo permite de forma natural la mejora en el proceso de integración predicha por Moore.

2. Aumentar el número de instrucciones de la secuencia del programa que se pueden lanzar en cada ciclo.

Respecto al primer objetivo, lo que se busca es segmentar la ejecución de las instrucciones individuales en una secuencia de etapas, permitiendo aumentar la frecuencia de reloj. Conforme las instrucciones se dividen en un número cada vez mayor de pasos, reduciendo la lógica que necesita cambiar de estado

dentro del mismo ciclo, podemos aumentar la frecuencia del reloj. Dichas etapas, al ocupar diferentes recursos del procesador, pueden solaparse entre si de forma que, aunque una instrucción haya pasado de ejecutarse en unos pocos ciclos a hacerlo en más de 30, varias instrucciones diferentes pueden estar en ejecución al mismo tiempo en diferentes etapas, terminando una en cada ciclo y aumentando la productividad del procesador. De este modo se ha podido aumentar la frecuencia de reloj proporcionalmente con el número de etapas, obteniendo una mejora sustancial del rendimiento.

Respecto al segundo objetivo, se han desarrollado lo que se conoce como procesadores superescalares, que permiten lanzar simultáneamente a ejecución múltiples instrucciones de un mismo flujo secuencial. El lanzamiento simultáneo se consigue inspeccionando de forma dinámica bloques de instrucciones del flujo secuencial en busca de instrucciones que no sean dependientes entre sí, lanzándolas en paralelo. Esta técnica se utiliza habitualmente en combinación con el lanzamiento fuera de orden, en el que no nos limitamos a instrucciones consecutivas en nuestra búsqueda de paralelismo. De este modo podemos aprovechar el posible paralelismo existente entre el gran número de instrucciones que ejecuta el procesador. Esto es lo que se conoce como "Paralelismo a Nivel de Instrucción" o *ILP* (de sus siglas en inglés, *Instruction Level Parallelism*).

Gracias al amplio desarrollo de ambos conceptos, la segmentación y el lanzamiento superescalar de instrucciones, se ha conseguido mejorar enormemente el rendimiento de los procesadores a la vez que se mantenía la ilusión, básica para los programadores, de que los programas se ejecutan de forma secuencial y en orden, en lugar de en paralelo y fuera de orden.

Límites en el diseño tradicional de procesadores

Esta forma de diseñar procesadores se ha encontrado con una serie de límites prácticos. Para empezar, se han alcanzado los límites prácticos en la segmentación de instrucciones. No podemos seguir segmentando la ejecución de una instrucción, más allá un cierto número de etapas el trabajo que queda para cada una es ya demasiado pequeño y no merece la pena dividir más. Además, cuantas más etapas tiene el procesador, más penalizado se ve cuando ocurre un fallo de predicción, al perderse todo el trabajo que hay ya en las distintas etapas. También nos encontramos con que el paralelismo a nivel de instrucción que podemos extraer de un flujo secuencial tiene un límite [Wal91]. Tampoco hay que despreciar el coste cada vez mayor en ingeniería que supone integrar más y más transistores en la lógica central de ejecución (diseño, verificación, etc). Esto, en la práctica, ha supuesto que muy pocas empresas puedan permitirse competir en el mercado de procesadores de alto rendimiento, obligando a las empresas más pequeñas a abandonarlo.

Pero el límite fundamental lo encontramos en el consumo de energía. A pesar de que la reducción en la escala de integración hace que los transistores consuman menos, las corrientes de fuga son cada vez mayores conforme disminuye la escala de integración. Además, los ingenieros han incluido un número creciente de transistores en sus diseños, aumentando el consumo estático del chip, y gracias a la segmentación se ha aumentado la frecuencia de reloj por encima de la mejora tecnológica. Los modelos de energía predicen un aumento cuadrático (algunos incluso cúbico) del consumo dinámico con la frecuencia del procesador. Esto, sumado al aumento de corrientes de fuga y el consumo

estático, ha supuesto en la práctica un incremento exponencial en el consumo energético de los procesadores, pasando de diseños con un consumo inferior al Vatio a diseños con un consumo superior a los 100. Este mayor consumo, y además localizado en un área cada vez más pequeña, supone un grave problema de disipación de calor. Desafortunadamente, la tecnología de refrigeración no ha escalado tan bien a lo largo del tiempo, pasando de procesadores sin necesidad de ningún tipo de disipador en los 80 a procesadores que requieren disipadores muy sofisticados con ventiladores dedicados en la actualidad. No podemos aumentar más la densidad de calor en los procesadores sin la necesidad de incluir costosísimos sistemas de refrigeración. Esto ha provocado que, tal y como podemos ver en la figura 1.3, hacia 2004-2005 se produjese un estancamiento en el aumento de la frecuencia de procesamiento.

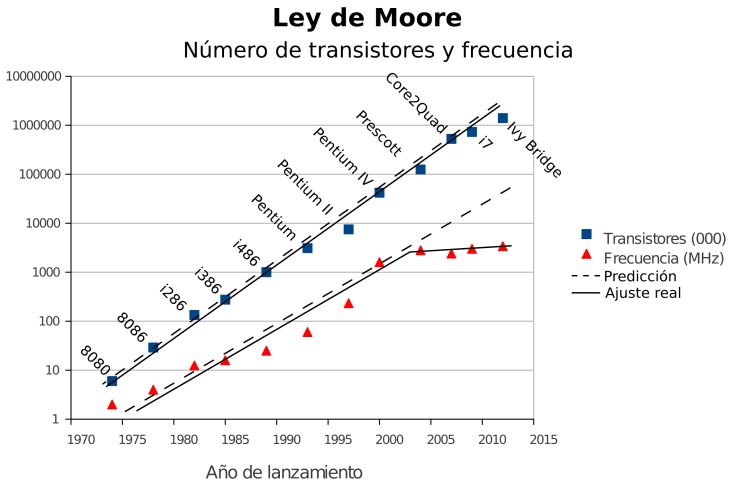


FIGURA 1.3: Predicción y ajuste a la Ley de Moore del número de transistores y la frecuencia de procesamiento.

La combinación de todos estos factores: disponibilidad de una cantidad finita de paralelismo a nivel de instrucción, límites prácticos a la segmentación,

el sobrecoste de diseños cada vez más complejos y, en especial, el problema del consumo energético, supone un límite a la mejora de rendimiento que nos ofrece la *Ley de Moore*, lo que obliga a realizar cambios drásticos en el diseño de los procesadores si queremos evitar que la tendencia en la mejora del rendimiento reduzca su velocidad de forma sustancial.

De uniprocesador a multiprocesador

En este escenario, en el que ya no compensa el esfuerzo de seguir construyendo procesadores más rápidos y más grandes, los ingenieros se han visto forzados a optar por nuevas vías en el diseño de procesadores. Diseños que permitan seguir aprovechando el creciente número de transistores disponibles a la vez que controlan el consumo del chip.

Con esta idea en mente, aparecen los nuevos *Chips Multi-Procesador* o *CMP*, diseños que integran varios núcleos de procesamiento dentro de un mismo chip. También se conocen en la industria como *multicore* o, de forma más específica, como procesadores *manycore*, cuando nos referimos a *CMP* con un gran número de procesadores simples integrados.

Han aparecido muchos y muy diferentes diseños *CMP* enfocados a todos los mercados. Tanto para servidores de alto rendimiento como para procesadores empotrados. Nos encontramos tanto con *CMP* de propósito específico con centenares de núcleos, como las unidades de procesamiento gráfico de las que hablamos en profundidad en la sección 1.2, como con procesadores de propósito general con seis núcleos como los *Intel i7*. Pero los núcleos de procesamiento no tienen por qué ser necesariamente simétricos. En el mercado encontramos *CMP* heterogéneos como el *Cell BE* de *IBM*, que incluye un procesador *Pow-*

er y 8 coprocesadores específicamente diseñados para el cómputo intensivo. La heterogeneidad se hace más habitual en los diseños empotrados, en los que se incluyen diferentes núcleos específicos para determinadas tareas, como los *chips* específicos para *smartphones* y otros dispositivos portátiles de alto rendimiento, que suelen incluir, además de un procesador central *ARM* de uno o varios núcleos, un procesador de señal digital (*DSP*) y una unidad de procesamiento gráfico. En estos entornos, en los que todo el sistema se integra dentro del mismo chip, se suelen conocer más habitualmente como Sistemas Multiprocesador en Chip o *MPSoC* (de las siglas en inglés *Multi-Processor System on Chip*).

De cara al diseñador de procesadores, el cambio supone una mayor presión sobre la memoria y los buses de entrada/salida conforme mayor es el número de procesadores integradores. Pero el escalado a lo largo de las generaciones de tecnologías de integración se puede hacer de forma sencilla y barata simplemente añadiendo más núcleos al *CMP*; y controlar el segmento del mercado al que nos queremos dirigir (precio y el rendimiento) es tan fácil como variar el número de procesadores y la velocidad de reloj.

El problema fundamental se lo encuentran los programadores. Estaban acostumbrados a ver el procesador como una arquitectura Von Neumann y ahora tienen que adaptarse a un nuevo paradigma en el que el paralelismo debe ser expuesto por el programador. Idealmente, las herramientas de desarrollo (compiladores, etc) deberían ser capaces de exponer ese paralelismo al procesador, pero a día de hoy eso sigue estando muy lejos. Los programadores necesitarán formación en programación paralela y aprender a enfrentarse a los nuevos problemas que trae asociados si no desean que el rendimiento de sus

aplicaciones se quede estancado a lo largo de las nuevas generaciones de procesadores. Esto también es un problema para los vendedores de procesadores, que verán su mercado muy reducido si no son capaces de ofrecer soluciones eficaces para que los desarrolladores puedan seguir aprovechando sus procesadores de manera eficaz.

Para facilitar la tarea de enfrentarse a estas nuevas plataformas, tanto desde la industria como desde el entorno académico se han desarrollado diferentes herramientas que tratan de exponer la arquitectura paralela y facilitar su programación, como podrían ser los casos de *CUDA* y *Brook+* [BFH+03] para la programación de unidades de procesamiento gráfico para cómputo de propósito general, *CellSs* [PBBL07] para la programación de *Cell BE* u *OpenCL* [Gro08] y, más recientemente, *OpenACC* [Ope11] para arquitecturas paralelas heterogéneas en general. Estas soluciones vendrían a unirse a soluciones de programación paralela más clásicas, como *OpenMP* o *MPI*, que no se adaptarían tan eficazmente a estas nuevas arquitecturas por ser más específicas que aquellas para las que fueron concebidas. Aún así, deben ser tomadas en cuenta junto con los esfuerzos de adaptación que se han hecho en este sentido [NJK11] [LE10].

1.2. Arquitectura de las GPUs

Entre las arquitecturas *CMP* más extendidas, nos encontramos con las unidades de procesamiento gráfico o *GPUs* (de sus siglas en inglés *Graphics Processing Units*), destinadas en sus orígenes a la síntesis de gráficos por computador. El mercado de los videojuegos, que a lo largo de la última década ha

crecido hasta superar con creces al resto de la industria del entretenimiento, ha impulsado la inversión en el desarrollo de este tipo de hardware tan específico; conforme crecía el mercado aumentaba la demanda de *GPUs* cada vez más potentes. De este modo nos encontramos una situación en la que la potencia bruta de las *GPUs* ha crecido más deprisa que la de las propias *CPUs*, y se han convertido en un elemento común en los computadores de consumo hoy en día.

Actualmente, hay *GPUs* disponibles en el mercado que alcanzan rendimientos máximos teóricos del orden de teraflops, lo que combinado con su ubicuidad las convierte en una plataforma muy interesante de cara al desarrollo de aplicaciones de alto rendimiento.

GPUs: de la síntesis de gráficos al cómputo científico

En un principio, las *GPUs* fueron diseñadas para acelerar la síntesis de imágenes 2D en tiempo real a partir de escenas representadas por geometría 3D. En *Informática Gráfica*, una escena tridimensional está representada por una geometría, luces y un punto de vista o cámara desde el cual se realizará la representación 2D de la escena (como si tomásemos una foto de la escena).

Hay varias formas de transformar una escena 3D en una imagen 2D, como *ray casting* [Rot82], *ray tracing* [App68] o *radiosity* [GTGB84]. Pero cuando nuestra intención es generar esas imágenes en tiempo real, el método utilizado es el que se conoce como *scanline rendering and rasterization*, en el que la geometría es representada por grupos de vértices que forman primitivas (como triángulos, cuadrados u otros polígonos), esos vértices se proyectan desde el punto de vista y la parte de la imagen resultante encerrada por las primitivas

proyectadas se rellena en función de ciertas propiedades contenidas en sus vértices.

Las unidades de procesamiento gráfico implementan lo que se conoce como una tubería gráfica, en la que por un extremo entra la geometría de la escena la cuál, etapa a etapa, se va transformando hasta convertirse en la imagen sintética. La tubería que implementan típicamente las tarjetas gráficas sigue el esquema que se muestra en la figura 1.4.

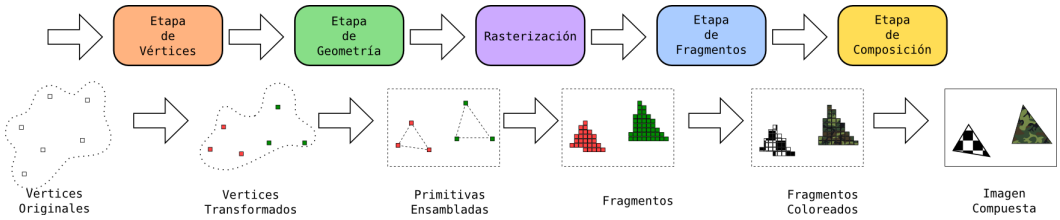


FIGURA 1.4: Tubería gráfica. Como entrada toma un conjunto de vértices y sus atributos, que se transforman en las sucesivas etapas en la imagen de la escena representada por dicha geometría.

En la tubería gráfica hay cinco etapas bien diferenciadas.

1. *Etapa de Vértices*. En esta etapa entran los vértices de la escena que queremos representar junto con los atributos asociados (color, coordenadas de textura, etc). Por norma general, estos atributos asociados a los vértices contendrán información sobre la geometría del objeto que componen (normales, tangentes, etc) y sobre el material del que está hecho el objeto (como el color y unas coordenadas de textura). Los atributos con información geométrica se utilizarán, típicamente, para calcular la forma en la que las luces de la escena iluminan el objeto. Los atributos relacionados con el material del objeto se utilizan para darle el color final. Un ejemplo de esto es la técnica que se conoce como *mapeado de*

texturas. Esta técnica consiste en extender una imagen bidimensional, la textura, sobre la superficie de un objeto, dando el aspecto de que está hecho de esa textura. Para esta técnica, cada vértice tiene asociadas unas coordenadas dentro de dicha imagen bidimensional, lo que se conoce como coordenadas de textura.

En principio, estos vértices son transformados uno a uno de manera independiente para situarlos en la escena, se calcula su iluminación respecto a las luces de la escena y se proyectan respecto a nuestro punto de vista. En general, tomamos los atributos de los vértices y los modificamos respecto a la información de la escena (luces, cámara, etc).

2. *Etapa de Geometría*. En esta etapa, los vértices agrupados por primitivas (segmentos, triángulos, cuadrados y todo tipo de polígonos) se recortan contra el marco de la zona de visualización y se ensamblan formando geometría rasterizable (por lo general, triángulos).

3. *Rasterización*. Esta es la etapa clave y la que da nombre a la técnica de síntesis de gráficos. Tomando las primitivas ensambladas en la etapa anterior, se calculan los puntos de la pantalla que quedan encerrados por ellas y se interpolan los atributos de los vértices para cada uno de esos puntos. Esos puntos se conocen como fragmentos. En el caso del mapeado de texturas, por ejemplo, cada fragmento contendrá las coordenadas de textura que le corresponderían dadas las coordenadas de textura de los vértices del polígono que contiene al fragmento.

4. *Etapa de Fragmentos*. En esta etapa, los fragmentos generados por el rasterizador son coloreados en función de los atributos que se hayan cal-

culado para cada uno de ellos. Es, por ejemplo, el caso de la técnica de mapeado de texturas. Utilizaríamos las coordenadas de textura interpoladas desde los vértices para indexar una imagen bidimensional, nuestra textura, y obtener el color base del fragmento.

De nuevo, este procesamiento se hace independientemente para cada uno de los fragmentos generados.

5. *Etapa de Composición.* En esta etapa, los fragmentos procesados se combinan entre ellos o con los que pudiera haber ya en la memoria que contiene la información de visualización, para calcular el aspecto de la escena final. Los efectos típicos que realiza esta etapa son el mezclado de canal alfa (*Alpha Blending*), para simular materiales transparentes y efectos especiales basados en transparencias; y *Anti-Aliasing*, para eliminar los artefactos debidos al submuestreo (como los bordes dentados típicos de un polígono rasterizado).

En sus orígenes, las *GPUs* implementaban la tubería gráfica como una función fija pero, en sucesivas generaciones, fueron flexibilizando las distintas etapas permitiendo que fuesen sustituidas por funciones especiales programadas por el usuario. De ese modo, la primera etapa que se hizo programable fue la etapa de procesamiento de vértices y, con la entrada de la que se conoce como cuarta generación de GPUs [FK03], también pasó a ser programable la etapa de procesamiento de fragmentos. Estas funciones se conocen como *programas de vértices* y *programas de fragmentos*, o también como *vertex shaders* y *fragment shaders*.

En la figura 1.5 se muestra el diagrama de bloques genérico de una *GPU*

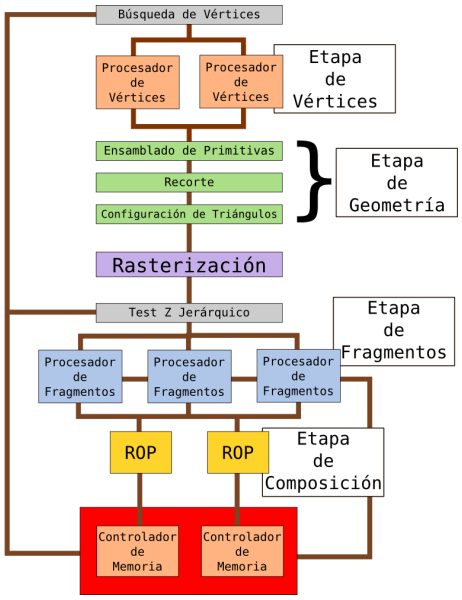


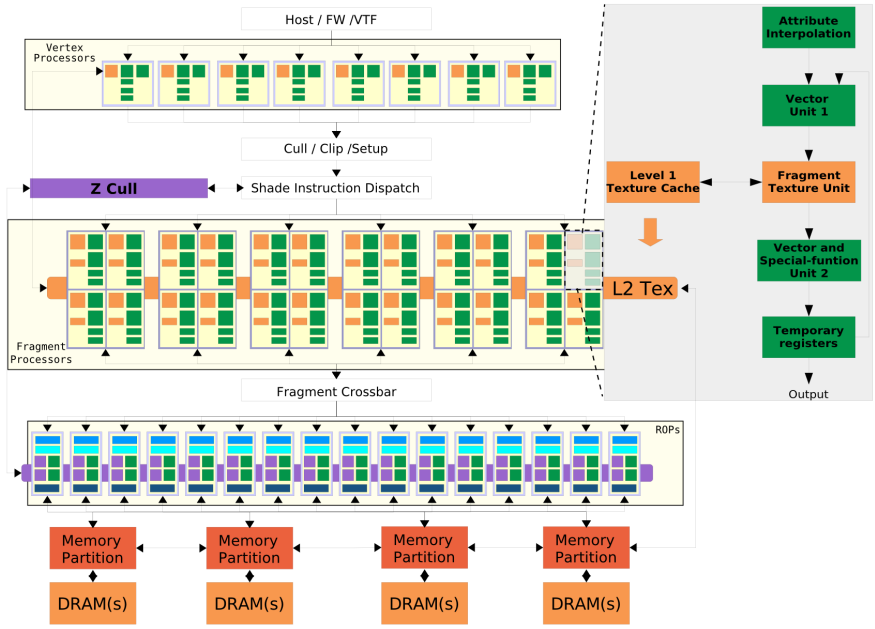
FIGURA 1.5: Diagrama de bloques de una GPU de 4ª generación.

de lo que la industria ha identificado como cuarta generación. Primero, los vértices son enviados a procesar a los procesadores de vértices, que son los encargados de realizar la etapa de vértices. Al no haber dependencias entre el procesamiento de los vértices individuales, simplemente añadiendo más procesadores de este tipo podemos mejorar el rendimiento de la etapa de vértices de forma sencilla. Una vez los vértices son procesados y se conoce su posición en la imagen final y sus atributos, la etapa de geometría se encarga de agruparlos en forma de primitivas y enviarlas al rasterizador para que calcule los fragmentos. Esos fragmentos son despachados a los procesadores de fragmentos para calcular su color en base a los atributos interpolados. De forma similar a lo que ocurre con la etapa de vértices, al ser el procesamiento de un fragmento totalmente independiente del de los demás, añadiendo más procesadores de fragmentos podemos escalar el rendimiento de la etapa de fragmentos. Lo típico en estas *GPUs*, donde los procesadores de vértices

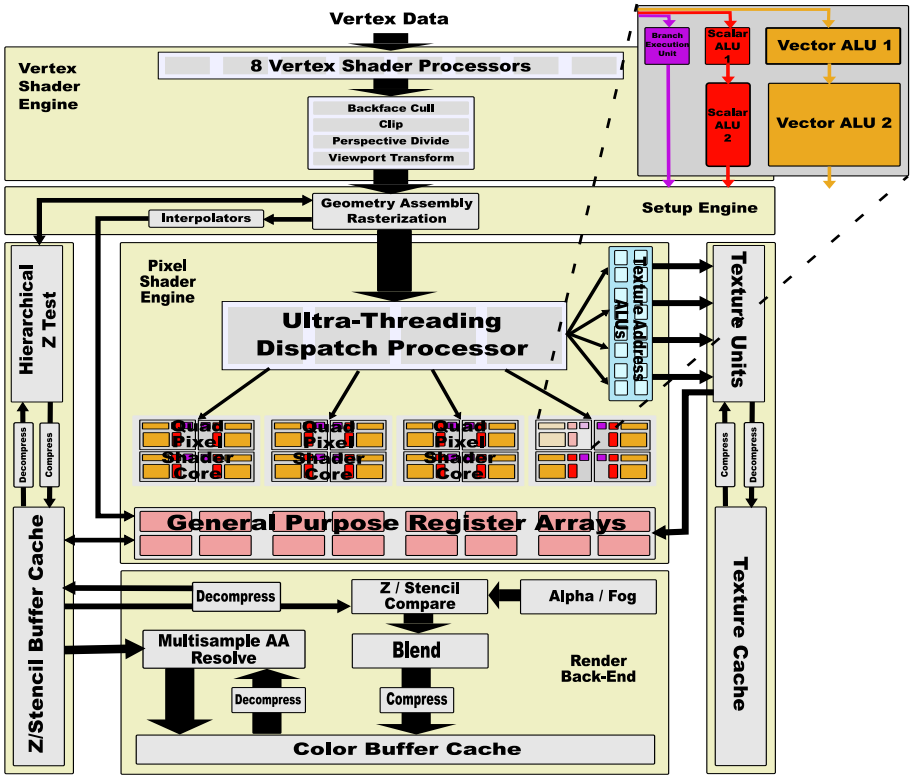
y fragmentos son diferentes, con distinta funcionalidad y distintos recursos disponibles, es disponer de más procesadores de fragmentos que de vértices. Esto es debido a que, típicamente, un pequeño grupo de vértices genera un gran número de fragmentos. Por lo tanto, para mantener la carga equilibrada son necesarios más procesadores de fragmentos que de vértices. En la etapa de composición, los fragmentos coloreados se componen para formar la imagen final en memoria. Para ello, las tarjetas disponen de una lógica dedicada, los *ROP* (Raster Operation Processors), que, si así se lo requerimos, combinarán los colores de los fragmentos ya en memoria con los nuevos fragmentos para obtener los píxeles finales.

Podemos ver el diagrama de bloques de dos arquitecturas de cuarta generación, la *G70* de *NVIDIA* y la *R520* de *ATI*, en la figura 1.6. En ella se puede apreciar cómo ambas *GPUs* tienen un diseño básico similar, con sus procesadores de vértices y fragmentos, su rasterizador, el hardware dedicado a la etapa de geometría (*Setup Engine* en la *R520* y *Cull/Clip/Setup* en la *G70*) y a la composición (*Render Back-End* para la *R520* y *ROP* para la de *NVIDIA*).

A pesar de seguir esquemas similares, entre ambas arquitecturas hay diferencias esenciales. En ambas *GPUs* podemos ver 8 procesadores de vértices, y 24 procesadores de fragmentos en la de *NVIDIA* y 16 en la de *ATI*, en ambos casos agrupados de 4 en 4. Pero podemos apreciar diferencias en la estructura de ambos tipos de procesadores. Mientras que *NVIDIA* adoptó una aproximación en la que los interpoladores estaban incluidos en los procesadores de fragmentos, *ATI* optó por dedicar un bloque de hardware separado a la tarea de interpolación. Algo similar ocurre con las unidades de texturas, acopladas



(a)



(b)

FIGURA 1.6: Diagrama de bloques de las GPUs (a) NVIDIA G70 y (b) ATI-RADEON R520.

con los procesadores de fragmentos en una *G70* y agrupadas en un gran bloque separado en la *R520*. De esa manera *ATI* mejora el rendimiento evitando que los procesadores de fragmentos queden innecesariamente bloqueados cuando la unidad de texturas está esperando datos.

En estas arquitecturas, los procesadores de vértices y fragmentos son capaces de hacer operaciones de tipo *SIMD* (*Single Instruction - Multiple Data*) con datos en coma flotante de precisión simple. Disponen de unidades vectoriales capaces de realizar una operación *MAD* (*Multiply And Add*) simultáneamente con otra operación vectorial simple. Funcionando a $430Mhz$ y teniendo en cuenta que disponemos de 24 procesadores solo en la etapa de fragmentos, vemos que el potencial de cálculo en coma flotante de una *G70* es de $124GFLOPs$ solo en esa etapa. Esto es más del doble de la capacidad de cómputo que tiene una *CPU* coetánea, lo que convertiría a la *G70* en una plataforma ideal para cómputo de alto rendimiento si fuésemos capaces de sacar partido de la etapa de fragmentos de una *GPU* como esta. En el capítulo 3 explicamos en detalle cómo se consigue esto.

GPUs de arquitectura unificada

A partir de la quinta generación de *GPUs*, la estructura interna sufre un cambio sustancial. Desparece la diferenciación entre procesadores de vértices y procesadores de fragmentos, y se sustituyen las unidades vectoriales por un número mucho mayor de unidades escalares agrupadas en un tipo especial de procesadores *VLIW* (*Very Long Instruction Word*). Estos procesadores son capaces de ejecutar en paralelo una misma instrucción de distintos hilos, algo que ha venido a llamarse *Single Instruction, Multiple-Threads* o *SIMT* para

abreviar. De este modo, se consigue mantener un número mayor de hilos en ejecución simultáneamente, lo que ayuda a compensar las altas latencias de acceso a memoria, y el equilibrado de carga entre los programas de fragmentos y los programas de vértices es óptimo en cualquier caso. Además, gracias a la generalización de la arquitectura, los nuevos procesadores escalares pueden ejecutar todo tipo de cómputos, con lo que la etapa de geometría también se vuelve programable en estas *GPUs*.

En la figura 1.7 podemos ver el aspecto que tiene la arquitectura de una de estas nuevas *GPUs*. En la de la gráfica, disponemos de 8 bloques de dos procesadores multi-núcleo con 8 cores cada uno; cada uno de ellos acoplado a un juego de unidades de búsqueda de texturas (*Texture Fetch Unit*) y a su propia memoria *cache* de primer nivel. Esto nos da un total de 128 procesadores integrados en un mismo chip.

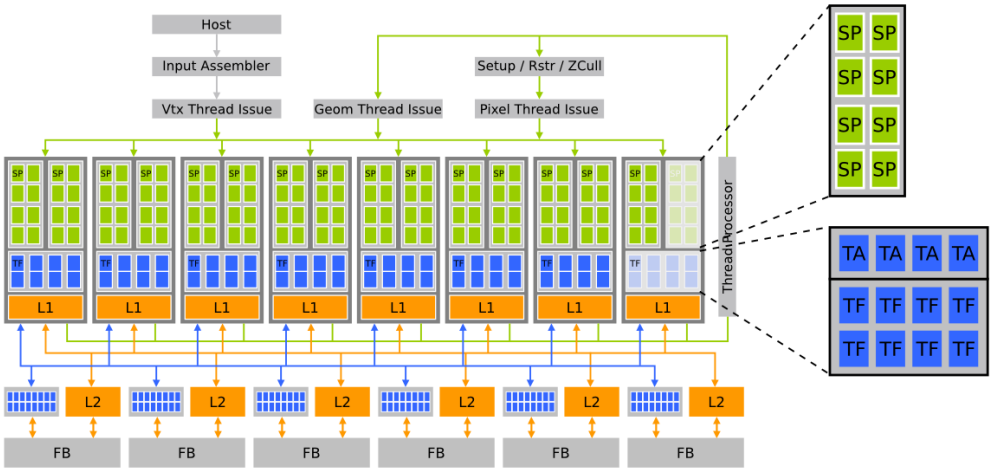


FIGURA 1.7: Diagrama de bloques de una GPU NVIDIA G80.

Típicamente, las *GPUs* operan sobre reales en punto fijo y en punto flotante de precisión simple, pero en posteriores evoluciones de las arquitecturas unifi-

cadas han aparecido *GPUs* que soportan operaciones en coma flotante de doble precisión, así como ciertas operaciones atómicas.

Esta generalización de la arquitectura, de diseños muy específicos para gráficos a un diseño *SIMT* con ciertas unidades específicas para la síntesis de gráficos (rasterizador, unidades de texturas, *ROPs*, etc), permite una mayor facilidad a la hora de utilizar la *GPU* para cómputos no dirigidos a la síntesis de gráficos. Los fabricantes, una vez reconocieron el potencial de las *GPUs* como procesadores de alto rendimiento para cálculo científico, empezaron a proporcionar herramientas y soluciones para explotar este mercado. Una de esas herramientas, propuesta por *NVIDIA*, es *CUDA* (*Compute Unified Device Architecture*). Entramos en más detalle sobre el modelo de programación de *CUDA* y sus peculiaridades en la sección 1.4 de este capítulo.

1.3. OpenGL

A principios de los 90, *Silicon Graphics* era líder en el mercado de los gráficos generados por computador. Distribuía sus estaciones de trabajo con una interfaz de programación propia, *IRIS GL API*. Gracias a su dominio del mercado, *IRIS GL* se convirtió en un estándar *de facto* en la industria; resultaba más potente, flexible y fácil de usar que los estándares del momento (fundamentalmente *PHIGS*).

Sin embargo, conforme entraron más competidores en el mercado del hardware 3D (*Sun Microsistems*, *Hewlett-Packard* e *IBM*, principalmente), con soluciones propias derivadas de *PHIGS*, *Silicon Graphics* tomó la determinación de liberar como estándar su interfaz de programación para síntesis de

gráficos como un estándar, para evitar la pérdida de mercado que la diversificación en las interfaces de programación le suponía.

FIGURA 1.8: *Rage*, de *ID Software*.

Silicon Graphics liberó la interfaz de la parte de gráficos de *IRIS GL* (dejando fuera partes más relacionadas con el sistema operativo), dando lugar a la publicación del estándar *OpenGL*.

OpenGL normaliza el acceso al hardware, cargando la responsabilidad del desarrollo de controladores del hardware a los fabricantes, y delegando la gestión de entrada/salida y las ventanas al sistema operativo. Con tantos tipos diferentes de hardware gráfico, que todos funcionasen con la misma interfaz supuso un gran apoyo para los desarrolladores dándoles una plataforma de desarrollo de alto nivel.

OpenGL funciona a más bajo nivel que otras plataformas de desarrollo para aplicaciones que requieren síntesis de gráficos 3D. Se basa en una máquina de estados y proporciona una serie de funciones que permiten cambiar el estado de la máquina y enviar geometría para representar.

Esta biblioteca está destinada a la síntesis de gráficos por *rasterización* (tal y como vimos en la sección 1.2) y, aunque es un mecanismo en apariencia sencillo, utilizando diferentes técnicas y con el control adecuado de la máquina de estados, se pueden conseguir resultados fotorrealistas como los que es costumbre ver en los videojuegos actuales (figuras 1.8 y 1.9).

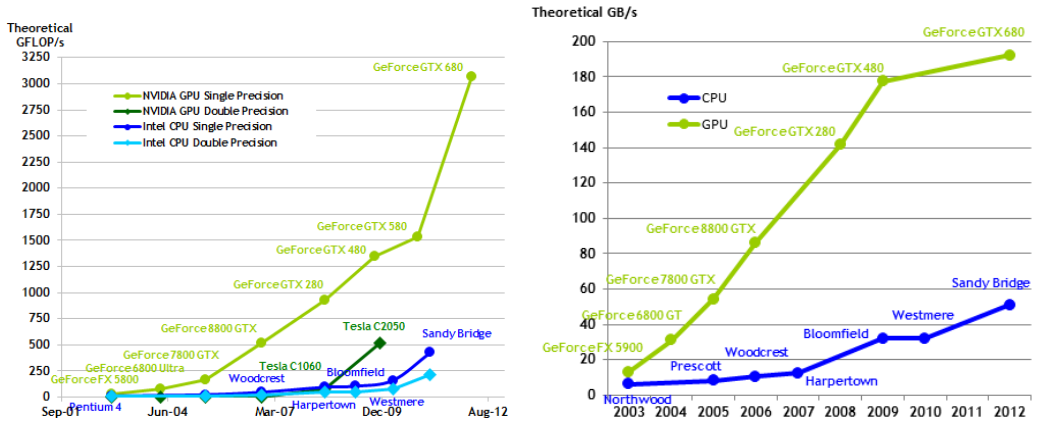
FIGURA 1.9: *Bioshock*, de *Irrational Games*.

Esa flexibilidad a la hora de manipular el sistema síntesis es la que nos va a permitir, con ciertos trucos, utilizar las *GPUs* como máquinas paralelas de propósito general.

En el apéndice B se incluye una pequeña introducción a la biblioteca *OpenGL* y la síntesis de gráficos por computador en general; y en el capítulo 3 comentamos algo más sobre cómo utilizar el sistema de síntesis propuesto por *OpenGL* para realizar cómputo de propósito general.

1.4. CUDA

Como podemos observar en la figura 1.10, tanto la capacidad de cómputo (1.10(a)) como el ancho de banda (1.10(b)) de las *GPUs* ha mejorado mucho más deprisa que la capacidad de cómputo y en ancho de banda de las *CPUs*. No solo tenemos una potencia de cálculo teórica muy superior, además esa ventaja crece con el tiempo. Este hecho, junto con el relativo bajo coste de las *GPUs* respecto a otras soluciones para el cómputo de alto rendimiento, ha llamado la atención del mundo académico y la industria, que han dedicado esfuerzos en los últimos años a portar aplicaciones y a estudiar la mejor manera de explotar las *GPUs* como máquinas de propósito general.



(a) (b)

FIGURA 1.10: La distancia en potencia de cálculo pico y ancho de banda entre *GPUs* y *CPUs* se ha ido incrementando notablemente con los años (gráficas obtenidas del manual de programación de *CUDA* [nvi12]).

Conforme fue aumentando el interés por las *GPUs* como plataforma de cómputo de propósito general, los propios fabricantes empezaron a ofrecer sus soluciones al problema: hicieron diseños más adaptables al cómputo de propósito general y desarrollaron entornos de trabajo y herramientas dirigi-

das a la programación paralela (compiladores, depuradores, bibliotecas, etc). *CUDA* (*Compute Unified Device Architecture*) es la solución propuesta por *NVIDIA*.

CUDA está basado en un modelo de programación multihilo masivo en el que el programador expone el paralelismo de datos del programa. La gestión de hilos se realiza mediante hardware dedicado en los dispositivos *CUDA*, de forma que nos ahorramos la sobrecarga inherente a la creación, planificación y destrucción de los centenares de miles de hilos que puede acarrear una aplicación típica para *CUDA*.

CUDA está diseñado para ser escalable, de forma que el mismo modelo y los mismos programas puedan servir en futuras generaciones, conforme se añadan más recursos y núcleos de cómputo a las *GPUs* que lo soportan.

Comparado con *OpenGL*, *CUDA* es prácticamente ajeno a la síntesis de gráficos y está totalmente orientado a programadores especializados en programación paralela; sin embargo, parte del hardware específico para síntesis de gráficos también está disponible desde *CUDA*, con lo cuál también podemos aprovechar algunas de las ventajas de ciertos elementos de síntesis de gráficos, como las unidades de texturas. Además, el modelo de *CUDA* está mucho más próximo al hardware real subyacente, permitiendo explotar mejor las *GPUs* como *CMPs* de propósito general aunque, para ello, el programador deba enfrentarse a un espacio de posibles optimizaciones muy amplio. Esto supone, en general, la búsqueda de estrategias de transformación del código más inteligentes, de manera que las nuevas oportunidades de optimización que nos brinda *CUDA* queden expuestas, pudiendo ser aprovechadas en la búsqueda de una configuración óptima.

En el capítulo 4.1 se hace una pequeña introducción a los detalles del funcionamiento y la programación de aplicaciones paralelas en *CUDA*.

1.5. Imágenes Hiperespectrales

Llamamos imagen hiperespectral a aquella cuya resolución en frecuencia es muy superior a las imágenes normales. En una imagen a color clásica encontramos tres bandas espectrales: rojo, con una longitud de onda entre los 625 y los 740nm; verde, con una longitud de onda entre los 520 y los 570nm; y azul, con una longitud de onda entre los 440 y los 490nm. Por el contrario, en una imagen hiperespectral podemos llegar a encontrar desde cientos hasta miles de bandas espectrales, con longitudes de onda que varían desde el ultravioleta (menos de 400nm) hasta el infrarrojo (más de 750nm). Es típico que las bandas estén uniformemente distribuidas entre todo el espectro que abarca el sensor, diferencia esencial con las imágenes multiespectrales, que además de disponer solamente de alrededor de una decena de bandas, suelen captar longitudes de onda más dirigidas [VGC$^+$93, KLC04] (rojo, verde, azul, infrarrojo cercano, infrarrojo medio y radar, por ejemplo).

Esa resolución espectral tan superior frente a las típicas imágenes en color, supone que los píxeles de las imágenes hiperespectrales tengan cientos de componentes frente a los típicos tres de las imágenes en color normales. Por ello, cuando hablamos de los píxeles de las imágenes hiperespectrales, solemos referirnos a los vectores-píxel o, de una forma más genérica, la firma espectral de un punto de la imagen. Seguiremos esta terminología de aquí en adelante.

Gracias a la elevada resolución espectral de las imágenes hiperespectrales,

podemos realizar análisis espectrográficos sobre los vectores-píxel. Esto puede ser útil para muchísimas aplicaciones tanto civiles como militares: control del crecimiento de cultivos, monitorización medioambiental, explotaciones mineras al aire libre, detección e identificación de estructuras hechas por el hombre, vigilancia, etc [SM02, SB03].

Tras todas estas aplicaciones subyace el hecho de que todas las substancias dispersan radiación electromagnética en longitudes de onda específicas siguiendo patrones distintivos en función de su composición. Con la información espectral adecuada, podemos distinguir e identificar dichas substancias (ver figura 1.11).

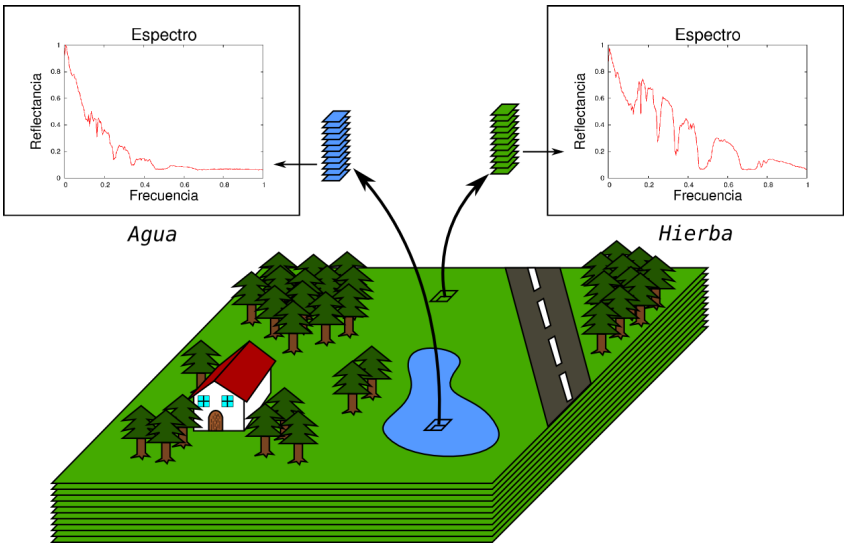


FIGURA 1.11: La resolución espectral en una imagen hiperespectral nos permite identificar los materiales de la imagen en base a su espectro.

1.6. Motivación

A pesar del potencial de las imágenes hiperespectrales, la cantidad masiva de datos que puede llegar a contener una de estas imágenes restringe su ámbito de uso a aplicaciones que no tengan requisitos de tiempo real ni de espacio de almacenamiento. Si, por ejemplo, trabajásemos con imágenes con una resolución de $1m^2$ por píxel, la foto hiperespectral de una región de $4km^2$ con una resolución espectral de 256 bandas, podría suponer tener que trabajar con imágenes de casi $2GB$ cada una. Además, el procesamiento de esa cantidad de datos requeriría una potencia de cómputo muy importante, lo que supondría, con una aproximación clásica, la necesidad de máquinas muy grandes con un gran consumo energético y/o una cantidad ingente de tiempo de cómputo.

Sin embargo, hay ciertas aplicaciones para las cuales podría ser muy interesante evitar estas limitaciones. Por ejemplo, el seguimiento de riesgos medioambientales. Un vertido de petróleo en el océano, al flotar a unos pocos centímetros por debajo de la superficie del agua, es invisible a una cámara tradicional. Sin embargo, en una imagen hiperespectral es evidente su presencia. Algo parecido ocurre con los incendios forestales que se mueven a ras de tierra, las copas de los árboles cubren el incendio y lo hacen difícil de percibir desde el aíre. Un sensor hiperespectral, sin embargo, permitiría visualizar dicho incendio. El problema es que en ambos casos necesitamos una respuesta rápida; una imagen procesada que nos mostrase la evolución o el estado de este tipo de fenómenos, que sufren cambios en periodos cortos de tiempo, no sería útil si hemos tardado varias horas (o incluso días) en obtenerla.

Otros ejemplos en los que el gran tamaño de las imágenes puede suponer

un problema, es la teledetección por satélite. Los satélites disponen tanto de un espacio limitado de almacenamiento y transmisión de datos, como de un consumo de energía limitado. Si cada imagen tomada puede ocupar del orden de gigabytes, un satélite no podría tomar demasiadas de estas imágenes cada vez. Este problema también puede surgir al embarcar un sistema de procesamiento en cierto tipo de aviones pequeño o incluso no tripulados. Sin embargo, una imagen hiperespectral ya procesada puede ocupar un espacio varios órdenes de magnitud menor, con lo que se aceleraría la transmisión de los resultados y se posibilitarían misiones más largas.

Para esquivar estos problemas vamos a necesitar una capacidad de cómputo elevada (que permita el procesamiento en tiempo real de las imágenes) y vamos a necesitar que lo haga con ciertas restricciones de espacio y consumo de energía.

En este punto es donde entran las *GPUs*. Una *GPU* proporciona una cantidad muy importante de potencia de cómputo en un volumen y con un consumo energético muy reducido en comparación a las soluciones más clásicas, como los grandes *clusters* computacionales. Además, gracias a su ubicuidad, debida al gran momento que disfruta el mercado de consumo de videojuegos, podemos disponer de estos sistemas a un precio muy inferior a casi cualquier otra solución de computación de altas prestaciones.

1.7. Objetivos

Nuestro objetivo principal es valorar la viabilidad de estas plataformas como soluciones de cómputo embarcable que nos proporcionen análisis de

imágenes hiperespectrales en tiempo real, abriendo todo un abanico de posibles aplicaciones que hasta ahora estaban vetadas.

Los objetivos de esta tesis serán:

- Analizar el paralelismo existente el algoritmo de extracción automática de *end-members AMEE* [PMPP02] con dos distancias *SID* y *SAM*.
- Desarrollar un entorno de trabajo (o *framework*) para programar *GPUs* siguiendo un *modelo de procesamiento de flujos* con *OpenGL* como biblioteca subyacente para acceso al hardware de las *GPUs*.
- Portar los algoritmos de análisis de imágenes hiperespectrales al modelo de procesamiento de flujos, e implementar dichos algoritmos de manera eficiente en *GPUs* utilizando la biblioteca de desarrollo implementada para tal propósito.
- Portar los algoritmos de análisis de imágenes hiperespectrales al entorno de desarrollo *CUDA* y estudiar cuál es la mejor aproximación posible.
- Estudiar el rendimiento de dichos algoritmos sobre ambas plataformas, *CUDA* y el *framework* para *GPGPU*, y valorarlos como posible solución al análisis de imágenes hiperespectrales en tiempo real.

El resto de esta tesis está organizada de la siguiente manera:

En el capítulo 2, se presentan los algoritmos con los que trabajamos y se realiza un análisis de su idoneidad para la tarea. En el capítulo 3 se presenta nuestro entorno de trabajo sobre *OpenGL* y se aborda la implementación de los algoritmos de análisis de imágenes hiperespectrales sobre él. En el capítulo 4 se estudia la implementación de nuestros algoritmos en *CUDA*. Y al final, en

el capítulo 5, presentamos nuestras conclusiones, las principales aportaciones de este trabajo e ideas para un posible trabajo futuro.

Capítulo 2

Procesamiento de Imágenes Hiperespectrales

Como ya se comentó en la introducción, las imágenes hiperespectrales son útiles para realizar análisis espectrográficos. Gracias a la resolución espectral de las imágenes, podemos determinar los materiales de los que están compuestos los píxeles de nuestra imagen. El proceso principal, presente en las principales aplicaciones de análisis de imágenes hiperespectrales, es encontrar los vectores-píxel que tengan el espectro más puro, conocidos como *end-members*

Las principales aplicaciones que se realizan en el análisis de imágenes hiperespectrales son las tres siguientes:

1. *Identificación:* Comparamos la firma espectral de los *end-members* con una base de datos de espectros de materiales, identificando el material concreto de cada uno de los *end-members*.

2. *Desmezclado:* Descomponemos cada uno de los vectores-píxel de la imagen en una combinación, por lo general lineal, de los *end-members*.

3. *Clasificación:* Localizamos regiones en la imagen que pertenezcan a la clase de un *end-member*, de forma que delimitamos zonas en las que predomina claramente un material.

La búsqueda de los *end-members* no solo va a ser fundamental para casi cualquier aplicación de análisis de imágenes hiperespectrales, además va a ser la que más tiempo consuma. Para encontrar los vectores-píxel más puros de una imagen tenemos que, de alguna manera, compararlos todos entre si; lo cual, dado el tamaño de las imágenes hiperespectrales, supone una gran cantidad de cómputos.

Los métodos para la búsqueda de *end-members* se pueden clasificar en dos grupos:

1. Supervisados: La aplicación requiere de algún tipo de intervención externa o control por parte del usuario.

2. No supervisados: La aplicación realiza el análisis a partir de los datos del sensor, sin ninguna intervención por parte del usuario.

En una aplicación que requiera análisis de imágenes hiperespectrales en tiempo real, un algoritmo supervisado no sería útil; necesitamos un algoritmo que funcione de manera autónoma, es decir, un algoritmo de búsqueda de *end-members* no supervisado.

El algoritmo *AMEE* (del inglés *Automated Morphological End-member Extraction*, o *Extracción de* end-members *morfológica automatizada*), además de

cumplir el requisito de no estar supervisado, fue diseñado para ser computacionalmente eficiente [PMPP02]. Es un algoritmo con mucho paralelismo y que puede, por tanto, aprovechar las capacidades de paralelismo masivo que tienen nuestras plataformas de estudio.

Otra característica distintiva de este algoritmo es el uso de la morfología matemática para realizar su análisis. Existen métodos que hacen un análisis basado en la información espectral de los vectores-píxel, mientras que hay otros que realizan un análisis basado en información espacial. Sin embargo, *AMEE* utiliza la morfología matemática para correlacionar la información espacial y espectral, y aprovecharlas simultáneamente para realizar la búsqueda.

En cuanto a precisión, en la sección 2.1.3 se muestra cómo *AMEE* es superior a otros algoritmos de detección de *end-members* en la práctica totalidad de los casos.

Además, con ciertas ampliaciones (explicadas en la sección 2.1.1) podemos elegir ciertos parámetros que nos permiten obtener un compromiso entre rendimiento y precisión, lo cuál puede resultarnos muy útil en escenarios de tiempo real.

Por todas estas características (eficiencia, paralelismo, precisión y la posibilidad de ajustar el compromiso entre precisión y rendimiento), hemos optado por centrarnos en la implementación y optimización del algoritmo *AMEE* sobre *GPUs* para la búsqueda de *end-members*, que es lo que supone la gran mayoría del coste computacional del análisis de imágenes hiperespectrales.

2.1. El Algoritmo *AMEE*

Definimos $\boldsymbol{f}$ como un conjunto de datos hiperespectrales en un espacio n-dimensional ($\mathbb{K}^n$), donde n es el número de canales o bandas espectrales:

$$\boldsymbol{f}: \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{K}^n$$

La idea principal detrás del algoritmo *AMEE* es definir una relación de orden, en términos de pureza espectral, en el conjunto de vectores-píxel dentro de una ventana espacial de búsqueda -o elemento estructural- alrededor de cada vector-píxel de la imagen [Soi03].

Con ese propósito, primero definimos la distancia acumulada entre un píxel particular $\boldsymbol{f}(x, y)$, es decir, un vector $\mathbb{K}^n$ en las coordenadas espaciales discretas (x, y), y todos los vectores-píxel en la vecindad espacial definida por B (B-vecindad) tal y como expresa la siguiente ecuación [PMPP02]:

$$D_B(\boldsymbol{f}(x, y)) = \sum_{(i,j)\in\mathbb{Z}^2(B)} Dist(\boldsymbol{f}(x, y), \boldsymbol{f}(i, j)) \tag{2.1}$$

donde (i, j) son las coordenadas espaciales en el dominio discreto de la B-vecindad, representado por $\mathbb{Z}^2(B)$, y *Dist* es una medida de la distancia entre dos vectores $\mathbb{K}^n$. Más adelante, en la sección 2.1.2, se habla en detalle sobre la elección de la función *Dist*, ya que es el elemento clave que define la relación de orden obtenida y, por tanto, la precisión del algoritmo.

La distancia acumulada se utiliza, dentro del elemento estructural, como una medida de lo representativo que es un vector-píxel del conjunto contenido por dicho elemento estructural. De este modo podemos elegir el más repre-

sentativo y el menos representativo del conjunto (el de menor y el de mayor distancia acumulada, respectivamente) para calcular el índice de excentricidad morfológica o *MEI* (del inglés *Morphological Eccentricity Index*) del vector-píxel en el que se centre el elemento estructural.

A partir de la distancia acumulada, definimos la erosión morfológica como:

$$(f \ominus B)(x,y) = argmin_{(i,j)\in\mathbb{Z}^2(B)}\{D_B[f(x+i,y+j)]\} \tag{2.2}$$

donde el operador $argmin_{(i,j)\mathbb{Z}^2(B)}$ selecciona, dentro del elemento estructural, el vector-píxel con la menor distancia acumulada.

Y de igual forma, definimos la dilatación morfológica como:

$$(f \oplus B)(x,y) = argmax_{(i,j)\in\mathbb{Z}^2(B)}\{D_B[f(x+i,y+j)]\} \tag{2.3}$$

donde el operador $argmax_{(i,j)\in\mathbb{Z}^2(B)}$ selecciona, dentro del elemento estructural, el vector-píxel con la mayor distancia acumulada.

Así pues, a partir de las ecuaciones 2.2 y 2.3, y con la ayuda de la función de distancia *Dist*, podemos definir el *índice de excentricidad morfológica* (*MEI*) de un vector-píxel como el gradiente morfológico extendido de la imagen en la posición de dicho vector-píxel:

$$\begin{aligned} MEI_B(f(x,y)) &= (f \oplus B)(x,y) - (f \ominus B)(x,y) \\ &= Dist((f \ominus B)(x,y), (f \oplus B)(x,y)) \\ &= Dist(argmin_{(i,j)\in\mathbb{Z}^2(B)}\{D_B[f(x+i,y+j)]\}, \\ &\quad argmax_{(i,j)\in\mathbb{Z}^2(B)}\{D_B[f(x+i,y+j)]\}) \end{aligned} \tag{2.4}$$

El *MEI* de un vector-píxel es representativo de la pureza espectral de dicho vector-píxel, y podemos utilizarlo para encontrar el vector-píxel más puro de la imagen; es decir, nuestro primer *end-member*.

Los demás *end-members* se hallan a partir del primero, eligiendo el vector-píxel mas ortogonal al primero, luego el que sea más ortogonal con esos dos, y así sucesivamente hasta encontrar todos los *end-members*.

En este punto, el algoritmo *AMEE* para encontrar el primer *end-member* funcionaría como sigue:

Primero, para cada vector-píxel (x, y) en el dominio espacial de $\boldsymbol{f}$:

1. Centrar el elemento estructural $\mathbb{Z}^2(B)$ en (x, y).

2. Tal y como se muestra en la figura 2.1, calcular la distancia acumulada de cada vector-píxel de la B-vecindad con todos los demás vectores-píxel dentro de dicha B-vecindad.

3. Buscar los vectores-píxel con mayor y menor distancia acumulada dentro del elemento estructural. Es decir, la erosión (ecuación 2.2) y la dilatación (ecuación 2.3) de la imagen.

4. Calculamos el *MEI* de la imagen en dicho punto; es decir, distancia entre el vector-píxel de máxima distancia acumulada y el de mínima distancia acumulada (ver figura 2.2), lo que equivale al gradiente morfológico extendido (ecuación 2.4) de la imagen en (x, y).

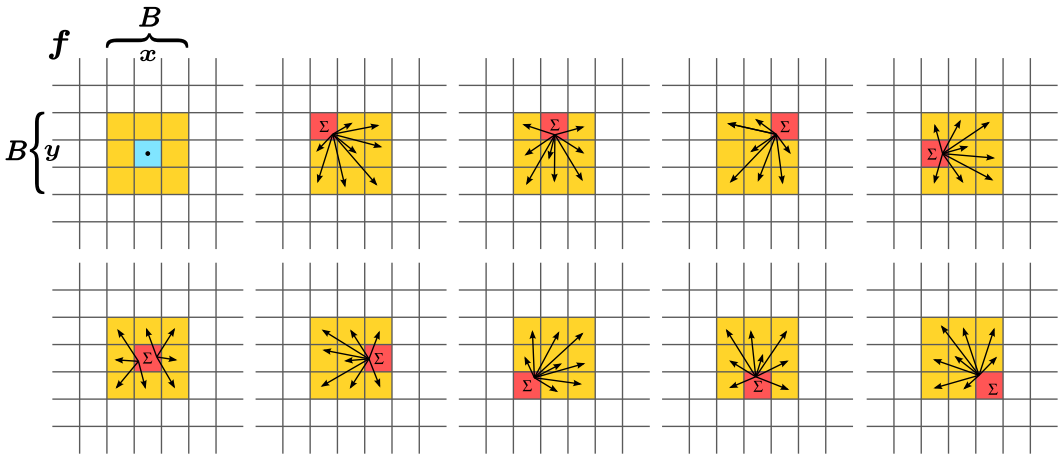


FIGURA 2.1: Cómputo de la distancia acumulada de los vectores-píxel para un elemento estructural $\mathbb{Z}^2(B)$ centrado en (x, y).

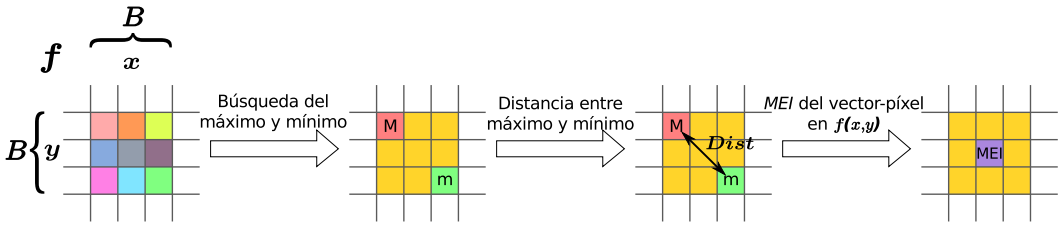


FIGURA 2.2: Búsqueda de los vectores-píxel con la máxima y mínima distancia acumulada, y cómputo del valor *MEI* para el vector-píxel $\boldsymbol{f}(x, y)$.

Una vez calculado el mapa *MEI*, para encontrar los *end-members*, buscamos entre los vectores-píxel con el valor *MEI* asociado más alto (los más puros), los q que sean más ortogonales entre sí. Añadimos el vector-píxel de *MEI* asociado más alto al conjunto de nuestros *end-members*, e iterativamente vamos añadiendo el siguiente vector-píxel con la proyección más ortogonal respecto al conjunto, hasta formar un conjunto espectral único de $e_{i\,i=1}^{q}$ vectores-píxel.

De esta descripción del algoritmo podemos obtener dos conclusiones:

1. El tamaño del elemento estructural determina, por un lado, la precisión

de nuestro algoritmo; y por otro lado, el rendimiento del mismo. Cuanto mayor sea el elemento estructural, mayor será la precisión, al haberse comparado cada vector-píxel con muchos más en el cálculo del mapa *MEI*. Por contra, cuanto mayor sea el elemento estructural, menor será el rendimiento de nuestra aplicación, ya que el número de operaciones necesario para construir nuestro mapa *MEI* crece del orden de la potencia cuarta con el tamaño de B-vecindad elegido. Para un coste K_λ para el cómputo de *Dist* entre dos vectores-píxel con λ bandas espectrales, el coste de calcular el mapa *MEI* de una imagen de $W \times H$ estaría en el orden de: $K_\lambda \cdot W \cdot H \cdot (B^4 - B^2 + 1)$.

2. El cómputo del mapa *MEI* inicial, y por tanto la búsqueda del primer *end-member*, es lo que va a consumir la mayor parte del tiempo de cómputo. En el caso anterior de la imagen de $W \times H$, la búsqueda de λ-*end-members* consumiría del orden de $K_\lambda \cdot W \cdot H \cdot (B^4 - B^2 + 1)$ para el primero, y a lo sumo $(n-1) \cdot K_\lambda \cdot W \cdot H$ para todos los demás. Por lo general, incluso para los valores más bajos de B, $(B^4 - B^2 + 1) \gg (n-1)$; por lo que el tiempo de cómputo para encontrar el primer *end-member* es significativamente superior al tiempo de cómputo para encontrar todos los demás.

Debido a la segunda conclusión, a lo largo de esta tesis nos centramos en el cálculo del mapa *MEI*, ya que es algorítmicamente lo más complejo y va a consumir casi la totalidad del tiempo de búsqueda de *end-members*.

2.1.1. *AMEE*: Cálculo multi-pasada del mapa *MEI*

Como ya hemos visto, la elección del tamaño del elemento estructural va a influir definitivamente tanto en el rendimiento como en la precisión de la búsqueda. Por eso es interesante plantearse mejoras al algoritmo que puedan mejorar su eficiencia.

Se puede comprobar experimentalmente que el mapa MEI_B, obtenido con un elemento estructural $\mathbb{Z}^2(B)$, tiene una precisión similar al que podemos obtener acumulando al mapa MEI_3 los mapas MEI_3 de las $\frac{B-3}{2}$ sucesivas dilataciones morfológicas de la imagen original.

De este modo, si definimos la dilatación i-ésima de $\boldsymbol{f}$ como:

$$(f \oplus B)^i = f \overbrace{\oplus B \cdots \oplus B}^{i}$$

Podemos definir *MEI* multi-pasada como:

$$MEI_B(\boldsymbol{f}) \approxeq \sum_{i=0}^{\frac{B-3}{2}} MEI_3((f \oplus 3)^i) \tag{2.5}$$

Así pues, habría que añadir al algoritmo, tras el cálculo de un primer mapa MEI_3, una etapa en la que calculamos la dilatación morfológica de la imagen, acumulando su MEI_3 y repitiendo la operación hasta llegar a la $\frac{B-3}{2}$-ésima dilatación. Al final, cambiamos un algoritmo en el que hay que realizar un mapa MEI_B por un algoritmo en el que calculamos $\frac{B-3}{2} + 1$ mapas MEI_3.

Si analizamos el rendimiento de cada versión, por un lado tendríamos que

el coste de calcular un mapa MEI_B, estaría en el orden de:

$$K_\lambda \cdot W \cdot H \cdot (B^4 - B^2 + 1)$$

Siendo K_λ el coste de calcular $Dist$ para dos vectores-píxel con λ bandas espectrales, y $W \times H$ las dimensiones de la imagen.

Por contra, el coste de calcular un mapa MEI equivalente al mapa MEI_B acumulando varios mapas MEI_3 estaría en el orden de:

$$(\frac{B-3}{2} + 1) \cdot K_\lambda \cdot W \cdot H \cdot (3^4 - 3^2 + 1)$$

En la figura 2.3 podemos ver la aceleración, o *speed-up*, obtenida para diferentes valores de B. Como se puede observar, crece de forma cúbica con el tamaño del elemento estructural.

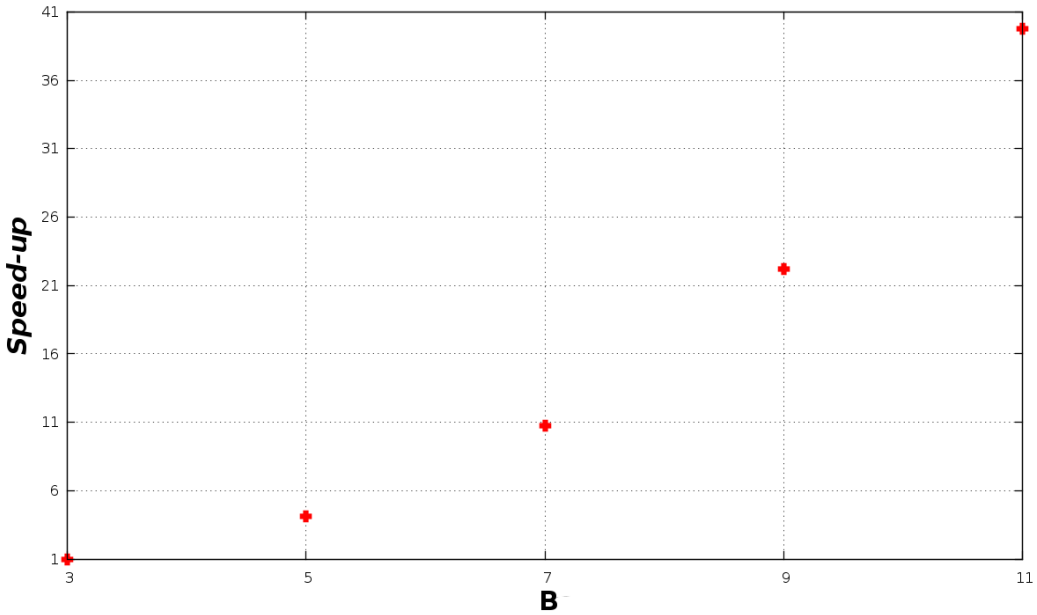


FIGURA 2.3: Comparación del rendimiento de *AMEE* multi-pasada con *AMEE* original

2.1.2. La función de cálculo de distancia

Como ya hemos comentado, la elección de *Dist* es un tema clave para la relación de orden obtenida. El algoritmo *AMEE* hace uso de la distancia angular o *SAM* (del inglés *Spectral Angle Mapper*) y la divergencia de información espectral o *SID* (del inglés *Spectral Information Divergence*), ambas medidas normales en el análisis de imágenes hiperespectrales [Cha03].

***SAM*: La distancia angular como medida de distancia**

Para simplificar, asumamos que $\boldsymbol{s}_i = (s_{i1}, s_{i2}, \cdots, s_{in})^T$ y $\boldsymbol{s}_j = (s_{j1}, s_{j2}, \cdots, s_{jn})^T$ son dos firmas espectrales en $\mathbb{K}^n$. Aquí, el término "firma espectral" no implica necesariamente "vector-píxel", de ahí que hayamos omitido las coordenadas espaciales de casos anteriores aunque el razonamiento sería el mismo si considerásemos vectores-píxel. La *SAM* entre $\boldsymbol{s}_i$ y $\boldsymbol{s}_j$ vendría dada por:

$$SAM(s_i, s_j) = cos^{-1}(\frac{s_i \cdot s_j}{\|s_i\| \, \|s_j\|}) \tag{2.6}$$

Nótese que *SAM* es invariante a factores multiplicativos constantes de los vectores de entrada y, por lo tanto, es invariante a factores de escala desconocidos que puedan surgir debido a diferencias en la iluminación y el ángulo de observación del sensor.

La distancia *SAM* nos va a dar la distancia angular entre dos firmas espectrales en el espacio $\mathbb{K}^n$

SID: La divergencia en la información espectral como distancia

La distancia *SID* está basada en el concepto de divergencia, y mide la discrepancia de comportamientos probabilísticos entre dos firmas espectrales. Dadas nuestras dos firmas espectrales $\boldsymbol{s}_i = (s_{i1}, s_{i2}, \cdots, s_{in})^T$ y $\boldsymbol{s}_j = (s_{j1}, s_{j2}, \cdots, s_{jn})^T$ en $\mathbb{K}^n$, si asumimos que $\boldsymbol{s}_i$ y $\boldsymbol{s}_j$ son entradas no negativas, entonces se pueden definir dos medidas probabilísticas de la siguiente manera:

$$\begin{aligned} M[s_{ik}] = p_k &= \frac{s_{ik}}{\sum_{l=1}^{n} s_{il}} \\ M[s_{jk}] = q_k &= \frac{s_{jk}}{\sum_{l=1}^{n} s_{jl}} \end{aligned} \tag{2.7}$$

Utilizando esta definición, la auto-información que proporciona s_j para la banda l nos la da $I_l(s_j) = -log\ q_l$. A partir de ahí podemos definir la entropía de s_j con respecto a s_i como:

$$\begin{aligned} D(s_i \| s_j) &= \sum_{l=1}^{n} p_l\ D_l(s_i \| s_j) \\ &= \sum_{l=1}^{n} p_l (I_l(s_j) - I_l(s_i)) \\ &= \sum_{l=1}^{n} p_l\ log \frac{p_l}{q_l} \end{aligned} \tag{2.8}$$

Utilizando la ecuación (2.8), definimos *SID* como:

$$\begin{aligned} SID(s_i, s_j) &= D(s_i \| s_j) + D(s_j \| s_i) \\ &= \sum_{l=1}^{n} p_l \; log\frac{p_l}{q_l} + \sum_{l=1}^{n} q_l \; log\frac{q_l}{p_l} \end{aligned} \tag{2.9}$$

Intuitivamente, *SID* nos da una medida de la entropía que hay entre dos vectores-píxel; o lo que es lo mismo, la diferencia entre la "cantidad" de información espectral de un vector-píxel respecto al otro. A mayor diferencia, más información "nueva" añade un vector-píxel respecto al otro. A menor diferencia, más representativo de ambos es cualquiera de los vectores-píxel.

2.1.3. Precisión

Como referencia para la prueba de precisión, se ha tomado la imagen "*cuprite*", disponible en la web de *AVIRIS* [NASA].

TABLA 2.1: Puntuaciones de similaridad para *AMEE SAM* y *AMEE SID* con diferente número de pasadas. Comparación con los espectros de minerales y *end-members* proporcionados por el *USGS*

	SAM AMEE			**SID AMEE**		
	$I_{max} = 1$	$I_{max} = 3$	$I_{max} = 5$	$I_{max} = 1$	$I_{max} = 3$	$I_{max} = 5$
Alunita	0.084	0.081	0.079	0.081	0.081	0.079
Buddingtonita	0.112	0.086	0.081	0.103	0.084	0.082
Calcita	0.106	0.102	0.093	0.101	0.095	0.090
Clorito	0.122	0.110	0.096	0.112	0.106	0.084
Caolín	0.136	0.136	0.106	0.136	0.136	0.102
Jarosita	0.115	0.103	0.094	0.108	0.103	0.094
Montmorillonita	0.108	0.105	0.101	0.102	0.099	0.092
Moscovita	0.109	0.099	0.092	0.109	0.095	0.078
Nontronite	0.101	0.095	0.090	0.101	0.092	0.085
Pirofilita	0.098	0.092	0.079	0.095	0.086	0.071

TABLA 2.2: Puntuaciones de similaridad de distintos métodos con respecto a los espectros de minerales y *end-members* proporcionados por la *USGS* (Servicio Geológico de los EEUU, del inglés *United States Geological Survey*)

	PPI	**N-FINDR**	**VCA**	**IEA**
Alunita	0.084	0.081	0.084	0.084
Buddingtonita	0.106	0.084	0.112	0.094
Calcita	0.105	0.105	0.093	0.110
Clorito	0.125	0.136	0.096	0.096
Caolín	0.136	0.152	0.134	0.134
Jarosita	0.112	0.102	0.112	0.108
Montmorillonita	0.106	0.089	0.120	0.096
Moscovita	0.108	0.094	0.105	0.106
Nontronita	0.102	0.099	0.099	0.099
Pirofilita	0.094	0.090	0.112	0.090

En la tabla 2.1 podemos ver los resultados de similaridad obtenidos por *AMEE*, en sus dos variantes (*SID* y *SAM*) para 1, 3 y 5 pasadas. Por otro lado, en la tabla 2.2 podemos observar los resultados obtenidos por otros cuatro métodos (*PPI*, *N-FINDR*, *VCA* y *IEA*) para la misma escena. Utilizando la distancia *SAM*, podemos ver que tras la quinta pasada solamente obtenemos menos precisión en la detección de la *montmorillonita*, donde *N-FINDER* es un poco mejor. Con todos los demás materiales, *AMEE* se muestra superior en precisión. Esto se debe a que los otros métodos utilizan únicamente información espectral en la clasificación, mientras que *AMEE* utiliza también la correlación espacial.

Con la distancia *SID*, aunque los resultados no son muy diferentes de los obtenidos con *SAM* sí se puede apreciar una cierta mejora. Esto es debido, probablemente, a que el sentido físico de la divergencia de información espectral (*SID*), es más representativo del problema de la distancia que queremos resolver; y aunque la distancia angular (*SAM*) también es una medida acept-

able, su sentido físico no se ajusta a lo que queremos medir. Por ejemplo, cuando estamos midiendo las distancias acumuladas, uno de los vectores-píxel podría tener acumulados 720 grados y otro 360 y, aunque físicamente sería lo mismo, en nuestro caso interpretamos como superior la distancia de 720 grados. Aunque no resulte válida como espacio métrico, se ha comprobado experimentalmente que la distancia es válida si se interpreta de esta manera. Pero *SID* tiene más sentido físico, lo que explicaría los resultados de precisión superiores obtenidos.

2.2. Conclusión

A lo largo de este capítulo hemos presentado y descrito un algoritmo de búsqueda de *end-members*, hemos comentado sus características principales y hemos justificado, en base a dichas características, su elección como algoritmo de estudio.

En el capítulo 3, el siguiente, exponemos los fundamentos para la programación de *GPUs* para aplicaciones de propósito general utilizando como base la biblioteca de síntesis de gráficos *OpenGL*. También describimos el entorno de desarrollo que hemos construido así como la implementación de nuestros algoritmos sobre dicho entorno.

Capítulo 3

Procesamiento de Imágenes Hiperespectrales con *OpenGL*

OpenGL es una biblioteca destinada a la síntesis de gráficos. Una buena forma de aprovechar la potencia de las tarjetas gráficas para cómputo de propósito general, es tratar la *GPU* como un conjunto de procesadores de flujo y construir una biblioteca de desarrollo sobre *OpenGL* que aproveche el modelo de procesamiento de flujos.

A lo largo de este capítulo explicamos en qué consiste el modelo de procesamiento de flujos y cómo hemos aprovechado las herramientas que ofrece la biblioteca *OpenGL* para construir una biblioteca de procesamiento de flujos basado en ella.

Seguidamente, detallamos la implementación del algoritmo *AMEE* (tanto con distancia *SID* como *SAM*) sobre esta biblioteca, y exponemos los resultados obtenidos en *GPUs* (tanto en rendimiento como en precisión) en comparación con *CPUs* coetáneas.

Al final del capítulo, en las secciones 3.4 y 3.5, mostramos los resultados y algunas conclusiones que se pueden extraer de nuestro trabajo con *OpenGL*.

3.1. El Modelo de Procesamiento de Flujos

A la hora de definir el concepto de programa, es común hacerlo como "el conjunto de datos y de algoritmos necesarios para llevar a cabo una determinada tarea". En el *modelo de procesamiento de flujos*, nuestros datos vendrían encapsulados en *flujos* (o *streams*), que son conjuntos ordenados de datos, y nuestros algoritmos en *kernels*, que son un tipo especial de función que toma como entrada elementos individuales de esos *flujos*.

Una función *kernel* puede tener uno o más *flujos* de entrada, y uno o más *flujos* de salida. El *kernel* procesa los elementos individuales de los *flujos* de entrada por tuplas, de forma que nos encontramos con un bucle implícito en el que se ejecuta el mismo *kernel* en sucesivas instancias, cada una de ellas consumiendo una tupla de entrada y produciendo una tupla de salida, hasta recorrer por completo los flujos.

La ventaja principal de este esquema de procesamiento es que, al no permitirse la realimentación, todas las instancias de un *kernel* son independientes entre si; de esta forma podemos acelerar de forma directamente proporcional el procesamiento de los flujos a base de añadir más procesadores de flujo. El paralelismo de datos queda expuesto entre instancias de un mismo *kernel*.

La desventaja es el esfuerzo que hay que hacer para exponer el paralelismo de un programa en su forma del modelo de procesamiento de flujos. Además de las dificultades típicas de la paralelización bajo otros modelos, hay que añadir

restricciones intrínsecas al modelo, como el que los *flujos* tienen que tener el mismo tamaño y el no permitirse el acceso aleatorio al contenido de un flujo. Pero si el programa dispone de paralelismo de datos en grandes cantidades, el beneficio que puede obtener de este modelo es significativo.

Típicamente, un procesador de flujos tiene una memoria para almacenar el *kernel*, una serie de unidades funcionales, un banco de registros y una serie de memorias cache y *buffers* para el procesamiento interno. En la figura 3.1 podemos ver el esquema genérico de un procesador de flujos.

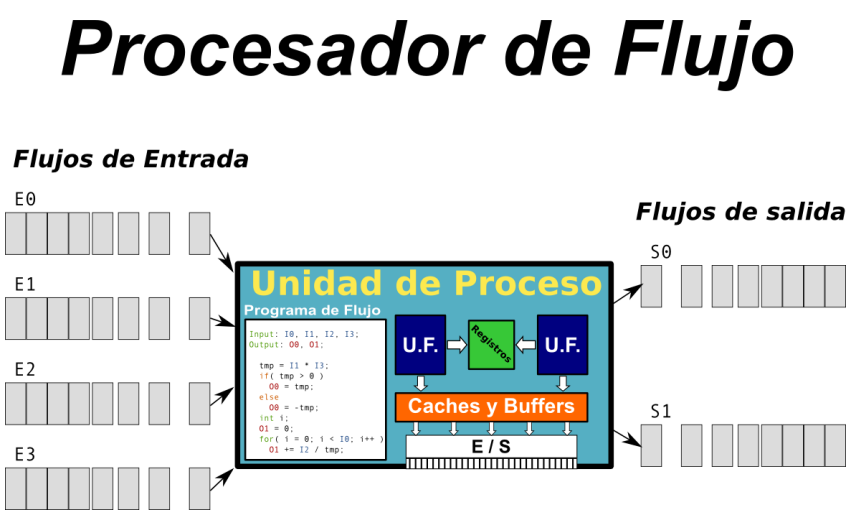


FIGURA 3.1: Unidad de procesamiento de flujos genérica.

En el modelo de procesamiento de flujos, nuestros programas están representados por un conjunto de flujos y una serie de *kernels* encadenados, que realizan el cómputo con los datos que contienen dichos flujos. En la figura 3.2 podemos ver cómo se transforma una función que suma dos vectores en un *kernel* que hará lo propio con procesamiento de flujos.

Una forma cómoda de describir programas en el modelo de procesamiento de flujos, es mediante flechas que representen nuestros flujos y cajas que repre-

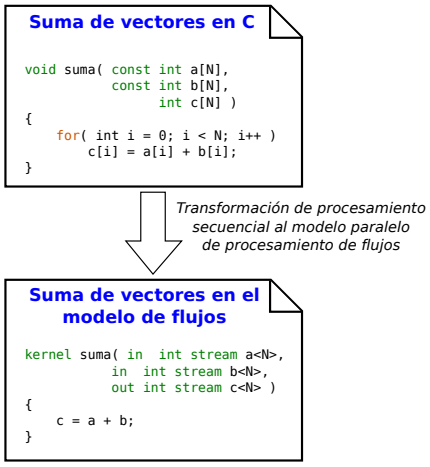


FIGURA 3.2: Suma secuencial de vectores y su transformación al modelo de procesamiento de flujos.

senten nuestros *kernels*. De esa forma podemos ver todo el flujo del programa en forma de flujos de datos que pasan de *kernel* a *kernel*. En la figura 3.3 se muestra esta forma gráfica de representación de programas en el modelo de procesamiento de flujos. Este ejemplo es muy simple, pero podrían incluirse bucles u otros elementos típicos de la programación estructurada con condiciones sobre las flechas o realimentación desde las salidas de los *kernels*. Por facilitar la lectura, durante este capítulo utilizamos este tipo de representación gráfica para los programas.

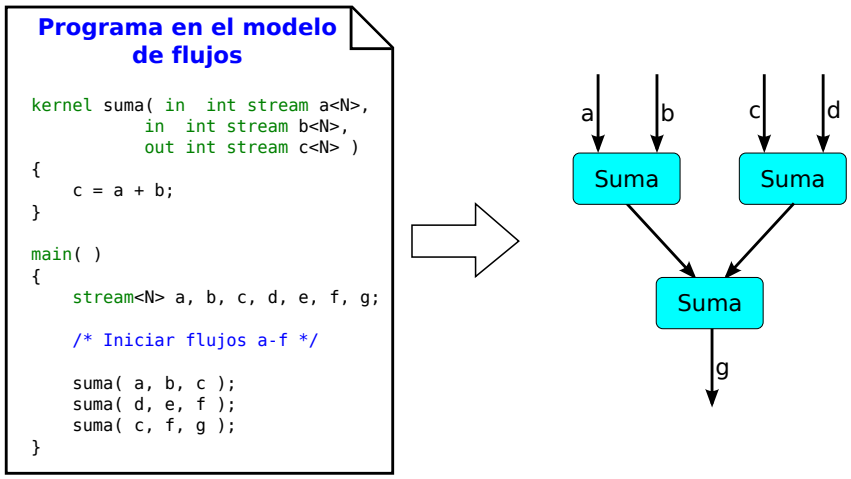


FIGURA 3.3: Representación gráfica de un programa en el modelo de procesamiento de flujos.

3.2. GPGPU sobre OpenGL

La idea detrás nuestra biblioteca de desarrollo es aprovechar al máximo posible estos procesadores como si fuesen procesadores de flujo. Nuestra intención es poder utilizar las *GPUs*, inicialmente pensadas para la síntesis de gráficos, en cómputo de propósito general. Esto es lo que se conoce como *GPGPU* (del inglés *General-purpose Processing on Graphics Processing Units*), o cómputo de propósito general sobre unidades de procesamiento gráfico.

Nuestro objetivo es abstraer los detalles de la biblioteca gráfica y del subsistema gráfico del sistema operativo, de forma que se nos presente una forma de acceso a los recursos computacionales sin necesidad de tener conocimientos avanzados sobre síntesis de gráficos. El esquema de nuestra biblioteca de desarrollo lo podemos ver en la figura 3.4.

Para la biblioteca que hemos desarrollado, localizamos tres funciones básicas que necesitamos como desarrolladores:

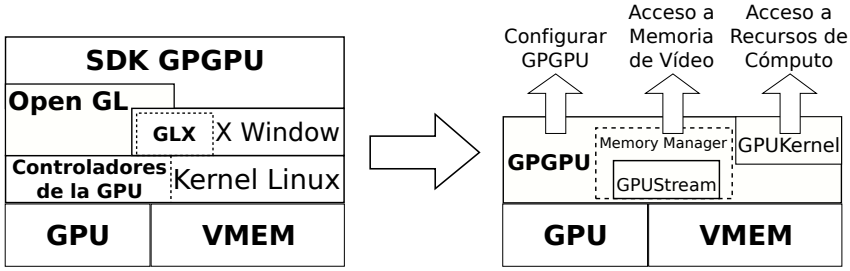


FIGURA 3.4: Nuestra biblioteca de desarrollo (*SDK*) abstrae toda la complejidad del sistema del sistema operativo y *OpenGL* con un modelo de desarrollo basado en flujos y *kernels*.

- Gestión del entorno: Inicialización, acceso a los recursos, sincronización y finalización.
- Gestión de memoria: Reserva y liberación de flujos (*streams*).
- Acceso a los recursos de cómputo: Construcción e invocación de *kernels*.

En nuestra biblioteca hemos encapsulado toda la funcionalidad en tres clases principales: *GPGPU*, *GPUStream* y *GPUKernel*; que se encargan, respectivamente, de la gestión del entorno, la gestión de memoria y el acceso a los recursos de cómputo.

Limitaciones

Debido a que por debajo tenemos una biblioteca de síntesis de gráficos, que es una aplicación muy específica, se nos presentan varias limitaciones con las que tenemos que contar a la hora de trabajar con nuestra biblioteca de desarrollo:

- Los *kernels* se programan en *Cg* [FK03]. Es decir, aunque la mayor parte del trabajo con síntesis de gráficos se abstrae, todavía es necesario traba-

jar con algunos conceptos a la hora de programar *kernels*. Afortunadamente, se puede hacer de forma mecánica y sin necesidad de aprender toda la disciplina de síntesis de gráficos por computador.

- Los *kernels* están limitados a un único flujo de salida y ocho de entrada. Debido a la forma en la que *OpenGL* trabaja con la memoria de las *GPUs*, necesitamos dividir nuestro programa en *kernels* que tomen como máximo ocho flujos de entrada, y tengan como salida un único flujo.

- Los datos en la memoria de la *GPU* se almacenan como valores en punto flotante (*float*). Si queremos trabajar con otros tipos de datos, deben ser convertidos previamente a punto flotante y el *kernel* será el encargado de tratarlos, si fuese necesario, como otro tipo de datos distinto.

Un ejemplo de *GPGPU*

Para ilustrar el funcionamiento de nuestra biblioteca de desarrollo, exponemos a continuación un ejemplo de *GPGPU*. Primero utilizamos directamente *OpenGL* y a continuación nos apoyamos en nuestra biblioteca de desarrollo para mostrar algunas de sus ventajas.

En este ejemplo realizamos una suma simple de dos vectores, a y b, y almacenamos el resultado en un tercer vector c.

Para realizar el cómputo utilizando directamente las primitivas *OpenGL*, lo primero que hemos de hacer es pedirle al sistema un entorno de trabajo o *contexto* de *OpenGL*. Esta operación es totalmente dependiente del sistema operativo, y en el código de la figura 3.5 mostramos cómo habría que hacerlo para *GLX*, que es la extensión de *OpenGL* para *X Window System*.

```
float adata[W*H] = { /* ... */ };
float bdata[W*H] = { /* ... */ };
float cdata[W*H];

void CrearContextoGL( )
{
    Display* dpy = XOpenDisplay( NULL );
    int fbCfgCount;
    int fbAttr[] = { GLX_RED_SIZE, 32, GLX_GREEN_SIZE, 32,
        GLX_BLUE_SIZE, 32, GLX_ALPHA_SIZE, 32,
        GLX_FLOAT_COMPONENTS_NV, True,
        GLX_DRAWABLE_TYPE, GLX_PBUFFER_BIT,
        GLX_DOUBLEBUFFER, False, None };

    GLXFBConfig* fbConfigs = glXChooseFBConfig(
            dpy, DefaultScreen( dpy ), fbAttr, &fbCfgCount );
    int pbufferAttr[] = {
        GLX_PBUFFER_WIDTH, 0, GLX_PBUFFER_HEIGHT, 0,
        GLX_LARGEST_PBUFFER, False, GLX_PRESERVED_CONTENTS,
        True, None };
    GLXPbuffer pbuf = glXCreatePbuffer(
            dpy, fbConfigs[0], pbufferAttr );

    GLXContext ctx = glXCreateNewContext(
            dpy, fbConfigs[0], GLX_RGBA_TYPE, NULL, True );
    glXMakeContextCurrent( dpy, pbuf, pbuf, ctx );
}
```

FIGURA 3.5: Código para la creación de un contexto de síntesis en *OpenGL* que soporte "síntesis en una textura", "síntesis fuera de la pantalla" y texturas con componentes reales.

Los diferentes parámetros que utilizamos en nuestra solicitud de un *contexto OpenGL* son necesarios para poder realizar operaciones de *GPGPU*, como que la salida de nuestras órdenes de *OpenGL* se pueda hacer a una textura o que dicha textura tenga componentes reales en coma flotante.

Una vez establecido el contexto de síntesis, el siguiente paso es configurar la máquina de estados de *OpenGL*. El código se puede ver en la figura 3.6.

Primero inicializamos y habilitamos las extensiones que vamos a utilizar, establecemos una proyección ortogonal para que *OpenGL* no realice ningún tipo de transformación geométrica a los vértices que enviemos a la *GPU* y finalmente configuramos las tres texturas que contendrán nuestros datos. Para este ejemplo, a diferencia de lo que hace nuestra biblioteca de desarrollo, utilizamos tres texturas separadas para nuestros tres vectores, para evitar la complejidad (y código) adicional que supondría configurar un manejo de la memoria como el que utiliza internamente nuestra biblioteca para *GPGPU*.

```
GLuint tex[3]; // a = 0, b = 1, c = 2

void ConfigurarGL( )
{
    // Inicializamos las extensiones
    glewInit( );

    glEnable( GL_TEXTURE_RECTANGLE_NV );
    glEnable( GL_FRAGMENT_PROGRAM_NV );
    glViewport( 0, 0, W/4, H );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    glOrtho( 0, W/4, 0, H, -1.0f, +1.0f );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();

    glGenTextures( 3, tex );
    for( int i = 0; i < 3; i++ ) {
        glBindTexture( GL_TEXTURE_2D, tex[i] );
        glTexParameterf( GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
        glTexParameterf( GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );
        glTexParameterf( GL_TEXTURE_MIN_FILTER, GL_NEAREST );
        glTexParameterf( GL_TEXTURE_MAG_FILTER, GL_NEAREST );
    }
}
```

FIGURA 3.6: Código para configurar la máquina de estados de *OpenGL* para que el espacio de proyección coincida con el espacio de la pantalla.

El siguiente paso, una vez configurado *OpenGL*, es configurar un entorno de síntesis sobre una textura para que la textura $tex[2]$ almacene los datos de salida de nuestro vector c, tal y como se muestra en la figura 3.7.

```
GLuint fbid, rbid;
void PrepararRenderToTexture()
{
    glBindTexture( GL_TEXTURE_RECTANGLE_NV, tex[2] );
    glTexImage2D( GL_TEXTURE_RECTANGLE_NV, 0, GL_RGBA,
            W/4, H, 0, GL_RGBA, GL_FLOAT, NULL );
    glGenFramebuffersEXT( 1, &fbid );
    glBindFrambufferEXT( GL_FRAMEBUFFER_EXT, fbid );
    glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT,
            GL_COLOR_ATTACHMENT0_EXT,
            GL_TEXTURE_RECTANGLE_NV, tex[2], 0 );
}
```

FIGURA 3.7: Preparar la síntesis sobre una textura. En nuestro caso, la textura $tex[2]$ es la que guarda los resultados del cómputo.

Con el entorno *OpenGL* ya configurado para realizar cómputos de propósito general, solo nos queda cargar nuestro programa en la *GPU* como se muestra en la figura 3.8, y ordenar a *OpenGL* que realice los cómputos.

```
GLuint psuma;
void CargarPrograma( )
{
    char* pdata = malloc( TAMMAX );
    FILE* pfile = fopen( "suma.fp", "r" );
    size_t leido;
    leido = fread( (void*)pdata, TAMMAX, 1, pfile );
    fclose( pfile );
    pdata[leido] = 0;

    glGenProgramsNV( 1, &psuma );
    glBindProgramNV( GL_FRAGMENT_PROGRAM_NV, psuma );
    glLoadProgramNV( GL_FRAGMENT_PROGRAM_NV, psuma, leido,
            (unsigned char*)pdata );

    free( pdata );
}
```

FIGURA 3.8: Cargamos nuestro programa de fragmentos y lo enviamos a la *GPU*

En la figura 3.9 podemos ver cómo se realizan los cómputos utilizando *OpenGL*. En un primer paso, cargamos sobre las texturas de entrada los contenidos de los vectores a y b. Seguidamente, ordenamos a *OpenGL* que dibuje un cuadrado con las texturas establecidas anteriormente; el código que ejecutamos en la figura 3.7 se encarga de que la salida se escriba en nuestra textura de salida ($tex[2]$). Por último, solo queda pedirle a *OpenGL* que copie al vector c el contenido de la memoria de vídeo que habíamos reservado para nuestra textura de salida.

```
void HacerComputo( )
{
    /* Cargamos la textura tex[0] con 'a' */
    glActiveTexture( GL_TEXTURE0 );
    glBindTexture( GL_TEXTURE_RECTANGLE_NV, tex[0] );
    glTexImage2D( GL_TEXTURE_RECTANGLE_NV, 0, GL_RGBA,
            W/4, H, 0, GL_RGBA, GL_FLOAT, (void*)tex[0] );

    /* Cargamos la textura tex[1] con 'b' */
    glActiveTexture( GL_TEXTURE1 );
    glBindTexture( GL_TEXTURE_RECTANGLE_NV, tex[1] );
    glTexImage2D( GL_TEXTURE_RECTANGLE_NV, 0, GL_RGBA,
            W/4, H, 0, GL_RGBA, GL_FLOAT, (void*)tex[0] );

    glBegin( GL_QUADS );
        glMultiTexCoord2f( GL_TEXTURE0, 0.0f, 0.0f );
        glMultiTexCoord2f( GL_TEXTURE1, 0.0f, 0.0f );
        glVertex2f( 0.0f, 0.0f );
        glMultiTexCoord2f( GL_TEXTURE0, 0.0f, H );
        glMultiTexCoord2f( GL_TEXTURE1, 0.0f, H );
        glVertex2f( 0.0f, H );
        glMultiTexCoord2f( GL_TEXTURE0, W/4, H );
        glMultiTexCoord2f( GL_TEXTURE1, W/4, H );
        glVertex2f( W/4, H );
        glMultiTexCoord2f( GL_TEXTURE0, W/4.0f, 0.0f );
        glMultiTexCoord2f( GL_TEXTURE1, W/4.0f, 0.0f );
        glVertex2f( W/4.0f, 0.0f );
    glEnd( );

    glFinish();

    glReadPixels( 0, 0, W/4, H, GL_RGBA, GL_FLOAT,
            (void*)cdata );
}
```

FIGURA 3.9: Cargamos en $tex[0]$ y $tex[1]$ nuestros datos de entrada y dibujamos un cuadrado del tamaño de la pantalla, asignando a sus vértices dichas texturas.

El proceso en sí es complejo y engorroso, requiere conocimientos bastante avanzados de síntesis de gráficos con *OpenGL* así como del subsistema de vídeo

de nuestro sistema operativo. Utilizando nuestra biblioteca de desarrollo todo eso queda oculto al programador, tal y como se puede ver en la figura 3.10.

```
float adata[W*H] = { /* ... */ };
float bdata[W*H] = { /* ... */ };
float cdata[W*H];

void main( )
{
    GPGPU::Initialize( W/4, H, 3 );

    /* Reservamos e inicializamos nuestros flujos */
    GPUStream* a = GPGPU::getGPUStream( W/4, H );
    GPUStream* b = GPGPU::getGPUStream( W/4, H );
    GPUStream* c = GPGPU::getGPUStream( W/4, H );
    a->writeAll( (StreamElement*)adata );
    b->writeAll( (StreamElement*)bdata );

    // 'suma.fp' contiene el código de la suma vectorial
    GPUKernel suma( "suma.fp" );

    suma( a, b, c );
    GPGPU::waitForCompletion();

    c->readAll( (StreamElement*)cdata );

    GPGPU::Finalize( );
}
```

FIGURA 3.10: Utilizando nuestra biblioteca de desarrollo.

Para realizar la misma tarea solo necesitamos unas pocas llamadas a nuestra biblioteca, dejando totalmente expuesto el modelo de procesamiento de flujo y ocultando lo referente a la síntesis de gráficos por computador. Primero hacemos una llamada para inicializar la biblioteca. Seguidamente reservamos nuestros tres flujos y cargamos en a y b los datos de entrada. Declaramos nuestra función *kernel* pasándole como parámetro el archivo con el código.

Invocamos nuestro *kernel* como si fuese una función más, solo que le pasamos como parámetros los flujos que hemos reservado. Con una llamada a nuestra biblioteca nos quedamos esperando a que la *GPU* termine su tarea, y ya podemos leer los datos de nuestro flujo de salida c a la memoria principal.

Gracias a nuestra biblioteca de desarrollo, podemos diseñar nuestros programas pensando en el modelo de procesamiento de flujos y dicho modelo quedará expuesto claramente en nuestro código, sin invocaciones a una serie de bibliotecas gráficas del sistema que dificulten su interpretación.

Para conocer más sobre nuestra biblioteca de desarrollo para *GPGPU*, al final de este trabajo se encuentra el apéndice C con más detalles sobre su implementación.

3.3. *AMEE* en OpenGL

Para implementar la extracción morfológica de *end-members* en nuestra biblioteca de desarrollo para *GPGPU*, lo primero que hay que hacer es transformar nuestro algoritmo al modelo de procesamiento de flujos con las restricciones que nos impone la propia arquitectura de la *GPU*:

- Un máximo de ocho flujos de entrada y un único flujo de entrada/salida por *kernel*.
- Elementos de cuatro componentes en los flujos.
- La limitación de memoria de la *GPU*.

Dado que las imágenes hiperespectrales suelen suponer un volumen de datos bastante grande, el primer punto a resolver es división de nuestra imagen

hiperespectral en flujos con elementos de cuatro componentes cada uno.

AMEE posee paralelismo implícito a muchos niveles, entre ellos paralelismo en las dimensiones espaciales. Esto significa que podemos realizar el cálculo de nuestro mapa *MEI* por trozos, dividiendo nuestra imagen original en regiones espaciales más pequeñas que contendrán toda la información espectral.

Seguidamente, debido al tipo de operaciones que realizamos con los vectores-píxel, es conveniente almacenar la información espectral contigua en los elementos de nuestros flujos.

De este modo, para mapear nuestra imagen hiperespectral en flujos de elementos con cuatro componentes, realizamos dos divisiones a nuestra imagen (ver figura 3.11):

- Espacial: Dividimos nuestra imagen original en *"regiones espaciales"* (R.E.), de forma que una región espacial (con toda la información espectral) quepa dentro de la memoria de la *GPU* para operar con ella.

- Espectral: A su vez, la región espacial se divide en "bloques" con cuatro bandas espectrales contiguas cada uno. Esta división supone la necesidad de reorganizar los datos antes de copiarlos en los flujos, pero obtendremos un gran beneficio al aprovechar las operaciones vectoriales.

El segundo punto a resolver son las fronteras. Al hacer la división espacial, tenemos que tener en cuenta que para calcular el *MEI* de los vectores-píxel fronterizos, necesitamos información de la región espacial adyacente. Además, si queremos utilizar la versión multi-pasada del algoritmo, esas fronteras dependerán del número de pasadas que queramos realizar. Con cada nueva pasada, se descarta el borde exterior de anchura un vector-píxel. Gracias al manejo

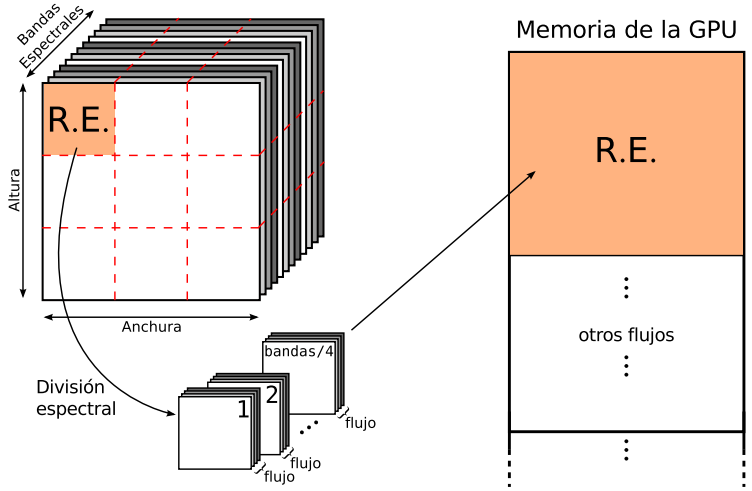


FIGURA 3.11: La imagen se divide en regiones espaciales (R.E.) para acomodarla a la cantidad de memoria disponible en la *GPU*. Cada región espacial se divide espectralmente en flujos con cuatro bandas espectrales cada uno para aprovechar las operaciones vectoriales de la *GPU*

de referencias a partes de un flujo que incorpora nuestra biblioteca de desarrollo (sección 3.2), podemos realizar el descarte de esa región exterior sin ningún tipo de penalización.

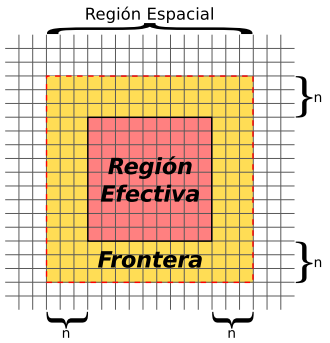


FIGURA 3.12: Región espacial para n pasadas.

Como vemos en la figura 3.12, para realizar n pasadas necesitamos una frontera de n filas/columnas de vectores-píxel por cada lado. Esto, para una región efectiva de $W \times H$ vectores-píxel, nos supone manejar $2 \cdot n \cdot (W+H)+4 \cdot n^2$ vectores-píxel adicionales en nuestra región espacial.

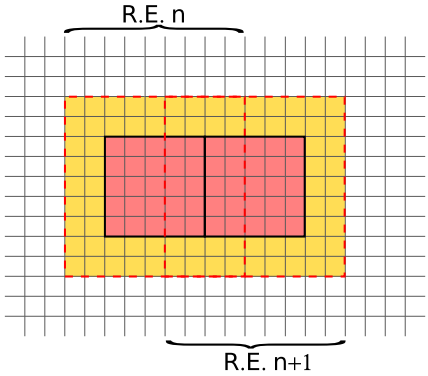


FIGURA 3.13: Las regiones fronterizas de diferentes regiones espaciales se solapan entre ellas y con las regiones efectivas de la región espacial contigua.

Los resultados de mapa *MEI* para la región efectiva son válidos y el resto se descartan. Si nos fijamos en la figura 3.13, podemos ver que existe una redundancia en los cómputos debida al solapamiento entre regiones espaciales. Hay que asegurarse de elegir un tamaño de la región efectiva que compense la redundancia debida a la región fronteriza. En general, para un número de pasadas típico y con la memoria disponible en cualquier *GPU* programable, $W \times H \gg 2 \cdot n \cdot (W + H) + 4 \cdot n^2$, por lo que la redundancia de cómputo y transferencia tiene un impacto despreciable en el tiempo total de procesamiento.

Otra observación importante que podemos realizar fijándonos en la figura 2.1, es que la distancia entre dos píxeles dados es necesaria más de una vez; tanto dentro de una determinada ventana delimitada por el elemento estructural, como en las adyacentes, al desplazar dicho elemento estructural.

Si analizamos todas las distancias en las que participa un determinado vector-píxel durante el cálculo del mapa *MEI*, observamos que, para un elemento estructural de 3×3, un vector-píxel cualquier tiene una región de influencia de 5×5 píxeles a su alrededor; lo que supone su intervención en el cálculo de 48 distancias (ver figura 3.14).

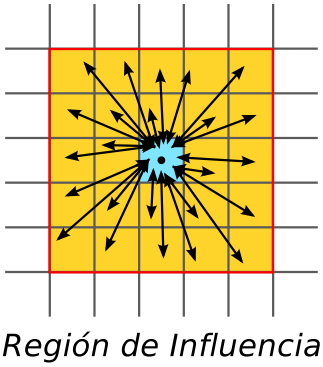


FIGURA 3.14: Cada vector-píxel tiene una región de influencia de 5×5 en el cálculo de distancias acumuladas.

Teniendo en cuenta la conmutatividad en el cálculo de la distancia ($Dist(A, B) = Dist(B, A)$), podemos reducir esas 48 distancias a la mitad, dejando 24 cómputos únicos en los que participa un vector-píxel determinado (figura 3.15).

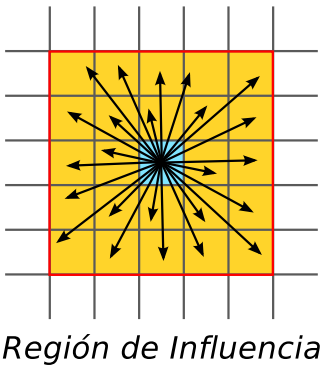


FIGURA 3.15: El número de distancias distintas se reduce a 24 si simplificamos la redundancia por conmutatividad.

Esto significa que podemos reducir el número de cómputos al mínimo si, en lugar de calcular cada vez la distancia, hacemos un pre-cálculo de todas esas distancias para cada vector-píxel. Y si tenemos precalculadas las distancias para cada píxel, aparece una nueva redundancia, esta vez por simetría.

Numerando las distancias, empezando desde el vector-píxel en $f(x, y)$, de izquierda a derecha y de arriba a abajo, obtenemos las 12 distancias distintas

del 0 al 11 que podemos observar en la figura 3.16; y las 12 distancias simétricas respecto a $f(x,y)$ que hemos nombrado añadiendo el signo gráfico "′".

De este modo, habiendo calculado las distancias $Dist_0(f(x,y)), \cdots, Dist_{11}(f(x,y))$ $\forall\ (x,y) \in dominio(f)$, obtenemos las siguientes igualdades:

$$
\begin{aligned}
Dist_{0'}(f(x,y)) &= Dist_0(f(x-1,y)) \\
Dist_{1'}(f(x,y)) &= Dist_1(f(x-2,y)) \\
Dist_{2'}(f(x,y)) &= Dist_2(f(x+2,y+1)) \\
Dist_{3'}(f(x,y)) &= Dist_3(f(x+1,y+1)) \\
Dist_{4'}(f(x,y)) &= Dist_4(f(x,y+1)) \\
Dist_{5'}(f(x,y)) &= Dist_5(f(x-1,y+1)) \\
Dist_{6'}(f(x,y)) &= Dist_6(f(x-2,y+1)) \\
Dist_{7'}(f(x,y)) &= Dist_7(f(x+2,y+2)) \\
Dist_{8'}(f(x,y)) &= Dist_8(f(x+1,y+2)) \\
Dist_{9'}(f(x,y)) &= Dist_9(f(x,y+2)) \\
Dist_{10'}(f(x,y)) &= Dist_{10}(f(x-1,y+2)) \\
Dist_{11'}(f(x,y)) &= Dist_{11}(f(x-2,y+2))
\end{aligned}
$$

3.3.1. *SID AMEE*

Para implementar *AMEE* con la distancia *SID* en la *GPU*, lo primero que tenemos que hacer es transformar el algoritmo a un esquema de procesamiento de flujos. Es decir, una cadena de operaciones que se realizan de manera individual sobre cada uno de los vectores-píxel; siempre respetando las restric-

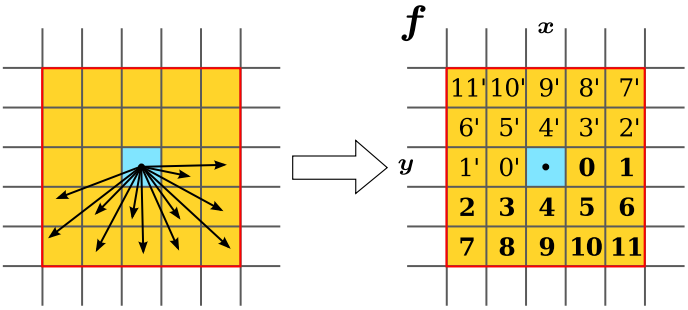


FIGURA 3.16: Por simetría, el número de distancias que es necesario calcular para cada píxel se reduce a 12.

ciones de nuestra biblioteca de desarrollo para *GPGPU* y teniendo en cuenta las limitaciones de memoria.

Ya hemos explicado cómo dividimos la imagen en regiones espaciales y estas, a su vez, en bloques de cuatro bandas espectrales para conformar nuestros flujos. De este modo, nuestro algoritmo en su modelo de procesamiento de flujos tendría las siguientes etapas. Para cada región espacial:

1. Envío de datos a la *GPU*: Reordenamos los datos en flujos tal y como se ha explicado, y copiamos dichos flujos a la memoria *GPU*.

2. Normalización: Calculamos los coeficientes p_k y q_k tal y como se especifica en la ecuación 2.7. Esto supone hacer una reducción en la dimensión espectral y luego dividir el vector-píxel por el resultado. Necesitamos tres *kernels* para ello:

   a) Acumulación: Suma vectorial de flujos y acumulación. Es decir, sumamos los componentes de un vector-píxel cuatro a cuatro y acumulamos el resultado.

   b) Reducción: Suma cuatro componentes en una. Una vez tenemos las sumas parciales en un flujo de cuatro componentes, sumamos esas

cuatro componentes en una.

c) División: Dividimos cada componente del vector-píxel por el resultado de la reducción.

3. Distancia *SID*: Para cada vector-píxel, calculamos las doce distancias *SID* mostradas en la figura 3.16. Esto es sencillo gracias a la posibilidad de trabajar con referencias a flujos parciales que nos permite nuestra biblioteca de desarrollo para *GPGPU*. En esta etapa necesitamos dos *kernels*:

   a) SID^*: Calcula de forma parcial la ecuación 2.9 aprovechando su asociatividad en l. Aprovechamos las funciones trascendentales que tienen las *GPUs* implementadas en *hardware* para realizar un logaritmo vectorial de forma muy eficiente. Esto nos proporciona una ventaja extra respecto a la *CPU*.

   b) Reducción: De nuevo es necesario realizar una suma de las cuatro componentes del elemento del flujo a una sola, completando así el cálculo de la distancia *SID*.

4. Distancia acumulada: De nuevo aprovechando la posibilidad de trabajar con referencias, hacemos el cálculo de las nueve distancias acumuladas en nuestro elemento estructural de 3×3. Hay que notar que esta ya no es una operación vectorial, sin olvidar que aún aprovechamos el paralelismo entre distintos núcleos de cómputo en la *GPU*.

5. Min/Max: Calculamos la posición relativa del mínimo y el máximo dentro del elemento estructural. Utilizamos las cuatro componentes del flujo

de salida para almacenar el desplazamiento en x e y, respecto al centro, para el máximo y el mínimo.

6. MEI_3: Nos quedamos, de las doce distancias *SID* calculadas anteriormente, con aquella correspondiente a la distancia entre el máximo y el mínimo, y la acumulamos en el mapa *MEI* de salida. Para realizar esta operación, utilizamos la ventaja que nos da tener un espacio de memoria unificado para los flujos, de forma que transformamos las coordenadas (x, y) del máximo y el mínimo para indexar una tabla que nos de el puntero a la distancia que buscamos. Dicha tabla tiene 81 entradas (una por cada una de las posibles combinaciones de máximo/mínimo) y, obviamente, es simétrica respecto a la diagonal. En la tabla 3.1 se muestra con las referencias a qué flujos y a qué puntos de esos flujos hay que inicializarla.

7. Dilatación: Si todavía quedan pasadas por dar, nos quedamos para cada vector-píxel con el de mayor distancia acumulada en el elemento estructural centrado en él. Hay que notar que no realizamos la dilatación sobre la imagen original, sino sobre la imagen ya normalizada, de forma que no repetimos cálculos innecesarios. Para descartar la frontera que queda inservible para la siguiente pasada, una vez más aprovechamos la posibilidad que tenemos de trabajar con referencias a una parte de un flujo.

El algoritmo en su modelo de procesamiento de flujos, se muestra de forma gráfica en la figura 3.17. En dicha figura se pueden ver los caminos que toman los flujos entre los *kernels*, y cómo se relacionan las salidas de unos con las

entradas de otros.

TABLA 3.1: Tabla de punteros a las distancias para el cálculo del *MEI*. *X* significa que la distancia debe ser 0 y *S* que es la simétrica. Los subíndices indican la dirección de qué distancia hay que almacenar, y entre paréntesis el desplazamiento en (x, y) necesario. Por ejemplo: Si centrando el elemento estructural en (x, y) el máximo está en la posición 5 y el mínimo en la 7, el $MEI_3(x, y) = Dist_2(x + 2, y + 1)$.

Min/Max	0	1	2	3	4	5	6	7	8
0	X	D_0	D_1	D_4	D_5	D_6	D_9	D_{10}	D_{11}
1	S	X	$D_0(+1,0)$	$D_3(+1,0)$	$D_4(+1,0)$	$D_5(+1,0)$	$D_8(+1,0)$	$D_9(+1,0)$	$D_{10}(+1,0)$
2	S	S	X	$D_2(+2,0)$	$D_3(+2,0)$	$D_4(+2,0)$	$D_7(+2,0)$	$D_8(+2,0)$	$D_9(+2,0)$
3	S	S	S	X	$D_0(0,+1)$	$D_1(0,+1)$	$D_4(0,+1)$	$D_5(0,+1)$	$D_6(0,+1)$
4	S	S	S	S	X	$D_0(+1,+1)$	$D_3(+1,+1)$	$D_4(+1,+1)$	$D_5(+1,+1)$
5	S	S	S	S	S	X	$D_2(+2,+1)$	$D_3(+2,+1)$	$D_4(+2,+1)$
6	S	S	S	S	S	S	X	$D_0(0,+2)$	$D_1(0,+2)$
7	S	S	S	S	S	S	S	X	$D_0(+1,+2)$
8	S	S	S	S	S	S	S	S	X

3.3.2. *SAM AMEE* en OpenGL

En la implementación de *SAM AMEE* seguimos una estrategia similar a la utilizada con *SID AMEE*. Hacemos el mismo tipo de división espacial y espectral, y el esquema en el modelo de procesamiento de flujos va a ser similar.

El algoritmo de *SAM AMEE* en su modelo de procesamiento de flujos comprende las siguientes etapas:

1. Envío de datos a la *GPU*: Al igual que con *SID AMEE*, dividimos y empaquetamos la imagen hiperespectral original en flujos que, posteriormente, son copiados a la memoria de la *GPU*.

2. Producto escalar: Calculamos los doce productos escalares además del producto escalar del vector-píxel por si mismo, haciendo un total de trece productos escalares. Para esta etapa utilizamos dos *kernels*:

   a) Multiplicar y acumular: Calcula los productos escalares parciales multiplicando los flujos en los que se divide la región espacial con los desplazamientos necesarios. Al final tenemos cuatro productos escalares parciales.

   b) Reducción: Suma las cuatro componentes en una para obtener el producto escalar total.

3. Cálculo de la norma: Realiza la raíz cuadrada sobre el flujo que contiene los productos de cada vector-píxel consigo mismo.

4. Distancia *SAM*: Una vez tenemos todos los productos escalares y las normas de los vectores-píxel, calculamos las doce distancias *SAM* de acuerdo a la ecuación 2.6.

5. Distancia acumulada: Al igual que en el cálculo del mapa *MEI* utilizando la distancia *SID*, calcula las nueve distancias acumuladas a partir de las doce distancias y diferentes desplazamientos.

6. Min/Max: A partir de las nueve distancias acumuladas, buscamos las posiciones relativas del máximo y del mínimo, al igual que con *SID AMEE*.

7. MEI_3: De nuevo, como en el algoritmo basado en la distancia *SID*, nos quedamos con la distancia entre el máximo y el mínimo para acumularla en el mapa *MEI* que estamos calculando. También nos apoyamos en una tabla como la descrita en el caso de *SID AMEE*.

8. Dilatación: Si aún no hemos terminado de dar pasadas, realizamos la dilatación de la imagen original a partir del máximo calculado en la etapa

Min/Max. A diferencia de *SID AMEE*, aquí no podemos re-aprovechar cómputos entre pasadas.

En el caso de *SAM AMEE*, hay que notar que a partir de la etapa del cálculo de los productos escalares, todas las operaciones que se realizan son escalares, con lo que no se puede sacar tanto partido a las unidades durante el cálculo de las distancias. En la figura 3.18 tenemos el esquema de procesamiento de flujos para este algoritmo.

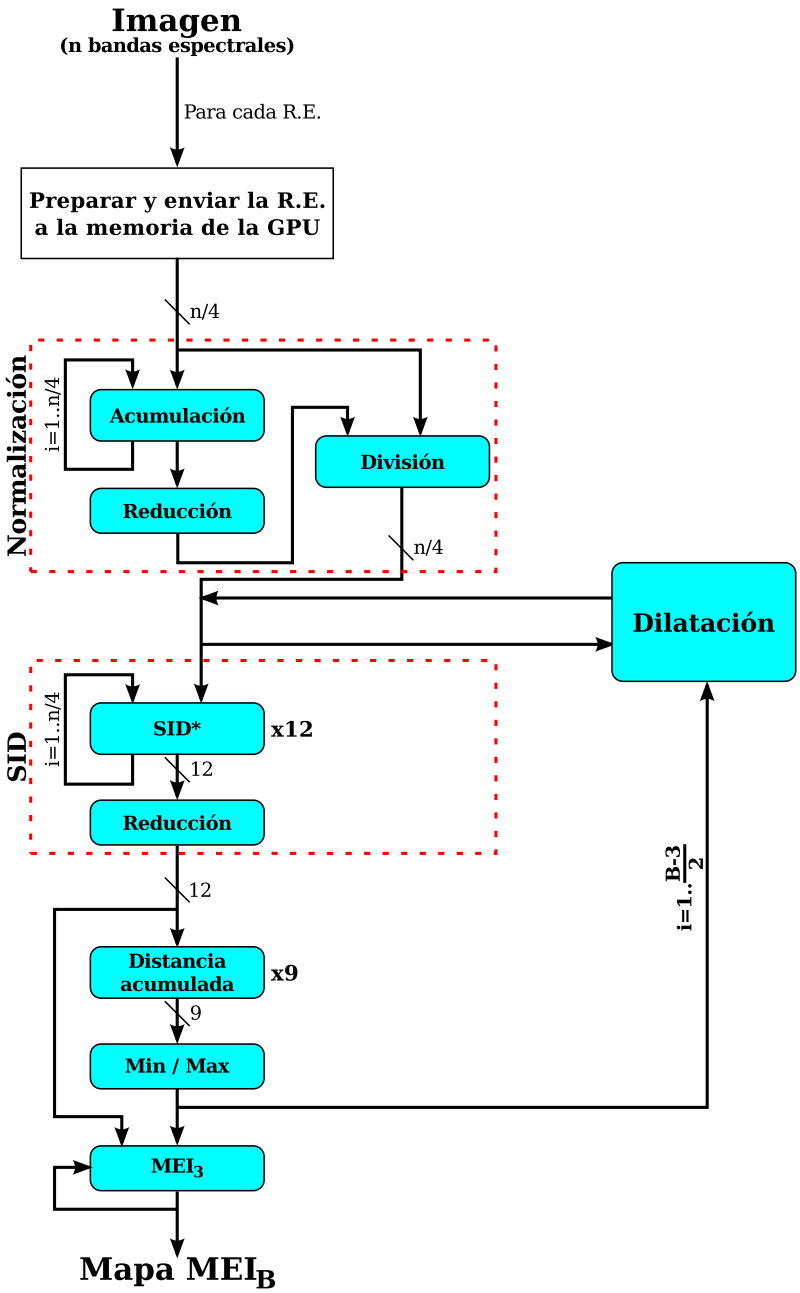


FIGURA 3.17: Representación gráfica del modelo de flujos para *SID AMEE*. Cómputo de la aproximación multi-pasada al MEI_B de una imagen en $\mathbb{K}^n$ (con n bandas espectrales).

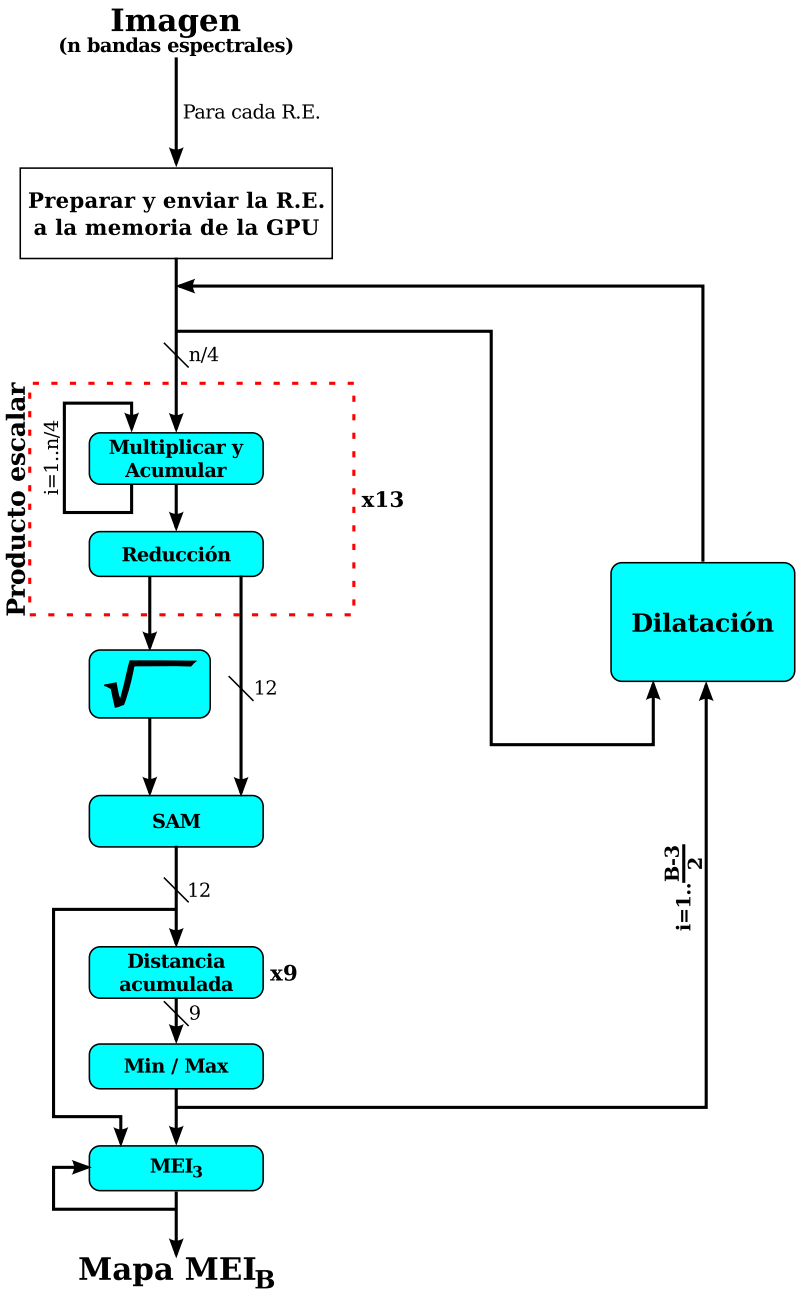


FIGURA 3.18: Representación gráfica del modelo de flujos para *SAM AMEE*. Cómputo de la aproximación multi-pasada al MEI_B de una imagen en $\mathbb{K}^n$ (con n bandas espectrales).

3.4. Resultados

En esta sección exponemos el entorno experimental en el que se han realizado las pruebas tanto para *CPUs* como para *GPUs*, los resultados experimentales obtenidos y las conclusiones que podemos sacar al respecto.

Lo primero de todo, respecto a los algoritmos de *SAM AMEE* y *SID AMEE*, para la *CPUs* hemos optado por realizar la misma implementación que en las *GPUs*. La razón es que las versiones del modelo de procesamiento de flujos también son más rápidas en una *CPU*. En primer lugar porque no realizan cálculos redundantes, y en segundo porque podemos realizar una mayor optimización; el modelo de procesamiento de flujos es más amistoso con técnicas de optimización de acceso a memoria como el *tiling*. No es un resultado extraño que el modelo de procesamiento de flujos funcione mejor también en las *CPUs* de propósito general, este comportamiento ya se había observado en otros casos [GR05].

3.4.1. Entorno experimental

Arquitecturas *CPU* y *GPU*

Para realizar los experimentos hemos tomado como referencia dos generaciones distintas de *GPUs* y dos generaciones de *CPUs* coetáneas a dichas *GPUs*.

La tabla 3.2 presenta las características de las dos generaciones de *CPUs* tomadas para los experimentos, y la tabla 3.3 hace lo propio con las *GPUs*.

TABLA 3.2: Características de las *CPUs*

	Pentium 4 (Northwood C)	**Prescott (6x2)**
Año	2003	2005
FSB	800 MHz, 6.4 GB/s	800 MHz, 6.4 GB/s
ICache L1	12KiB	12KiB
DCache L1	8KiB	16KiB
L2 Cache	512KiB	2MiB
Memoria	1GiB	2 GiB
Frecuencia	2.8 GHz	3.4 GHz

Las *CPUs* elegidas han sido dos *Intel*, un *Pentium IV Northwood* (P4N) de 2003, y un *Pentium IV Prescott* (P4P) de 2005. De igual manera hemos elegido dos *GPUs*, una de 2003 y otra de 2005, ambas de *NVIDIA*. Como representante de 2003 hemos utilizado una tarjeta gráfica del tipo *GeForce FX 5950 Ultra* [BK03] (GPU *NV38*), y como representante de 2005 una tarjeta gráfica del tipo *Geforce 7800 GTX* [Cor05] (GPU *G70*).

TABLA 3.3: Características de las *GPUs*

	NV38	**G70**
Año	2003	2005
Arquitectura	NV38	G70
Bus	AGPx8	PCI Express
Memoria de vídeo	256MiB	256MiB
Frecuencia del núcleo	475 MHz	430 MHz
Frecuencia de la memoria	950 MHz	1.2 GHz GDDR3
Interfaz con memoria	256-bit	256-bit
Ancho de banda	30.4 GB/s	38.4 GB/s
Nº de procesadores	4	24
Tasa de relleno de texturas	3800 MTexels/s	10320 MTexels/s

Datos hiperespectrales

Para realizar las pruebas hemos tomado como referencia una escena muy conocida capturada con un *AVIRIS* sobre un sector minero de Nevada (EEUU) dedicado a la extracción en superficie de la cuprita. Los datos hiperespectrales y la información de verdad-terreno se encuentran disponibles libremente en la web de *AVIRIS* [NASA].

Con la intención de evaluar el rendimiento con imágenes de distinto tamaño, hemos recortado partes de diferentes dimensiones de la imagen original, obteniendo imágenes hiperespectrales de 16, 32, 64, 128, 256 y 512 *MiB*. Hay que tener en cuenta que estas son las dimensiones de las imágenes con los coeficientes almacenados como enteros cortos (*short int*); para almacenarlas en la memoria de la *GPU* deben ser convertidos a valores en punto flotante (*float*), lo que duplica efectivamente su tamaño. De este modo, aunque la imagen de mayor tamaño tratada ocupe 512 *mebibytes* en disco, en la memoria de la *GPU* ocupará un *gibibyte* entero.

3.4.2. Resultados experimentales

En las tablas 3.4 y 3.5 presentamos los tiempos obtenidos con cinco pasadas para las *CPUs* y las *GPUs* respectivamente. En los tiempos de las *GPUs* se ha incluido el coste de reorganizar los datos y transferirlos de memoria principal a memoria de vídeo, así como la transferencia de vuelta del mapa *MEI*. La versión *CPU*, además de estar optimizada manualmente, ha sido compilada con las opciones de optimización automática del compilador *Intel ICC* (*icc -O3 -ttp7 -restrict -xP*).

TABLA 3.4: Tiempo de ejecución (en milisegundos) para la implementación en *CPU* sobre un Pentium IV Northwood (P4N) y un Pentium IV Prescott (P4P).

	SAM AMEE		**SID AMEE**	
Tamaño (MiB)	P4N	P4P	P4N	P4P
16	6588.76	4133.57	22369.8	16667.2
32	13200.3	8259.66	45928	33826.7
64	26405.6	16526.7	92566.6	68185
128	52991.8	33274.9	187760	137412
256	106287	66733.7	377530	277331
512	212738	133436	756982	557923

TABLA 3.5: Tiempo de ejecución (en milisegundos) para la implementación en *GPU*

	SAM AMEE		**SID AMEE**	
Tamaño (MiB)	NV38	G70	NV38	G70
16	898.36	457.37	1923.63	513
32	1817.52	905.93	3909.91	1034.42
64	3714.86	1781.58	7873.9	2035.01
128	7364.12	3573.3	15963.1	4144.82
256	14877.2	7311.05	31854.5	8299.07
512	29794.8	14616	63983.9	16692.2

Podemos ver cómo las versiones de *GPU* son considerablemente más rápidas que las de *CPU*, en especial en el caso del algoritmo *AMEE* con distancia *SID*, ya que tiene una mayor complejidad aritmética y además podemos explotar las operaciones vectoriales en su mayor parte. También podemos observar cómo nuestro algoritmo escala perfectamente, conforme doblamos el tamaño de la imagen, doblamos también el tiempo necesario para procesarla.

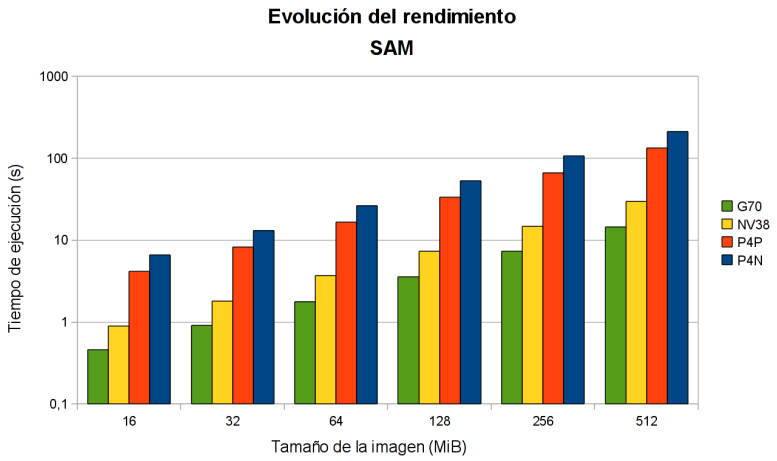


FIGURA 3.19: Rendimiento de la implementación de *SAM AMEE* en *CPU* y *GPU* para diferentes tamaños de imagen y cinco pasadas ($I_{max} = 5$).

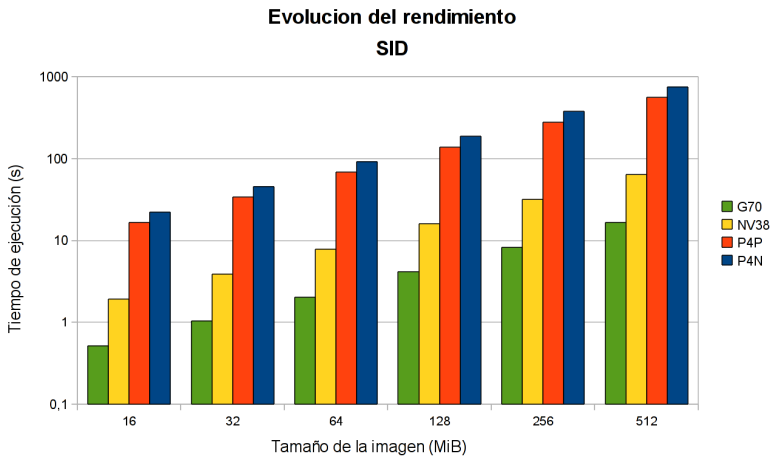


FIGURA 3.20: Rendimiento de la implementación de *SID AMEE* en *CPU* y *GPU* para diferentes tamaños de imagen y cinco pasadas ($I_{max} = 5$).

En las gráficas (en escala logarítmica) de las figuras 3.19 y 3.20 podemos ver dichos resultados y su evolución con el tamaño de la imagen. Mientras que para procesar una imagen hiperespectral de $512MiB$ en la *CPU* podemos necesitar del orden de hasta 10 minutos, esa misma imagen puede ser procesada en torno a los 15 segundos en una *GPU*.

La figura 3.21 muestra la aceleración obtenida en la versión *GPU* respecto a la versión *CPU* para una imagen de $512MiB$ con diferente número de pasadas. Aquí podemos observar que, cuantas más pasadas damos, mayor es la aceleración de la *GPU* respecto a la *CPU*. Esto es debido a que, al no ser necesaria ninguna transferencia de datos adicional entre pasadas, la sobrecarga que suponen las transferencias de datos entre memoria principal y memoria de vídeo, queda compensada por una mayor cantidad de cómputos. En total, para la versión *SAM AMEE* alcanzamos una aceleración de hasta 10, mientras que para *SID AMEE* la versión *GPU* llega a ser entre 35 y 40 veces más rápida que la versión *CPU*.

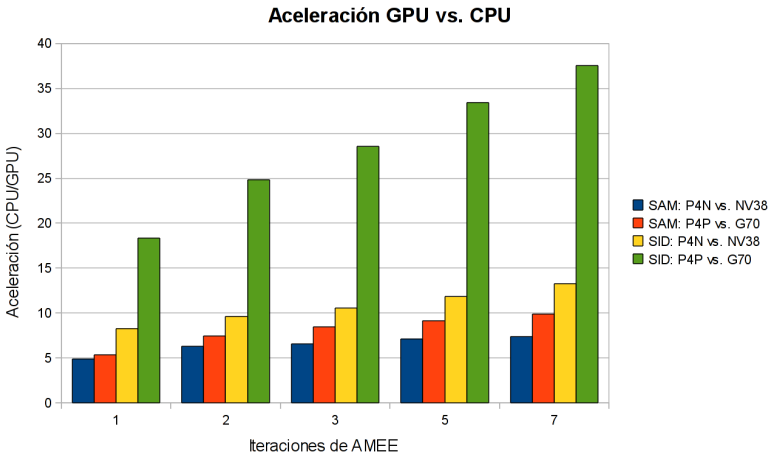


FIGURA 3.21: Aceleración de las implementaciones en *GPU* respecto a las implementaciones en *CPU* para distinto número de pasadas.

En lo que concierne a la comparación entre versiones de *AMEE*, en la figura 3.22 se muestra la aceleración de la versión con distancia *SAM* respecto a la versión con distancia *SID*. Podemos ver que la aceleración es consistente independientemente del tamaño de la imagen. Además, mientras que la diferencia entre versiones es muy destacada en la *CPU*, pudiendo llegar a ser *SAM*

AMEE hasta cuatro veces más rápido que *SID AMEE*; en la *GPU* la diferencia no es tan abultada, aunque sigue siendo notable, en especial en la *GPU* de 2003, donde *SID AMEE* puede llegar a tardar algo más de doble que *SAM AMEE*.

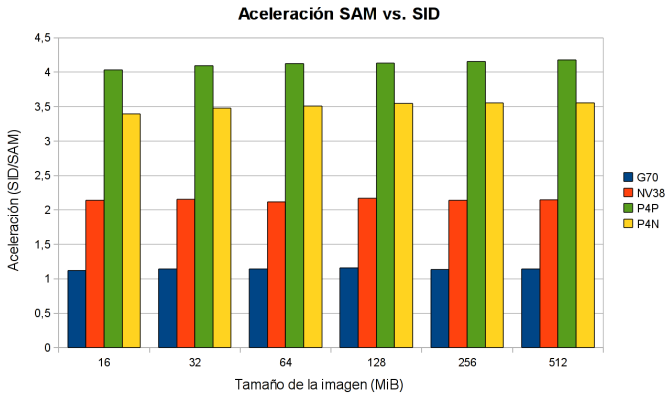


FIGURA 3.22: Aceleración de la implementación de *SAM AMEE* respecto a *SID AMEE* en las diferentes plataformas experimentales, para distintos tamaños de imagen y con cinco pasadas ($I_{max} = 5$).

También podemos comprobar cómo han mejorado las arquitecturas para la *CPU* y para la *GPU* entre generaciones. En la figura 3.23 se ve la aceleración, para ambas versiones de *AMEE* que se experimenta entre las dos generaciones de *GPUs* y las dos generaciones de *CPUs* que estamos comparando.

Podemos observar cómo para las *CPUs* de propósito general la mejora está en torno al 30 %-50 %, mientras que las *GPUs* han doblado prácticamente su rendimiento entre generaciones en el peor de los casos, llegando incluso a ser cuatro veces más rápida la versión con distancia *SID* en la *GPU* de 2005 respecto a la *GPU* de 2003.

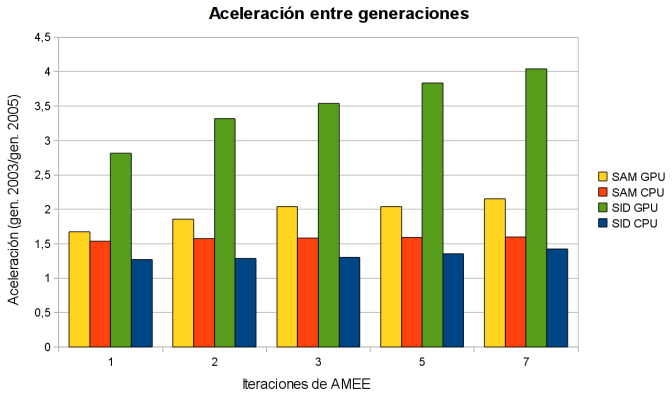


FIGURA 3.23: Comparación de la aceleración dos generaciones de *CPU* (*Pentium IV Northwood* del 2003 y *Pentium IV Prescott* del 2005), y dos generaciones de tarjetas gráficas (*NV GeForce 5950 Ultra* del 2003 y *NV GeForce 7800 GTX* del 2005).

3.5. Conclusiones

Nuestros experimentos han probado que podemos sacar un gran provecho de las *GPUs* para el procesamiento de imágenes hiperespectrales. Hemos observado una mejora importante en los tiempos de ejecución de *AMEE* respecto a *CPUs* contemporáneas. De los datos expuestos podemos extraer las siguientes conclusiones:

- Utilizando una *GPU* para el análisis de imágenes hiperespectrales podemos reducir drásticamente el tiempo necesario de dicho análisis, pasando de varios minutos a pocos segundos.

- El rendimiento de nuestro algoritmo mejora en mucha mayor medida entre generaciones de *GPUs* que entre generaciones de *CPUs*, demostrando una buena escalabilidad en arquitecturas de tipo *manycore*.

- La mejora en la *GPU* respecto la *CPU* es mucho más destacada cuando

utilizamos la versión *SID* de *AMEE*. Esto es debido a que *SID* tiene una mayor densidad de operaciones aritméticas, y además de operaciones aritméticas trascendentales, que en la *CPU* tienen que ser resueltas mediante ciertas funciones software complejas mientras que en la *GPU* se dispone de unidades dedicadas que pueden resolverlas mucho más deprisa.

- Por último, como se vio en la sección 2.1.3, los resultados de precisión para la distancia *SAM* son algo inferiores a los de la distancia *SID*, aunque los resultados de rendimiento son claramente superiores.

Como conclusión general, nuestra biblioteca de desarrollo para *GPGPU* se ha demostrado válida y capaz de sacar el máximo partido de las *GPUs* escondiendo la complejidad de *OpenGL* detrás del modelo de procesamiento de flujos. Además hemos mostrado cómo las unidades de procesamiento gráfico son una plataforma ideal para el procesamiento de imágenes hiperespectrales, obteniendo resultados que nos abren la posibilidad de un procesamiento en tiempo real.

Cuando se inició este trabajo, utilizar *OpenGL* era la única manera en la que podíamos sacar provecho de la potencia que encierran las *GPUs*. En la actualidad, por parte de la industria se nos ofrecen varias alternativas más enfocadas al cómputo de propósito general que nos abren nuevas posibilidades de optimización así como nuevos retos en el desarrollo de aplicaciones sobre *GPUs*.

A pesar de la existencia de estas alternativas, algunas opciones comerciales como la rama *Tegra* de *NVIDIA*, cuyo objetivo es optimizar el consumo para

sistemas incrustados, solo pueden programarse utilizando *OpenGL* por el momento, y podría ser interesante utilizar una biblioteca de desarrollo como la desarrollada en este trabajo para su explotación para cómputo de propósito general.

En el siguiente capítulo (4) exploramos la utilización de *CUDA*, que es actualmente la opción más popular para el cómputo de altas prestaciones sobre *GPUs*, en el análisis de imágenes hiperespectrales.

Capítulo 4

Procesamiento de Imágenes Hiperespectrales con *CUDA*

En este capítulo estudiamos las particularidades del entorno de desarrollo *CUDA* y su aplicación al análisis de aplicaciones hiperespectrales. Comenzamos dando una introducción al entorno *CUDA* en la sección 4.1 para establecer los conceptos sobre los que se trabaja en el resto del capítulo. Seguidamente, en la sección 4.2, realizamos un análisis del rendimiento de *CUDA* en busca de los factores que más nos van a afectar a la hora de portar nuestros algoritmos a este entorno. Para terminar, en la sección 4.3, presentamos diferentes aproximaciones a la implementación de *AMEE* sobre *CUDA* en busca de la mejor versión posible, así como nuestras conclusiones al respecto.

4.1. Introducción a *CUDA*

Para comenzar, vamos a presentar los elementos básicos del desarrollo de aplicaciones en *CUDA*, su funcionamiento y su arquitectura.

Modelo de Programación

El entorno de desarrollo de *CUDA* proporciona un compilador para una extensión a *C*. Dicha extensión permite al programador definir un tipo especial de funciones, los *kernels*, que se invocan de forma que la *GPU* las ejecuta un cierto número de veces especificado en la propia invocación. La *GPU* crea un hilo para cada una de las ejecuciones, y se encarga, en la medida en que los recursos computacionales lo permiten, de ejecutarlos en paralelo. La idea es que el mismo tipo de procesamiento se realiza sobre una gran número de bloques de datos diferentes, cada uno procesado por un hilo distinto que ejecuta una función común (el *kernel*).

Desde los *kernels*, el programador tiene acceso a una serie de índices que identifican el hilo dentro de todos los lanzados para el mismo *kernel*. De esta forma se puede saber qué bloque de datos tiene que procesar esa instancia concreta del *kernel*.

En *CUDA* se distinguen dos espacios de trabajo, el *anfitrión* (*host*) y el *dispositivo* (*device*). *Anfitrión* se refiere a la *CPU* y su memoria principal, mientras que *dispositivo* se refiere a la *GPU* y la memoria de vídeo.

Jerarquía de Hilos

El parámetro principal a la hora de lanzar un *kernel* es el número de hilos que queremos que ejecuten dicho *kernel*. En *CUDA*, los hilos se especifican como una jerarquía de tres niveles. En el nivel superior nos encontramos con las *rejillas* o *grids*. A su vez, una *rejilla* está compuesta por *bloques* de hilos. Los *kernels* son ejecutados por una *rejilla* de *bloques* de hilos. De este modo, cuando invocamos un *kernel* para ejecución, tenemos que especificar la geometría de la *rejilla* que lo ejecutará así como la geometría de los *bloques* que componen la *rejilla*. En la figura 4.1 podemos ver un esquema de esta jerarquía. Se lanzan un total de 72 hilos en 6 bloques de 12 hilos cada uno.

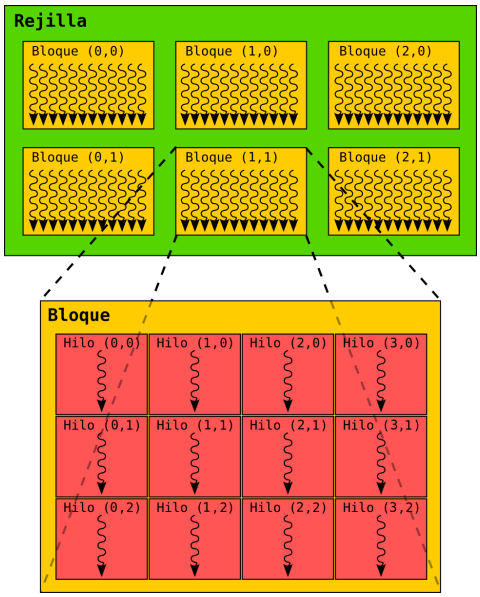


FIGURA 4.1: Una rejilla de $3x2$ bloques de $4x3$ hilos.

Cada instancia de un hilo ejecutando un determinado *kernel* tiene acceso a unos registros que le indican el índice del *bloque* en el que se encuentra dentro de la *rejilla*, así como el índice del hilo dentro de ese *bloque*. También puede

acceder a unos registros globales con las dimensiones tanto de la *rejilla* como de los *bloques*. De este modo, cada hilo ejecutando una instancia de un *kernel* queda unívocamente identificado.

Una *rejilla* puede especificarse como un grupo de *bloques* en una o dos dimensiones. A su vez, un *bloque* puede especificarse como un grupo de hilos en una, dos o tres dimensiones.

Jerarquía de Memoria

En *CUDA* disponemos también de una jerarquía de memoria. En el nivel más bajo de la jerarquía nos encontramos la memoria *local*, que es privada para cada hilo. A nivel de *bloque*, tenemos una memoria que comparten todos los hilos del mismo bloque: la *memoria compartida* o *shared memory*. Y en el nivel más alto, y común a todos los *bloques* y *rejillas*, disponemos de una *memoria global*. En la figura 4.2 se puede ver un esquema de la jerarquía de memorias.

Disponemos de otros dos espacios de direccionamiento especiales para la memoria global: el espacio de memoria de *constantes* y el espacio de memoria de *texturas*. Ambos espacios pueden ser accedidos por los hilos solo para lectura, y ambos disponen de sistemas de *cache* para mejorar su rendimiento. La diferencia del espacio de memoria de *texturas* con la memoria de *constantes* es que la memoria de *texturas* permite una serie de modos de direccionamiento complejos. Además del clásico direccionamiento en una dimensión, similar al que se produce a la memoria de constantes, podemos acceder a la memoria de *texturas* en dos y tres dimensiones, lo que significa que el modo de operación de la *cache* está optimizado para explotar la localidad en dos y tres dimensiones

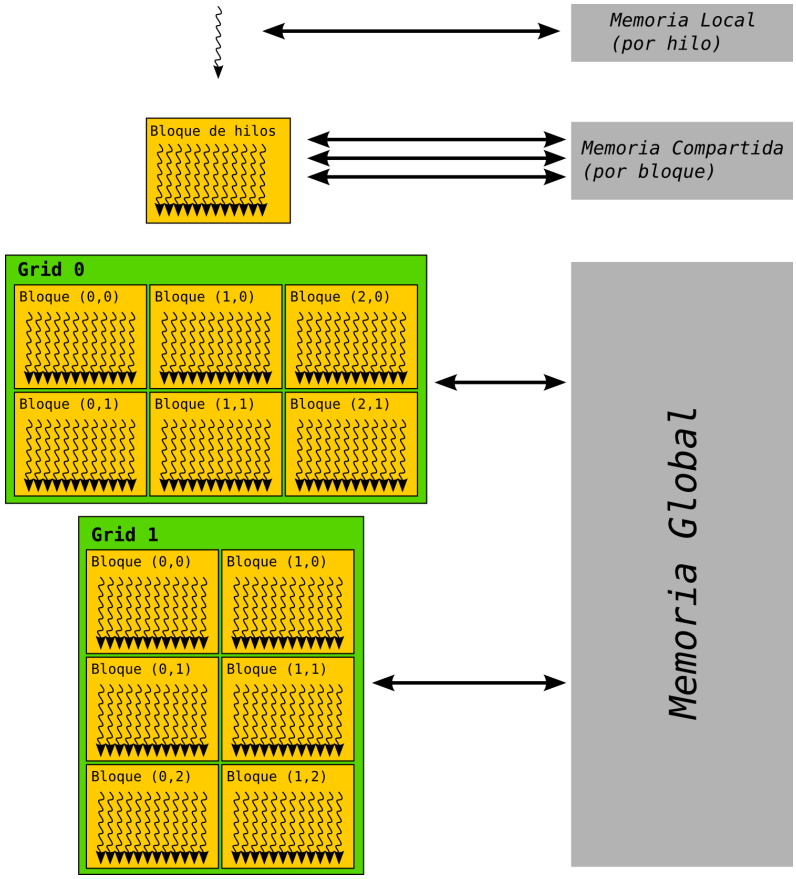


FIGURA 4.2: Esquema de las distintas memorias y su ámbito.

respectivamente.

Los accesos al espacio de direcciones de *texturas* se realizan a través de unas unidades especiales, las *unidades de acceso a texturas* (*texture fetch units*), que también nos permiten configurar una serie de operaciones especiales a la hora de acceder a los datos almacenados. Por ejemplo, podemos pedir a las *unidades de acceso a texturas* que el valor devuelto no sea el almacenado directamente, sino una combinación del valor almacenado con los adyacentes (en dos y tres dimensiones) en función de las coordenadas suministradas. Esto es lo que se conoce como *filtrado de texturas*, y es muy común para evitar artefactos del submuestreo en aplicaciones gráficas.

Los diferentes espacios de direccionamiento se marcan con palabras reservadas al efecto, pero el espacio de texturas funciona de forma diferente a los demás. No se accede a través de variables declaradas dentro de ese espacio, sino que el programador debe configurar las *unidades de acceso a texturas* y desde el *kernel* se utilizan unas funciones especiales para acceder a la memoria de texturas a través de ellas.

Implementación Hardware

CUDA funciona sobre una arquitectura basada en una serie multiprocesadores de flujo (MF). Cuando una aplicación *CUDA* invoca un *kernel*, los *bloques* de la *rejilla* se enumeran y se distribuyen entre los multiprocesadores disponibles de la forma que se muestra en la figura 4.3. Un *bloque* de hilos se ejecuta de forma concurrente en un mismo multiprocesador, y en paralelo con todos los demás bloques distribuidos entre los demás multiprocesadores. De

este modo, la aplicación escala sobre la arquitectura conforme añadamos más multiprocesadores de flujo.

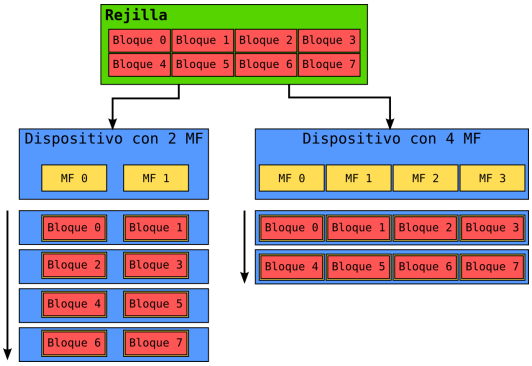


FIGURA 4.3: Reparto de bloques entre los multiprocesadores de flujo en función del número de ellos disponibles en el dispositivo.

En el caso de una *GPU* con núcleo *G80*, cada uno de estos multiprocesadores consiste en dos bloques de ocho núcleos escalares, cada bloque con dos unidades de función especial para operaciones trascendentales, una unidad de instrucciones multihilo y una memoria compartida integrada en el multiprocesador.

El multiprocesador crea, gestiona y ejecuta los hilos de forma concurrente por hardware, de forma que no hay penalización por la planificación de hilos. También implementa una barrera de sincronización entre hilos del mismo bloque mediante una única instrucción. De este modo disponemos de barreras de sincronización rápidas junto con creación ligera de hilos y planificación sin penalización, lo que permite la explotación del paralelismo a grano muy fino. Podríamos encargar a cada hilo que se ocupase de un solo dato sin miedo a que todo el mecanismo de gestión y sincronización de hilos echase a perder el rendimiento de la aplicación, a la vez que la haría más escalable en caso de disponer de más procesadores.

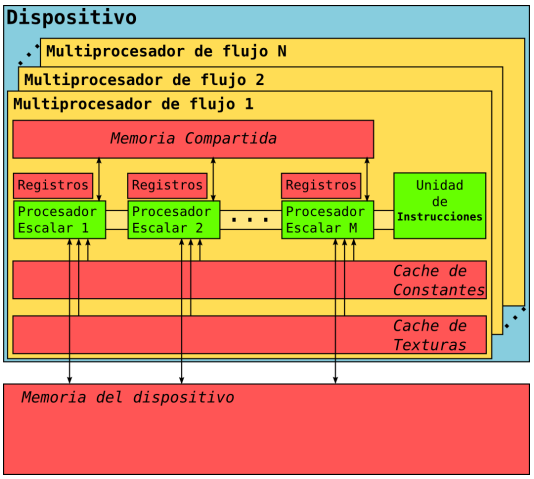


FIGURA 4.4: Implementación en hardware del modelo SIMT de *CUDA*.

Para la gestión de miles de hilos que siguen flujos de ejecución diferentes, el multiprocesador emplea una técnica llamada *SIMT* (una-instrucción, multiples-hilos; de las siglas en inglés *Single-Instruction, Multiple-Threads*). El multiprocesador asigna cada hilo a un núcleo escalar, y cada hilo se ejecuta de forma independiente con su propio puntero de instrucción y registro de estado. El multiprocesador *SIMT* crea, gestiona y ejecuta los hilos en grupos de 32 hilos paralelos llamados *warps* [1]. Los hilos individuales que forman un *warp SIMT* empiezan juntos en una misma dirección, pero son libres de divergir y ejecutarse de forma independiente.

Cuando se asigna uno o varios bloques a un multiprocesador, se parten en *warps* que son planificados por la unidad *SIMT*. Los bloques se dividen en *warps* siempre de la misma manera, cada *warp* contiene hilos con identificadores consecutivos, estando el hilo 0 contenido en el primer *warp*.

En cada turno de lanzamiento, la unidad *SIMT* selecciona un *warp* listo

[1] Como curiosidad, el término *warp* viene del inglés, y está relacionado la tejeduría; en un tejido tienes unos hilos que forman la urdimbre o *warps*, y un hilo que hace la trama o *weft*

para ejecutar y lanza la siguiente instrucción de los hilos activos dentro del *warp*. Un *warp* ejecuta una instrucción común a la vez, con lo que la eficacia máxima se consigue cuando todos los hilos del *warp* siguen el mismo flujo de ejecución. Cuando existe divergencia en el flujo de ejecución debido a un salto condicional el *warp* se divide en dos, dejando activos en uno los hilos que toman en salto y en el otro los que no. Al finalizar la ejecución de ambos caminos en la divergencia, el *warp* se reunifica y todos los hilos siguen desde el mismo punto. La divergencia es algo que ocurre solo dentro de un mismo *warp*, y es independiente del camino de ejecución que sigan los demás *warps*. En realidad, dado que en *CUDA* los *warps* se planifican y ejecutan por mitades o semi-*warps*, la divergencia supone una penalización solo cuando se produce entre hilos de un mismo semi-*warp*.

Una arquitectura *SIMT* es similar a una arquitectura vectorial *SIMD* (*Single-Instruction, Multiple-Data*), en la que una única instrucción controla múltiples elementos de ejecución. La diferencia clave es que las instrucciones *SIMD* exponen la anchura del operando al software, mientras que una instrucción *SIMT* especifica la ejecución de cada uno de los hilo de forma individual. A diferencia con las arquitecturas vectoriales *SIMD*, las arquitecturas *SIMT* permiten al programador escribir código paralelo tanto a nivel de hilo para hilos escalares e independientes, como a nivel de datos para hilos coordinados. A efectos de corrección, un programador puede ignorar la naturaleza *SIMT* de la máquina de forma segura y preocuparse solo de los problemas de la concurrencia, aunque a efectos de rendimiento sí deba tener en cuenta también la divergencia en los *warps*. Esto es análogo a lo que ocurre con las memorias *cache*, en las que el programador puede ignorar de forma segura el tamaño de la línea *cache* y su

comportamiento, pero a efectos de rendimiento es algo a tener en cuenta.

En la figura 4.4 podemos ver a nivel de bloques la implementación física de una máquina *CUDA* genérica con N multiprocesadores con M núcleos de cómputo cada uno. La unidad de instrucciones es la encargada de gestionar los *warps* siguiendo un comportamiento *SIMT*. También podemos ver en la figura las memorias *cache* para constantes y texturas que se mencionaron con anterioridad.

Como hemos podido ver, a la hora de utilizar *CUDA* para programar las *GPUs* tenemos que tener en cuenta que hay muchísimos más factores en manos del programador que cuando utilizamos *OpenGL*.

En *OpenGL*, los parámetros configurables del hardware en el que corren las aplicaciones se dejan al controlador, que está optimizado para ejecutar aplicaciones de síntesis de gráficos lo más rápidamente posible. Esto no es necesariamente lo mejor que podría ocurrir cuando trabajamos con aplicaciones de propósito general. Sin embargo, en *CUDA*, todos esos detalles quedan expuestos al programador, lo que supone una mayor labor de afinación de los algoritmos, pero también más posibilidades de mejorar el rendimiento.

Así pues, lo primero que hacemos es intentar analizar cuáles son los factores que afectan al rendimiento de las aplicaciones en *CUDA* y de qué forma lo hacen (sección 4.2), para seguidamente realizar un análisis de las diferentes posibles adaptaciones de *AMEE* a *CUDA* (sección 4.3).

4.2. Análisis del rendimiento de *CUDA* en *GPUs*

Para analizar el rendimiento, empezamos identificando una serie de elementos sobre los que *CUDA* nos da control y que van a ser relevantes para el rendimiento de nuestras aplicaciones (sección 4.2.1). Seguidamente, diseñamos una serie de experimentos destinados a averiguar hasta qué punto afectan al rendimiento, cómo afectan unos a otros y si hay alguno al que debamos prestar especial atención sobre los demás (sección 4.2.2).

4.2.1. Factores de rendimiento

En las *GPUs* más modernas, con soporte para *CUDA*, tenemos varios factores que van a ser determinantes en el rendimiento obtenido:

- La explotación de la jerarquía de memoria. En *CUDA*, tenemos control directo sobre la memoria global y sobre la memoria compartida (una memoria de baja latencia local a los procesadores), y podemos elegir acceder a la memoria global a través de las unidades de texturas, que disponen de memorias *cache*. Así pues, es responsabilidad del programador explotar la jerarquía de memoria; y hacerlo correctamente va a ser un factor determinante en el rendimiento.

- Patrones de acceso a las memorias. En una *GPU* tenemos miles de hilos corriendo en paralelo y accediendo concurrentemente a la memoria, lo cual supone un gran estrés sobre el bus del sistema. Para paliar esto, las *GPUs* incluyen un controlador de memoria con funcionalidades avanzadas, de forma que si los hilos de un mismo *warp* acceden siguiendo

unos determinados patrones, las lecturas de todos ellos se fusionan en una sola, acelerando enormemente los accesos a memoria. En la plataforma que hemos escogido para realizar los experimentos (*NVIDIA G80*), el controlador de memoria fusiona los accesos del *warp* si se cumplen dos condiciones: primero, todos los accesos tienen que ocurrir dentro de un mismo segmento alineado a 64 bytes; segundo, los accesos de cada hilo deben estar alineados a 4 bytes dentro de ese segmento, y la posición del acceso debe corresponderse con la posición del hilo dentro del *warp*. Como vemos en la figura 4.5, no hace falta que todos los hilos del *warp* realicen un acceso a memoria, pero sí que se respeten las condiciones de alineamiento descritas. Si los accesos por parte de los hilos son cruzados o no están alineados (figura 4.6) no es posible la fusión, del mismo modo que tampoco es posible si los accesos no son consecutivos o algún hilo accede fuera del segmento de 64 bytes (figura 4.7).

- Concurrencia y divergencia en los *warp*. Las sentencias condicionales pueden ser un problema, ya que provocan divergencia y la serialización de hilos. Eso puede hacernos perder mucho paralelismo con la consecuente degradación del rendimiento.

- Ocupación. Nuestros hilos se reparten entre todos los multiprocesadores disponibles y, dependiendo de los recursos que consuma cada hilo, cabrán más o menos hilos por procesador, ya que el reparto de dichos recursos es estático. Elegir el tamaño adecuado de los bloques va a ser determinante para optimizar la ocupación de los procesadores.

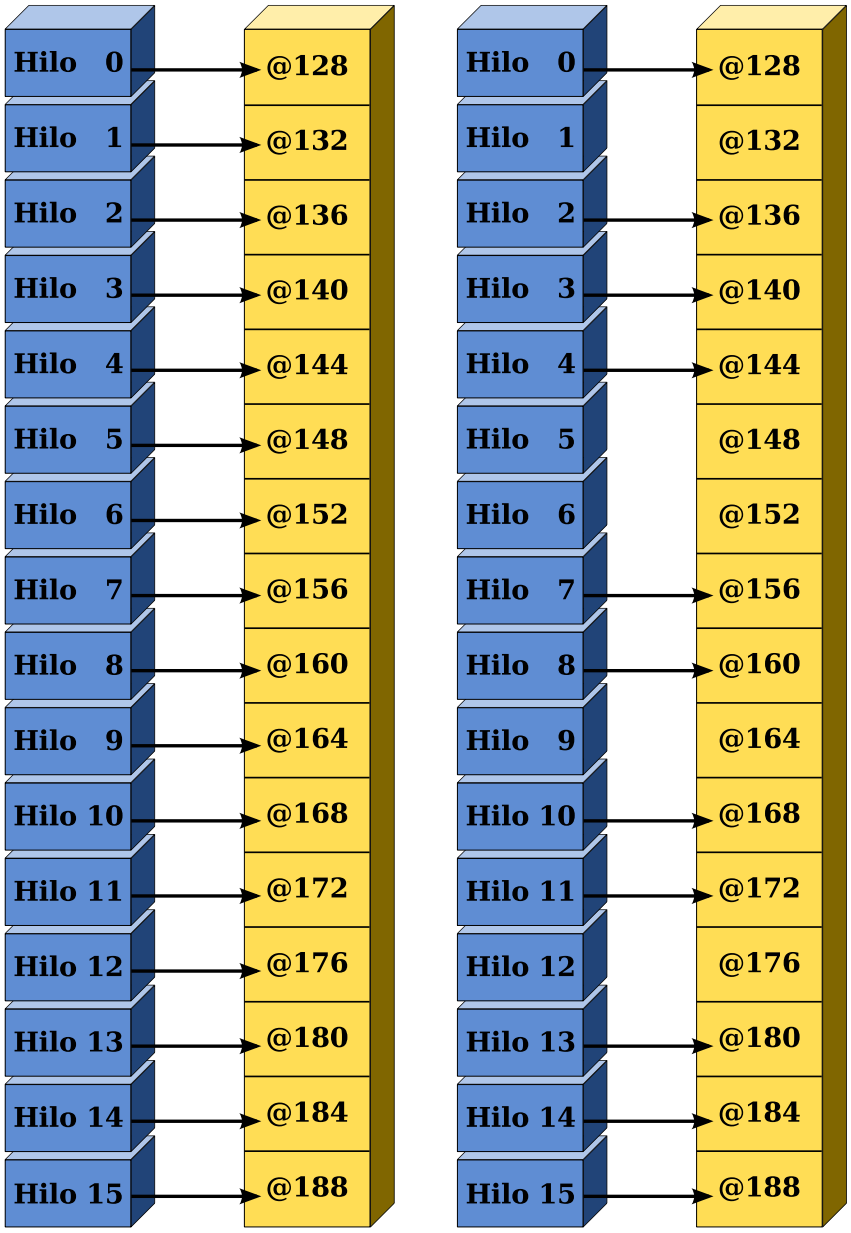


FIGURA 4.5: Accesos fusionados a datos tipo *float*. A la derecha, acceso fusionado con divergencia en el *warp*.

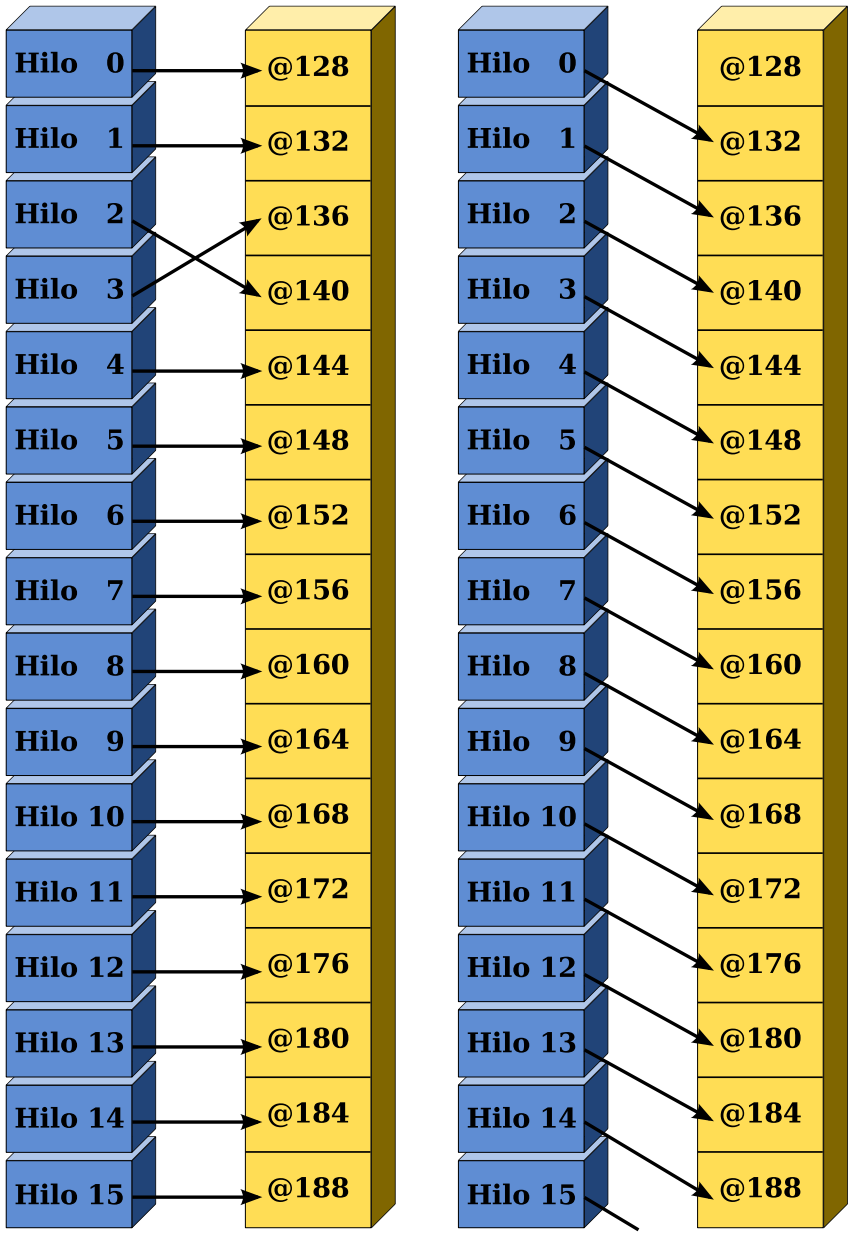


FIGURA 4.6: Accesos no fusionados a datos tipo *float*. A la izquierda, por acceso no secuencial. A la derecha, por accesos no alineados.

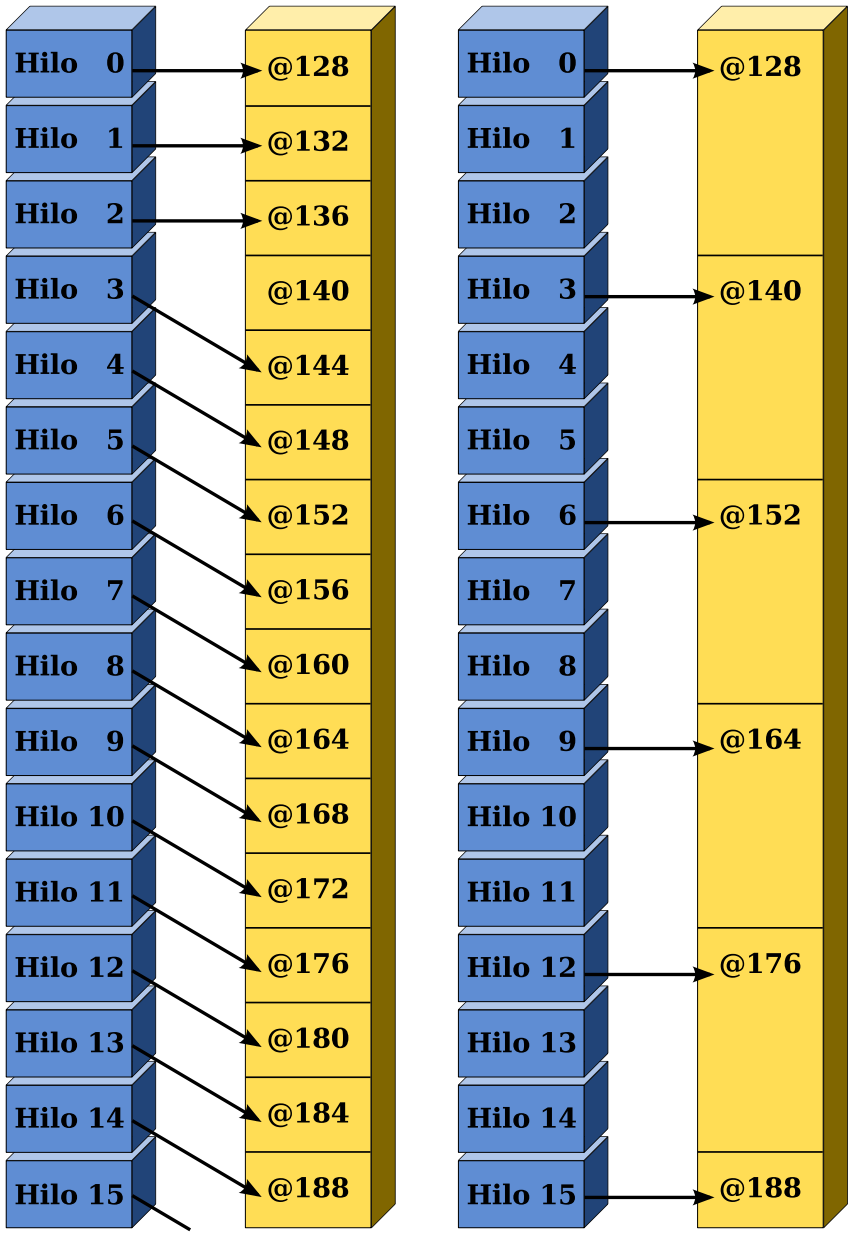


FIGURA 4.7: A la izquierda, acceso a datos de tipo *float* no fusionados por acceso a memoria no contigua. A la derecha, acceso a datos de tipo *float3* no fusionados.

El programador tiene diferentes grados de control sobre estos factores, en algunos casos la mejora de uno podría significar empeorar otro y habrá que llegar a un compromiso.

Si analizamos los factores listados fijándonos en el control que, como programadores, tenemos sobre dichos factores, podemos hacer la siguiente observación: por un lado tenemos factores que son fundamentalmente semánticos, es decir, que dependen principalmente de la propia semántica del algoritmo; por otro lado tenemos factores que son fundamentalmente estructurales, y su explotación está sobre todo en manos del programador y la estructura que le dé al algoritmo.

Dentro del primer caso (factores fundamentalmente semánticos) caería el patrón de acceso a la memoria, sobre el que tenemos más o menos control dependiendo de la semántica del algoritmo, o la divergencia de los *warp*, que también depende en gran medida del algoritmo que estemos tratando. Sobre estos factores podemos actuar, en el caso general, de manera muy limitada. En algunos casos nos será posible reestructurar el algoritmo para optimizarlos (por ejemplo, si los índices de acceso a memoria son fijos e independientes), en otros apenas podremos hacer nada (por ejemplo, si las posiciones de memoria a las que accedemos dependen de los datos de entrada).

Por otro lado, tenemos factores que son fundamentalmente estructurales, como la explotación de la jerarquía de memoria. Está en manos del programador hacer un buen uso de los distintos mecanismos de gestión de la memoria que pone *CUDA* a su disposición, siendo más extraño encontrar algoritmos que no permitan el uso de uno o varios de estos mecanismos.

El caso de la ocupación de los recursos es un poco particular. Por un lado, parecería un caso de factor fundamentalmente semántico, puesto que el programador no puede hacer mucho más que dividir el proceso en *kernels* más pequeños que consuman menos recursos permitiendo una mayor ocupación a

cambio de lanzar un mayor número de *kernels* a ejecución, con el sobrecoste asociado. Pero el propio compilador de *CUDA* dispone de opciones de compilación [nvi12] que nos ofrecen un cierto control sobre los recursos que utilizan los *kernels* y, por tanto, de la ocupación. En este caso, podemos decidir utilizar un mayor número de registros, lo que reduce el intercambio de valores intermedios con memoria, a cambio de reducir el número de bloques que pueden estar ejecutándose en un mismo multiprocesador de manera simultánea, y viceversa.

Con motivo de mantener nuestro estudio del rendimiento acotado y poder generalizarlo dentro de lo razonable, en la sección 4.2.2 nos centramos principalmente en los factores sobre los que tenemos mayor control como programadores.

4.2.2. Análisis del rendimiento

Como punto de partida utilizamos una versión simplificada de *AMEE*, que por sus propiedades (densidad de cómputo y paralelismo) es un caso ideal para un análisis, además de conseguir que nuestras conclusiones sean más fácilmente extrapolables a nuestro caso de estudio específico: *AMEE*.

La simplificación de *AMEE* que hemos desarrollado para utilizar como código base ha consistido en reducir la dimensionalidad espacial de dos dimensiones a una; es decir, eliminamos una de las dimensiones espaciales de las imágenes de entrada. De este modo reducimos sensiblemente el grado de anidación de los bucles sin perjudicar en exceso la riqueza algorítmica desde el punto de vista de las posibilidades de explotación del paralelismo.

Podemos ver nuestro algoritmo en la figura 4.8. En él se observa que los

dos bucles externos del algoritmo *AMEE* se han visto reducidos a uno (de índice ij).

```
for ij=1,n {
    for b1=1,m {
        acc[b1]=0
        for b2=1,l {
            dp=0
            for li=1,S {
                dp+=f1(I,ij,b1,b2,li)
            }
            acc[b1]+=f2(dp)
        }
    }
    max=-Inf
    for b3=1,m {
        if acc[b3]>max {
            max=acc[b3]
            maxpos=b3
        }
    }
    np=0
    for li=1,S {
        np=np+f3(I,ij,maxpos,li)
    }
    res[ij]=f4(np)
    respos[ij]=maxpos
}
```

FIGURA 4.8: Algoritmo simplificado para estudio del rendimiento. La imagen de entrada es I; f_1, f_2, f_3 y f_4 son funciones matemáticas dependientes de los parámetros que reciben.

Del mismo modo, el bucle que calculaba las distancias acumuladas dentro de una ventana ha pasado de recorrer dos dimensiones espaciales a una, dejando los cuatro bucles originales reducidos a dos (b_1 y b_2).

Además, ya no buscamos las posiciones de las distancias acumuladas máxima y mínima, sino solo la máxima, sustituyendo la función que calculaba el *MEI* por otra que solo depende de la posición del máximo.

Por legibilidad, las operaciones matemáticas que se realizan han sido sustituidas por funciones del tipo f_i. En la figura 4.8, f_1 calcularía un producto

parcial entre dos posiciones (b_1 y b_2) alrededor del elemento que se está procesando (ij); por su parte, f_2 haría el cálculo de la distancia a partir de dicho producto. El tipo de producto específico que realiza f_1 y la operación que nos da la distancia en f_2 depende del tipo de distancia que estemos utilizando (*SID* o *SAM*). Lógicamente, f_4 realiza el mismo tipo de operación que f_2, y f_3 el mismo que f_1 con la salvedad de que el producto parcial lo realiza entre la posición ij y $maxpos$.

El algoritmo presenta el mismo tipo de paralelismo en los mismos puntos que nuestro *AMEE* original. Con esto conseguimos dos cosas: por un lado, un algoritmo que, al ser más simple, resulta más fácil de estudiar; por otro lado, que las conclusiones que saquemos de dicho estudio puedan ser más útiles como guía a la hora de portar *AMEE* a *CUDA*.

A la hora de mapearlo en la *GPU* hemos optado por paralelizar el bucle más externo por ser el que mayor grado de paralelismo exhibe, ya que es el que más iteraciones tiene y además no presenta dependencias entre ellas.

Para nuestro análisis, implementamos una versión básica en la que asignamos una iteración del bucle a cada hilo. Luego trabajamos con distintas transformaciones del código y agrupaciones de hilos en bloques, tratando de descubrir los factores de rendimiento más determinantes. Los experimentos se ejecutan sobre una tarjeta gráfica *NVIDIA GeForce 8800 GTX* con núcleo *G80*, que tiene soporte para *CUDA 1.0*.

4.2.2.1. Explotación de la ocupación

Para estudiar la influencia de la ocupación de los procesadores en el rendimiento global, vamos a experimentar con dos transformaciones de código bien cono-

cidas: fisión de bucles y desenrollado de bucles.

En la fisión de bucles, el bucle se divide en varios bucles que recorren el mismo rango de índices pero que solo realizan una parte del trabajo del cuerpo del bucle original. Por ejemplo, en nuestro caso podemos dividir el bucle exterior (ij) para obtener varios bucles con menos carga de trabajo. Con esto, lo que conseguimos es reducir el tamaño y los requisitos de nuestros *kernels*, lo cuál podría permitir una mayor ocupación. Como contrapartida, tenemos que lanzar más *kernels* a la *GPU*, lo que conlleva una cierta sobrecarga. La fisión de bucles también puede acarrear una pérdida de localidad, lo que supondría una pérdida de rendimiento. Por el lado de la memoria, la fisión de bucles puede implicar la expansión de variables, lo que supondría un aumento en los requisitos de memoria. Todo esto puede hacer que la mejora en la ocupación no compense la fisión de bucles.

El desenrollado de bucles también tiene un impacto en el consumo de recursos. Consiste en la replicación del cuerpo de un bucle para construir una secuencia de instrucciones que reduzca el número de iteraciones de ese bucle. De esta forma, la presión sobre el banco de registros aumenta y el número de hilos concurrentes se reduce. Según un estudio [RRB+08], el desenrollado de bucles (en el contexto de la multiplicación de matrices) sigue mereciendo la pena debido al incremento en el paralelismo a nivel de instrucción. Sin embargo, en muchas ocasiones el incremento en del paralelismo a nivel de instrucción no compensaría la disminución en la ocupación de los procesadores.

Algo que también debemos tener en cuenta a la hora de explorar el impacto de la ocupación es que, según el tamaño de bloque elegido, podríamos hacer que la ocupación máxima fuese imposible. Por ejemplo, si disponemos

de multiprocesadores capaces de ejecutar 768 hilos concurrentes y elegimos bloques de 512 hilos, debido a la imposibilidad de partir un bloque de hilos entre varios procesadores nos sería imposible llenar los procesadores por completo. Solo podríamos asignar un bloque a cada multiprocesador con lo que al menos el 33 % permanecería desocupado solo por eso. Algo similar ocurre si elegimos tamaños de bloque demasiado pequeños, debido a la existencia de un límite físico al número máximo de bloques asignados por multiprocesador.

Por último, una forma directa que tenemos de controlar la ocupación de los procesadores es limitando el uso de registros. Los registros son un recurso compartido, y el número que necesita un *kernel* para correr determinará el número de ellos que pueden ejecutarse simultáneamente. El compilador de *CUDA* nos permite poner una restricción al número máximo de registros utilizados por un *kernel*, lo que nos permite controlar la ocupación a costa de perder rendimiento debido al volcado de registros que será necesario.

Resultados experimentales

Para empezar a estudiar la influencia de la ocupación hemos ejecutado dos versiones de nuestro algoritmo simplificado con diferente número de hilos por bloque (de 16 a 384). En una versión, que hemos llamado *simple*, se ejecuta el algoritmo completo con un único *kernel*. En la segunda versión, que llamamos *fisión*, hemos partido el bucle externo ij en dos partes, tal y como se muestra en la figura 4.9. Esta fisión de bucles implica la expansión de la variable *acc* local al bucle ij, debido a la dependencia de *lectura después de escritura* que presenta y que cruza la división. En ambos casos, el mapeo del algoritmo se hace asignando cada una de las iteraciones del bucle ij a un hilo distinto.

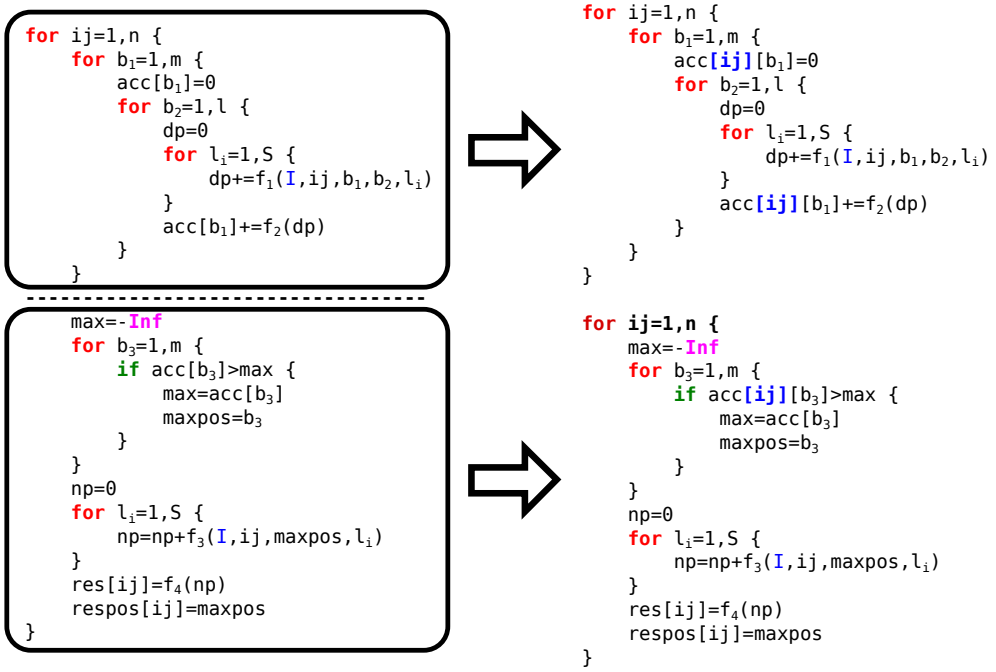


FIGURA 4.9: Transformación de fisión de bucles. Dividimos el algoritmo de la figura 4.8 en dos, expandiendo la variable *acc* debido a la dependencia dentro del bucle.

Cuando lanzamos los primeros experimentos, y en contra de lo que cabría esperar, nos encontramos con que, aunque las diferencias son pequeñas (menos del 10 %), los mejores resultados se obtienen para tamaños de bloque de 16 y 32 hilos, para los cuales la ocupación máxima baja hasta el 33 %. Aparentemente, hay otro factor que está limitando el rendimiento mucho más que la ocupación.

En la figura 4.10(b) podemos confirmar nuestra intuición. En este segundo grupo de experimentos hemos mejorado el acceso a memoria utilizando la *cache* de texturas. Esto supone una presión menor sobre la memoria principal ya que aprovechamos la localidad y paliamos el problema de los accesos no alineados. En estas condiciones, en las que el acceso a la memoria ya no supone un cuello de botella que oculta cualquier otro efecto, el rendimiento óptimo lo conseguimos con bloques de 128 hilos. Lo esperable, dado que con este tamaño

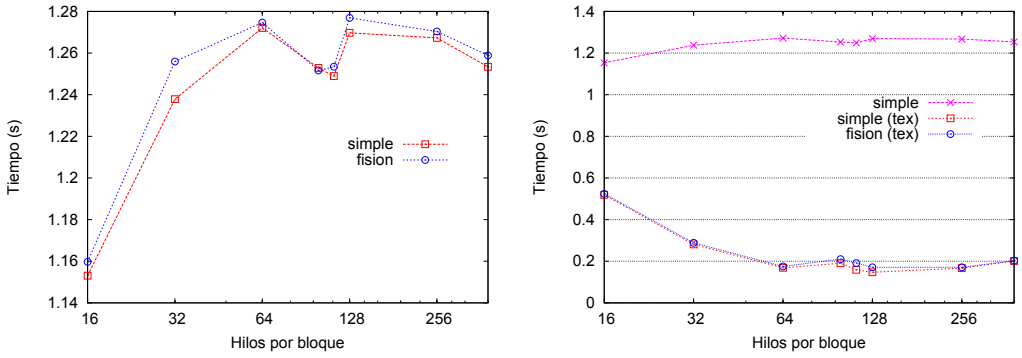
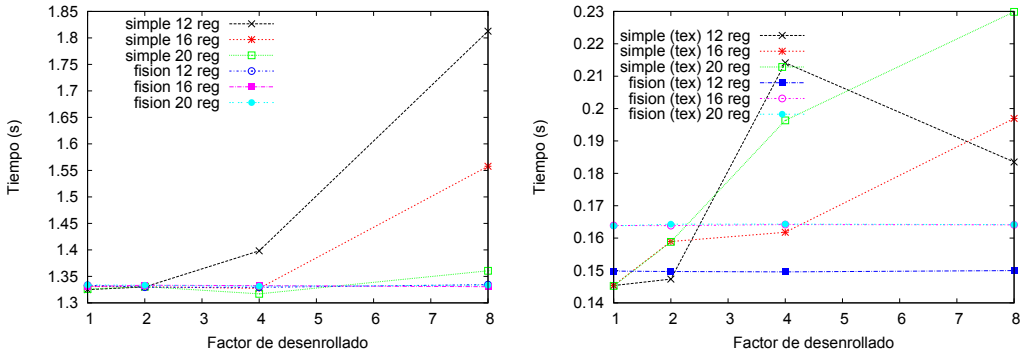


FIGURA 4.10: Tiempo de ejecución en función del número de hilos por bloque. Comparamos una versión del algoritmo en un único *kernel* con una versión en la que hemos realizado una fisión del bucle principal. (a) Con acceso directo a la memoria. (b) Con acceso a través de *cache* de texturas.

de bloque alcanzamos una ocupación del 83 %, el máximo posible para nuestro código de ejemplo.

Otro resultado interesante que observamos en estos experimentos es que la versión fisionada (*fisión*) nunca supera a la versión de un único *kernel* (*simple*). La diferencia en rendimiento se reduce en el caso en el que utilizamos la *cache* de texturas para paliar los problemas derivados del acceso a memoria, pero la sobrecarga del lanzamiento de dos *kernels* sigue siendo más importante que la mejora debida a una menor presión sobre los recursos del multiprocesador.

También hemos estudiado el impacto en el rendimiento del desenrollado del bucle más interno (l_i) de nuestro algoritmo de ejemplo (figura 4.8). Para tener un cierto control sobre la ocupación debida al consumo de recursos, controlamos el número máximo de registros que puede utilizar el *kernel* con parámetros del compilador. Hemos hecho pruebas con 12, 16 y 20 registros, que suponen una ocupación del 83 %, 67 % y 50 % respectivamente, y siendo 20 el número de registros que se utilizan al compilar sin restricciones.



(a) (b)

FIGURA 4.11: Efecto del desenrollado de bucles en el rendimiento. Forzamos la compilación con 12, 16 y 20 registros para establecer la ocupación de los procesadores al 83 %, 67 % y 50 % respectivamente. Las pruebas se han realizado con 128 hilos por bloque.

En la figura 4.11(a) podemos ver el resultado cuando accedemos directamente a la memoria global. Los resultados muestran que apenas hay mejoría con el desenrollado, siendo imperceptible en la versión *fisión* y muy pequeña en la versión *simple* sin limitación en el uso de registros.

En la figura 4.11(b) podemos ver el resultado cuando accedemos a los datos a través de la *cache* de texturas. Al margen de la obvia mejoría con respecto a la versión sin uso de *cache*, observamos que el desenrollado no consigue mejoras en ningún caso, mientras que limitando el número de registros utilizados se nota una leve mejora en la versión *fisión*.

Dado que el rendimiento del algoritmo con fisión de bucles (*fisión*) y la versión con un único *kernel* (*simple*) es muy similar, los experimentos con la jerarquía de memoria de la siguiente sección (4.2.2.2) los haremos solo con la versión *simple*, que es ligeramente más rápida que *fisión*.

4.2.2.2. Explotación de la jerarquía de memoria

En la explotación de la jerarquía de memoria nos vamos a encontrar, como se comentaba al principio del capítulo, tres elementos a tener en cuenta: los accesos alineados, la *cache* de texturas y la memoria compartida.

En la mayoría de las situaciones no es posible garantizar los accesos alineados para todo el *kernel*, algo que ocurre tanto en nuestro ejemplo de la figura 4.8 como en *AMEE*. En nuestro algoritmo, si se garantiza un acceso alineado para un valor del índice b_1, en el resto de las iteraciones de ese bucle los accesos están desalineados en su práctica totalidad. Realizamos varios experimentos dirigidos a evaluar el impacto en el rendimiento de esa desalineación.

La *cache* de texturas es una memoria *cache* de solo lectura a través de la cual podemos acceder a la memoria de la tarjeta gráfica simplemente vinculando una zona de memoria a una *unidad de texturas*. Cuando se produce un fallo de *cache*, el controlador de memoria trae una línea de *cache* completa de la memoria principal. De este modo, podemos evitar los accesos desalineados a memoria a cambio de aumentar potencialmente el consumo de ancho de banda con memoria. Si tenemos localidad espacial en los hilos dentro de un mismo bloque, la *cache* de texturas lo capturará y reducirá la presión sobre el bus de memoria; en caso contrario, la demanda de ancho de banda aumentará.

La memoria compartida es un tipo de memoria local rápida similar a una *cache*, pero controlada totalmente por *software*. Su latencia es sensiblemente menor que la de la memoria *cache* de texturas, se comparte entre los hilos de un mismo procesador y se divide entre el número de bloques que ejecuta dicho multiprocesador. De este modo, el contenido de esta memoria puede ser

leído y escrito solo por los hilos de un mismo bloque. Los beneficios reales del uso de este tipo de memoria dependen enteramente de dos factores: por un lado, la copia de los datos desde la memoria principal de la *GPU* a la memoria compartida, ya que aún se podrían presentar problemas de alineamiento en los accesos; por otro lado, la sobrecarga que implica la sincronización entre hilos de un mismo bloque para evitar condiciones de carrera en el acceso a la memoria compartida.

Resultados experimentales

A la hora de explotar la jerarquía de memoria, es importante notar que en nuestro algoritmo (figura 4.8) se accede a los datos de entrada (I) desde dos puntos: f_1 y f_3. En el primer caso estamos dentro de dos bucles anidados, b_1 y b_2, que provocan que haya un cierto reuso entorno al índice del bucle más exterior (ij) en el bucle dirigido por el índice l_i. En el segundo caso esos dos bucles no están presentes, con lo que el acceso a los datos de entrada I será mucho menos redundante, aunque aún podría haber reuso entre distintos hilos (índices de ij) en los casos en que '*maxpos*' haga coincidir el índice de acceso (que es función de ij y *maxpos* para todos los valores posibles de l_i). Éste sería el caso en el que $ij = x$ para un hilo y $ij = x + 1$ para otro, y mientras que en el primer hilo $maxpos = 2$ en el segundo resulte ser $maxpos = 1$. Desde ambos hilos se accedería a I en $x+2$ para todos los valores de l_i (esto es, $I[x + 2][1..m]$). En el primer caso hay un reuso de datos claro y consistente, mientras que en el segundo es algo circunstancial y totalmente impredecible. Otro factor a tener en cuenta en este caso,es que si dos hilos del mismo *warp* tratan de leer datos distintos del mismo banco de memoria

compartida, se produciría un conflicto que serializaría los accesos, produciendo una pérdida de rendimiento. Éste sería el caso del ejemplo expuesto si el hilo $ij = x$ y $ij = x + 1$ estuviesen en el mismo *warp* (si $x\,\%16 = (x + 1)\,\%16$).

Esta diferencia entre ambos bucles nos hace plantearnos varias estrategias distintas de implementación:

- [1] (simple tex). Para empezar, probamos cómo se comporta el algoritmo cuando todos los accesos se realizan a través de la *cache* de texturas.

- [2] (simple sh). En esta implementación, los datos son precargados en la memoria compartida, lo que exige una primera fase de carga en la que cada hilo del bloque trae un cierto grupo de datos, y una barrera de sincronización para asegurarnos de que los datos están en la memoria compartida antes de empezar a operar con ellos.

- [3] (simple tex+sh). En una tercera versión, para intentar aliviar la carga de los accesos desalineados a la memoria realizamos la precarga en la memoria compartida desde la *cache* de texturas. En este caso la localidad es capturada por la memoria compartida, más rápida pero con el sobrecoste de la sincronización, y el problema de los accesos desalineados debería ser paliado por la *cache* de texturas.

- [4] (B1 sh B2 tex). Dado que el reuso en f_3 es, como hemos comentado, menos frecuente (y predecible) que en f_1, resulta interesante comprobar cómo se comporta nuestro algoritmo si aliviamos la sobrecarga de la sincronización a f_3 dejando la *cache* de texturas para paliar el acceso desalineado y capturar la posible localidad. Para ello hemos realizado

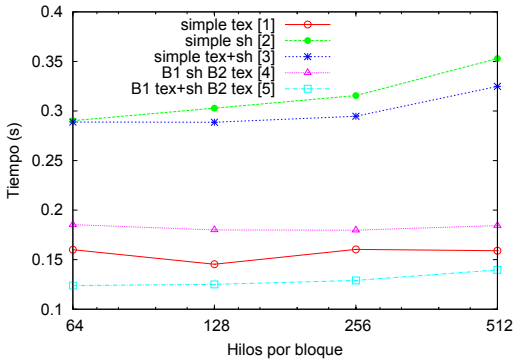


FIGURA 4.12: Efecto del uso de las distintas memorias en el rendimiento. Los resultados, en función del método de acceso a la imagen, son: [1] Acceso a través de la *cache* de texturas; [2] Precarga en memoria compartida; [3] Precarga en memoria compartida a través de la *cache* de texturas; [4] Precarga en memoria compartida en el primer acceso y uso de *cache* de texturas en el segundo; [5] Precarga en memoria compartida a través de la *cache* de texturas en el primer acceso y uso de la *cache* de texturas en el segundo.

esta versión, en la que f_1 accede a los datos precargándolos en la memoria compartida y f_3 lo hace a través de la *cache* de texturas.

- [5] (B1 tex+sh B2 tex). Por último, para ver si en el caso de f_1 podemos beneficiarnos de una posible sinergia entre la *cache* de texturas y el uso de memoria compartida, en una última versión hemos realizado la precarga de datos a memoria compartida para f_1 a través de la *cache* de texturas, con la esperanza de que la *cache* reduzca los problemas de accesos desalineados mientras que la memoria compartida, al ser más rápida, se encarga de capturar el reuso de datos.

En la figura 4.12 podemos ver el tiempo de ejecución de las diferentes versiones implementadas. Tal y como era de esperar debido al patrón de acceso, la versión más rápida es la que utiliza la *cache* de texturas para paliar los problemas de alineamiento en los accesos y utiliza memoria compartida para capturar la localidad de los accesos desde f_1 (B1 tex+sh B2 tex [5]). Si

utilizamos la misma estrategia para el acceso desde f_3 (simple tex+sh [3]), podemos observar que la sobrecarga por la sincronización en dicho acceso reduce drásticamente el resultado global, dejándolo comparable a la versión en la que solo se utiliza memoria compartida ([2]), y que resulta el peor de los casos. Que haya tan poca diferencia entre las versiones [2] y [3] es achacable a dos factores:

1. La naturaleza del bucle que rodea a f_3 causa muchos conflictos en los bancos de la memoria compartida. Desde f_3 se accede a la memoria en la posición del índice del bucle modificado por *maxpos*. El valor de *maxpos* es impredecible, ya que dependerá de los datos de entrada, pero es esperable que provoque conflictos de manera regular. Esto es algo que no ocurre con f_1, que accede a datos en posiciones perfectamente predeterminadas.

2. Como podemos comprobar por las diferencias entre [1] y [4], resulta más eficiente cargar los datos en registros a través de la *cache* de texturas en f_1 en lugar de precargar en la memoria compartida; los conflictos en los bancos de memoria compartida y la sincronización suponen una penalización mayor que la ganancia debida a un acceso más rápido a los datos.

4.2.2.3. Conclusiones del análisis

De nuestros experimentos podemos sacar tres conclusiones principales:

- El aprovechamiento del ancho de banda entre los núcleos de cómputo y la memoria principal de la *GPU* va a ser el factor determinante en

nuestras aplicaciones *CUDA*.

- Podemos obtener mucho beneficio del uso de la *cache* de texturas para paliar los efectos de los accesos desalineados, así como para aprovechar la localidad en el acceso a los datos.

- No siempre es conveniente utilizar la memoria compartida. A la hora de decidirse por su uso, hay que asegurarse de que el reuso de datos es suficientemente alto como para compensar la sobrecarga debida a la sincronización entre hilos y los conflictos en los bancos de memoria compartida.

También es interesante notar la sinergia que presenta el uso combinado de la *cache* de texturas y la memoria compartida. La primera se encarga del perjuicio que causan en el rendimiento los accesos desalineados, mientras que la segunda nos permite aprovechar la localidad con una mayor eficiencia.

Una vez bien establecidos una serie de principios de diseño, en la siguiente sección (4.3) pasamos a implementar nuestro algoritmo *AMEE* sobre *CUDA*.

4.3. *AMEE* sobre *CUDA*

En esta sección vamos a analizar distintas posibilidades de implementación de *AMEE* en *CUDA*. Ya hemos visto en el capítulo 3 que *AMEE* se comporta mucho mejor en una *GPU* que la mejor versión posible en *CPU*, con lo que en este estudio nos centramos en averiguar cuál es la estrategia óptima de implementación, dadas las nuevas posibilidades que se nos presentan con *CUDA*.

En las siguientes secciones se exponen las diferentes implementaciones y los resultados obtenidos tal y como sigue:

- Sección 4.3.1: Como base implementamos una versión análoga a la que utilizamos en *OpenGL*, con los mismos *kernels* que tenían las versiones desarrolladas en el capítulo 3 pero aprovechando la memoria compartida y la *cache* de texturas acorde a lo aprendido en el análisis realizado en la sección 4.2.

- Sección 4.3.2: Aprovechando que las restricciones respecto al tamaño del *kernel* y el número de flujos de entrada/salida que tenía *OpenGL* desaparecen en *CUDA*, implementamos una versión con *kernels* fusionados, ya que los resultados obtenidos en 4.2.2.1 sugieren que debería ser más rápido.

- Sección 4.3.3: Utilizamos la técnica de relleno" (o *padding*, en inglés) para que la imagen en memoria quede alineada por filas, evitando conflictos en el acceso a la *cache*. También, con el ánimo de optimizar los accesos a la memoria compartida, probamos una geometría en los bloques de hilos que maximice el reuso de la memoria compartida.

- Sección 4.3.4: A partir de la versión anterior, tratamos de capturar localidad $2D$ en nuestro algoritmo sustituyendo el acceso a memoria a través de unidades de texturas unidimensionales por unidades de texturas bidimensionales.

- Sección 4.3.5: Con la misma intención que la versión anterior, pero con ánimo de capturar localidad en $3D$, desarrollamos una versión en

la que utilizamos las unidades de texturas tridimensionales que están disponibles en las *GPUs*.

Tras exponer las distintas versiones y sus resultados, realizamos una comparación entre ellas (sección 4.3.6) y presentamos nuestras conclusiones (sección 4.4).

En los experimentos realizados, las distintas implementaciones se ejecutan sobre una tarjeta gráfica *NVIDIA GeForce 8800 GTX* con núcleo *G80*, que fue la primera generación de *GPUs* de *NVIDIA* que soportaba *CUDA*. Las especificaciones técnicas de la *GPU* de núcleo *G80* se pueden ver en el manual de desarrollo de *CUDA*([nvi12]).

4.3.1. De *OpenGL* a *CUDA*

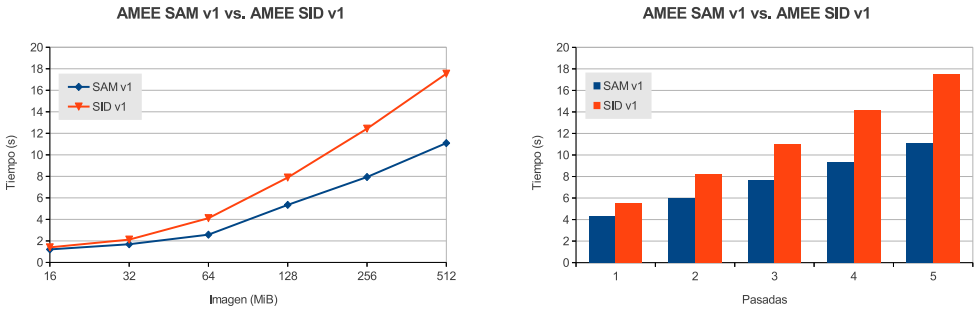
En esta primera versión vamos a tratar de portar la estructura algorítmica que tenemos para *OpenGL* directamente a *CUDA*. Esta va a ser nuestra versión de referencia, a partir de ahí tratamos de hacer mejoras explotando todas las posibilidades que *CUDA* pone a nuestra disposición. Nuestro algoritmo tiene los siguientes elementos:

- Con los resultados de la sección 4.2 en mente, aprovechamos la memoria compartida realizando la precarga a través de la *cache* de texturas para evitar, en la medida de lo posible, las penalizaciones debidas a los accesos no fusionados.
- Un *kernel* por cada distancia a calcular.
- Un *kernel* por cada producto escalar.

- Un *kernel* por cada distancia acumulada.
- Un *kernel* para calcular el mínimo y el máximo.
- Un *kernel* para hacer el cálculo del *MEI*.
- Un *kernel* para realizar la dilatación morfológica.

Como se puede ver, tendríamos el mismo número de *kernels* encargándose de tareas análogas a lo que se podía ver en el capítulo 3. Desde el punto de vista del programa principal las tareas a realizar son las mismas, solo que en lugar de ceder el control a nuestra biblioteca de *GPGPU* para hacer el cómputo invocamos las funciones de *CUDA*.

En la figura 4.13 se pueden ver los resultados obtenidos. Si nos fijamos en los tiempos obtenidos para distintos tamaños de imagen (figura 4.13(a)), observamos que los tiempos para imágenes de 16, 32 y 64 MiB apenas varían, a pesar de estar doblando el tamaño de la imagen en cada instancia respecto a la anterior. Esto nos indica que los tiempos de preparación del entorno *CUDA* son significativamente superiores a los tiempos de transferencia y cómputo en el caso de imágenes pequeñas. Hay que tener en cuenta que para estas pruebas se utilizó una versión *"beta"* del controlador, uno de los primeros que soportaba *CUDA*, y en versiones posteriores el rendimiento del controlador ha sido optimizado. Observando el rendimiento en función del número de pasadas para una imagen grande (figura 4.13(b)), podemos ver tiempos entre un 20 % y un 30 % mejores a los obtenidos en el *OpenGL* en el capítulo 3, que es lo esperable de la mejora tecnológica.



(a) (b)

FIGURA 4.13: Resultados de la conversión directa de *OpenGL* a *CUDA*. (a) Con 5 pasadas y en función del tamaño de la imagen; (b) Con la imagen de 512MiB y en función del número de pasadas.

En esta versión, a pesar de estar basada en la de *OpenGL*, utilizamos una *cache* de texturas unidimensionales que captura localidad en una sola dimensión. Sin embargo, en *OpenGL* la *cache* de texturas estaba configurada en modo bidimensional, con dos coordenadas (x, y) para acceder a un elemento, de modo que capturaba mejor la localidad $2D$ que presenta el algoritmo.

4.3.2. Minimización del número de *kernels*

La primera ventaja de *CUDA* frente a *OpenGL* que vamos a explotar es la restricción en el número de entradas y salidas de los *kernels*, así como la habilidad de acceder de manera arbitraria a la memoria del sistema.

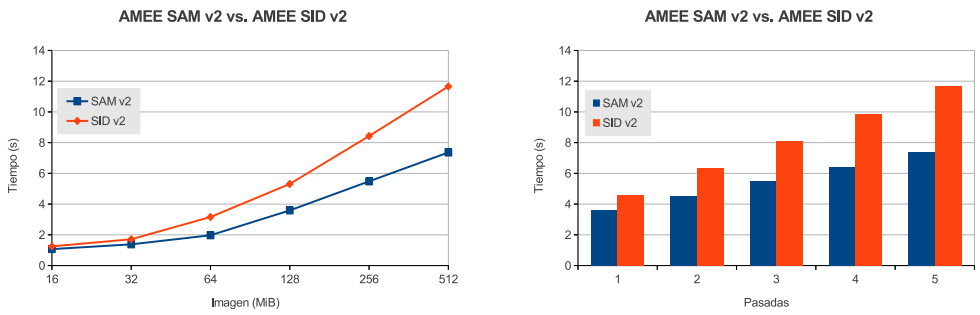
Esto nos permite reducir drásticamente el número de *kernels* que necesitamos lanzar, lo que supone un uso más eficiente de los recursos de la *GPU* así como una reducción en la sobrecarga de las tareas que tiene que llevar a cabo la *CPU*.

En esta versión reducimos las varias decenas de *kernels* de la versión

OpenGL a cuatro:

- Un *kernel* que normalice la imagen para *SID* o que calcule la norma de los vectores-píxel para la versión *SAM*.
- Un único *kernel* que calcule las doce distancias de la vecindad.
- Un único *kernel* que calcule las nueve distancias acumuladas, busque el máximo y el mínimo, y compute el *MEI*.
- Un único *kernel* encargado de la dilatación morfológica.

En la figura 4.14 podemos ver los resultados de esta nueva versión. Como era previsible tras lo observado en nuestro análisis de la sección 4.2, hemos mejorado notablemente los resultados tanto de la versión *SID* como de la versión *SAM*.



(a) (b)

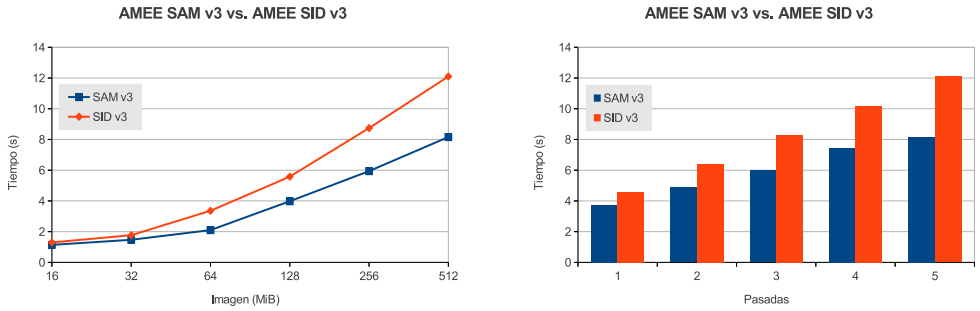
FIGURA 4.14: Resultados minimizando el número de *kernels*. (a) Con 5 pasadas y en función del tamaño de la imagen; (b) Con la imagen de 512MiB y en función del número de pasadas.

4.3.3. Mejora del acceso a la memoria

La siguiente mejora obvia sería utilizar las unidades de texturas en modo *2D* y *3D*, para lo cuál es necesario reservar la memoria de la *GPU* utilizando funciones especiales que "rellenan" la memoria, de forma que queda alineada de manera adecuada para el acceso en dos y tres dimensiones (ver *cudaMallocPitchr* en el manual de programación de *CUDA* [nvi12]).

Para poder separar la mejora debida a las unidades de texturas en dos y tres dimensiones de la mejora debida a un alineamiento óptimo de la memoria, en esta versión seguimos utilizando la *cache* unidimensional, pero con la memoria bien alineada.

Además, también vamos a mejorar el reuso de la memoria compartida cargándola con una geometría que optimice su uso. En lugar de lanzar bloques de hilos cuadrados o que sean óptimos solo para una de las distancias, utilizamos una geometría que minimice la cantidad de datos frontera que necesitamos traer. Con bloques de $16x14$ hilos, los datos que traemos a la memoria compartida y que solo se leen una vez son los mínimos imprescindibles. Únicamente con esta mejora ya se observa un aumento en el rendimiento de hasta un 10 %.



(a) (b)

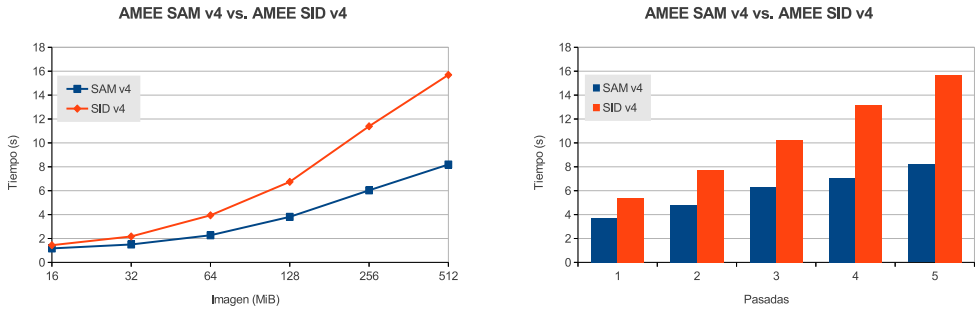
FIGURA 4.15: Resultados optimizando el acceso a la memoria. (a) Con 5 pasadas y en función del tamaño de la imagen; (b) Con la imagen de 512MiB y en función del número de pasadas.

En la figura 4.15 mostramos los resultados de nuestros experimentos. Como se puede observar, aunque retenemos una clara mejora respecto a la versión de la sección 4.3.1, hemos perdido rendimiento respecto a la de 4.3.2.

Se hace aparente que el uso de memoria alineada no supone ninguna ventaja para la *cache* de texturas en modo unidimensional, ya que nuestro algoritmo no presenta ese tipo de localidad.

4.3.4. Aprovechamiento de la *cache* bidimensional

Para esta versión, vinculamos la memoria reservada para la imagen con las unidades de texturas en modo *2D*. Supone un cambio mínimo respecto a la versión anterior, y lo que pretendemos conseguir es capturar la localidad bidimensional presente en el algoritmo.



(a) (b)

FIGURA 4.16: Resultados con texturas bidimensionales. (a) Con 5 pasadas y en función del tamaño de la imagen; (b) Con la imagen de 512MiB y en función del número de pasadas.

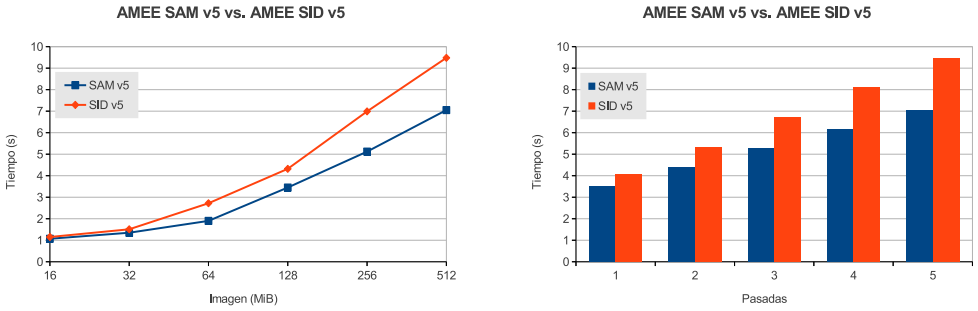
En la figura 4.16 se observan los resultados experimentales obtenidos. Como podemos observar, conseguimos una ligera mejora en el caso medio, aún sin llegar a lo obtenido en al sección 4.3.2.

4.3.5. Aprovechamiento de las texturas tridimensionales

En la versión *CUDA* de *AMEE*, puesto que ya no dividimos la imagen en en bloques de cuatro bandas espectrales para poder aprovechar los procesadores vectoriales, podemos ver claramente que trabajamos con datos en tres dimensiones (dos espaciales más una espectral). En este punto cabe preguntar qué tal se comportaría *AMEE* si tratásemos explotar la localidad en tres dimensiones utilizando la opción que nos da *CUDA*.

En la figura 4.17 podemos ver los resultados de este último experimento. Como se puede observar, los resultados obtenidos superan con claridad a nuestra mejor versión hasta el momento. Esto confirma la intuición de que la

explotación de la localidad *3D* presente en *AMEE* supone una ventaja determinante.



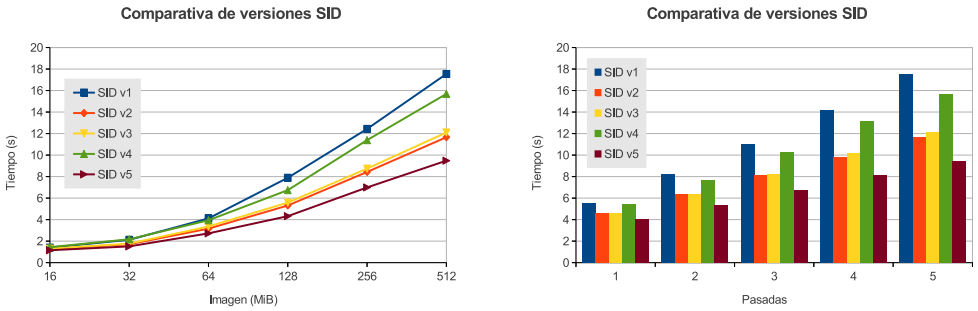
(a) (b)

FIGURA 4.17: Resultados con texturas tridimensionales. (a) Con 5 pasadas y en función del tamaño de la imagen; (b) Con la imagen de 512MiB y en función del número de pasadas.

4.3.6. Comparación de versiones

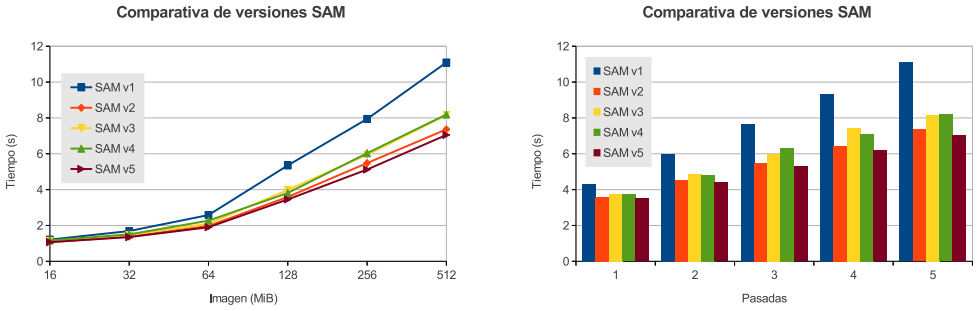
Por último, nos queda comparar las diferentes versiones presentadas entre sí. En la figura 4.18 podemos ver los resultados para *AMEE* con la distancia *SID*, mientras que en la figura 4.19 podemos ver los correspondientes para la distancia *SAM*.

Como se puede observar, en ambos casos nos encontramos con que la implementación con uso de texturas *3D* es la óptima, mientras que muy por detrás de todas las demás se encuentra la versión que es una traducción directa del diseño para nuestro entorno de trabajo en *OpenGL*.



(a) (b)

FIGURA 4.18: Resultados de las distintas implementaciones para la versión de *AMEE* utilizando la distancia *SID*. (a) Con 5 pasadas y en función del tamaño de la imagen; (b) Con la imagen de 512MiB y en función del número de pasadas.



(a) (b)

FIGURA 4.19: Resultados de las distintas implementaciones para la versión de *AMEE* utilizando la distancia *SAM*. (a) Con 5 pasadas y en función del tamaño de la imagen; (b) Con la imagen de 512MiB y en función del número de pasadas.

4.4. Conclusiones

En esta capítulo hemos realizado un análisis de los factores de rendimiento de *CUDA* con el que hemos guiado nuestra implementación de *AMEE*, tanto con distancias *SID* como con distancias *SAM*, para *GPUs* con soporte para este entorno de desarrollo. De los resultados del análisis, así como de los resultados de *AMEE* en *CUDA*, podemos sacar las siguientes conclusiones principales:

1. El buen uso de los diferentes niveles de la jerarquía de memoria va a ser el factor determinante a la hora de desarrollar sobre *CUDA*.

2. Se pueden obtener buenos resultados siguiendo un esquema similar al que seguíamos cuando desarrollábamos sobre *OpenGL*, aunque con relativamente poco esfuerzo podemos conseguir mejoras significativas con *CUDA*.

3. Utilizando las unidades de texturas para traer datos a la memoria compartida en lugar de realizar accesos directos a la memoria principal, podemos mejorar el rendimiento cuando el controlador de memoria no puede fusionar los accesos.

4. Aunque *kernels* más grandes pueden suponer un mayor consumo de recursos con la consiguiente disminución en la ocupación, la sobrecarga de lanzar un mayor número de *kernels* de menor tamaño no compensa la ganancia en ocupación.

Este trabajo, con motivo de mantenerlo acotado, se ha realizado sobre la primera familia arquitectónica que soportó *CUDA*; pero con el tiempo han

llegando *GPUs* con nuevas arquitecturas que eliminan algunas de las limitaciones de *CUDA 1.0* y añaden nuevas capacidades, lo que se traduce en más oportunidades para explotar el paralelismo. Al mismo tiempo, también se ido construyendo *GPUs* destinados a cubrir otras necesidades, como restricciones en el consumo de energía, que podrían ser especialmente útiles en algunos escenarios como el análisis de imágenes hiperespectrales en satélites de observación terrestre. En estos entornos no es suficiente con tener una capacidad de cómputo que haga practicable el análisis abordo; entre otras cosas, hay que cumplir con las restricciones energéticas que tenga el satélite, que suelen estar en el orden de algunos centenares de Vatios que han de ser compartidos entre todos los subsistemas del satélite.

Nuevas arquitecturas

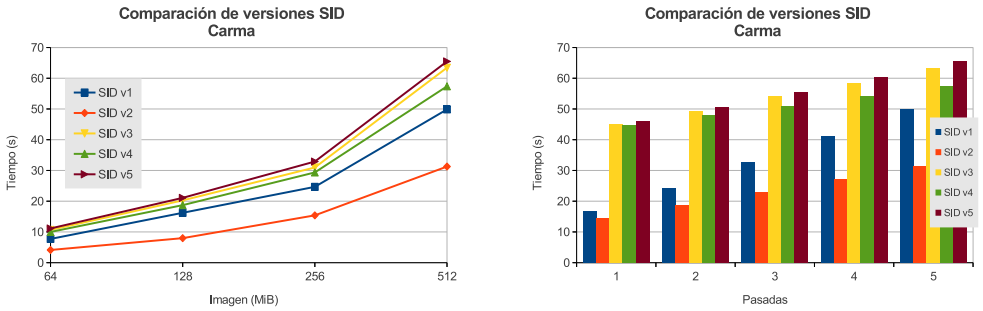
Como ejemplo de estas nuevas arquitecturas y sus posibilidades, en el ámbito de la optimización del consumo tenemos disponibles las placas *Carma* [SEC13], destinadas a proporcionar alto rendimiento basado en *GPUs* a la vez que se optimiza el rendimiento por Vatio. El objetivo es reducir el consumo sustituyendo la *CPU* por un procesador de bajo consumo.

La idea es que si la carga pesada de cómputo se la va a llevar una *GPU*, que ofrece un rendimiento por Vatio muy elevado, podemos sustituir el procesador central por uno de bajo consumo aunque su rendimiento sea muy inferior. En el caso de *Carma*, el procesador principal es un *MPSoC* de tipo *Tegra 3*, que integra una *CPU* de cuatro cores *ARM Cortex A9* y una *GPU ULP GeForce* (*Ultra Low Power*, consumo ultra bajo). Además, como coprocesador

destinado al cómputo intensivo, incorpora una *GPU* de tipo *NVIDIA Quadro 1000M* (núcleo *GF108GLM*) que soporta hasta *CUDA 2.1*.

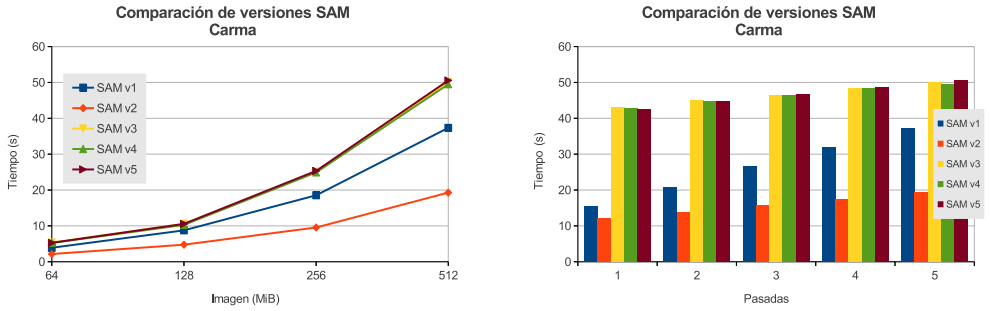
Esta *GPU*, que incluye 96 cores escalares, tiene un consumo de 45 Vatios, frente a los 185 Vatios de las *GPUs* con núcleo *G80*. Proporciona un rendimiento pico de 270 GFlops que, sin llegar a los 518 de la *G80*, proporciona un rendimiento en relación al consumo de $6GFlops/Vatio$ frente a los $2{,}8GFlops/Vatio$ de la *G80*. Es decir, aunque no alcanza el mismo rendimiento bruto, sí hace un uso más eficiente de la energía. Si nuestro objetivo es realizar una tarea en el menor tiempo posible *Carma* no es la solución óptima, pero si tenemos limitaciones en el consumo de energía (como es el caso de una solución embarcada en un satélite) o nuestra principal preocupación es la eficiencia energética, este tipo de placas son una propuesta a tener en cuenta.

Como referencia, hemos lanzado nuestras cinco versiones de *AMEE* portadas a *CUDA* en una placa *Carma*, y podemos ver los resultados obtenidos en las figuras 4.20 y 4.21.



(a) (b)

FIGURA 4.20: Resultados de las distintas implementaciones para la versión de *AMEE* en Carma utilizando la distancia *SID*. (a) Con 5 pasadas y en función del tamaño de la imagen; (b) Con la imagen de 512MiB y en función del número de pasadas.



(a) (b)

FIGURA 4.21: Resultados de las distintas implementaciones para la versión de *AMEE* en Carma utilizando la distancia *SAM*. (a) Con 5 pasadas y en función del tamaño de la imagen; (b) Con la imagen de 512MiB y en función del número de pasadas.

En ellas podemos observar que las versiones que utilizan una reorganización de la memoria (secciones 4.3.3, 4.3.4 y 4.3.5) pasan a ser significativamente más lentas. Si nos fijamos en las diferencias de tiempo entre pasadas, podemos ver que, en realidad, no hay una gran diferencia entre algoritmos. Las diferencias entre 1 y 5 pasadas están todas entre los 7 y los 8 segundos. La gran mayoría del tiempo de cómputo se la está llevando, en este caso, las tareas de *CPU* (transferencia de memoria principal a memoria de vídeo y tareas relacionadas en la *CPU*). En el caso de las versiones con reorganización de la memoria, este trabajo se está llevando un sobrecoste importante. En los sistemas de propósito general, el coste de esta reorganización se veía compensado por un acceso mejor a la memoria. En el caso de *Carma*, que utiliza una *CPU* de bajo consumo y, consecuentemente, menor rendimiento, el sobrecoste de la reorganización es mucho más importante. A esto tenemos que sumarle que, al haberse relajado las condiciones para la fusión de accesos a la memoria, no estamos sacando tanto partido a la memoria *cache*. En definitiva, nos encontramos con un

sobrecoste mucho mayor y una ventaja menor al utilizar caches $2D$ y $3D$, por lo que obtenemos unos resultados relativos significativamente diferentes a los obtenidos en la *GPU* con núcleo *G80*.

Como conclusión general, se hace evidente que los resultados específicos de cada implementación dependerán en gran medida tanto de la arquitectura de la *GPU* como del sistema sobre el que esté montada. Si se busca una solución óptima, y teniendo en cuenta tanto el ritmo al que van cambiando las arquitecturas de las *GPUs* como al que aparecen nuevos sistemas que las incorporan, es necesario un estudio específico para cada plataforma. Una posible solución para hacer este problema abordable es proporcionar una biblioteca con primitivas auto-optimizables, de un modo similar al que *ATLAS* (*Automatically Tuned Linear Algebra Software*) ofrece primitivas para álgebra lineal cuyos parámetros internos se determinan de manera automática sobre cada plataforma específica.

En el siguiente capítulo exponemos las principales conclusiones de nuestro trabajo y las principales publicaciones a las que ha dado lugar, así como algunas ideas para un trabajo futuro.

Capítulo 5

Principales Aportaciones

Nuestro objetivo principal en esta tesis ha sido el estudio del análisis de imágenes hiperespectrales sobre unidades de procesamiento gráfico, en concreto la extracción de *end-members* utilizando el algoritmo *AMEE*, un algoritmo de búsqueda no supervisado que aprovecha la correlación espacial y espectral apoyándose en el uso de operaciones morfológicas extendidas.

Nuestro estudio se ha dividido en dos partes atendiendo a las dos grandes familias arquitectónicas: la presente al inicio del trabajo, cuando no existían entornos de desarrollo que permitiesen trabajar con *GPUs* como procesadores de propósito general; y la que apareció durante el desarrollo del trabajo, junto con la cual los fabricantes publicaron sus propios entornos de desarrollo que permitían el uso de las *GPUs* como sistemas destinados al cómputo de altas prestaciones.

Para la primera parte, dado que la única manera de aprovechar el *hardware* de una *GPU* era utilizar una biblioteca de síntesis de gráficos –*OpenGL* en nuestro caso– empezamos por diseñar e implementar una biblioteca de desarrollo

que abstrajese toda la complejidad de la síntesis de gráficos exponiendo un modelo de procesamiento basado en flujos. Apoyándonos en nuestra biblioteca para la implementación, hemos transformado dos versiones del algoritmo *AMEE* –una basada en la distancia *SID* (*Spectral Information Divergence*) y otra basada en la distancia *SAM* (*Spectral Angle Mapper*)– al modelo de procesamiento de flujos, extrayendo todo el paralelismo implícito en el algoritmo y eliminando toda la redundancia presente en la versión ingenua del algoritmo.

En la segunda parte, y gracias a las nuevas arquitecturas que aparecieron, hemos trasladado nuestro estudio al entorno *CUDA*, con muchas menos limitaciones que nuestra biblioteca de desarrollo y, a la vez, muchos más retos a la hora de explotar el rendimiento. Para ello, hemos empezado realizando un análisis del rendimiento que nos permitiese decidir qué parámetros iban a ser los más influyentes en nuestro caso de estudio, localizar problemas potenciales y estudiar posibles soluciones. Hemos utilizado las conclusiones de este primer análisis para guiar la implementación de las dos versiones de *AMEE* sobre el entorno *CUDA*, lo que nos ha llevado a analizar los resultados de cinco aproximaciones diferentes.

Este estudio ha dado lugar a las siguientes publicaciones:

[STP+06] J. Setoain, C. Tenllado, M. Prieto, D. Valencia, A. Plaza, and J. Plaza. Parallel Hyperspectral Image Processing on Commodity Graphics Hardware. In *ICPPW '06: Proceedings of the 2006 International Conference Workshops on Parallel Processing*, pages 465–472, Washington, DC, USA, 2006. IEEE Computer Society

[SPT+07] J. Setoain, M. Prieto, C. Tenllado, A. Plaza, and F. Tirado. Parallel Morphological Endmember Extraction Using GPUs. *IEEE Geoscience and Remote Sensing Letters*, pages 441–445, 2007

[SPTP08] J. Setoain, M. Prieto, C. Tenllado, and A. Plaza. GPUs for Parallel on-board Hyperspectral Image Processing. *International Journal of High Performance Computing Applications*, pages 424–437, 2008

[TSPT08] C. Tenllado, J. Setoain, M. Prieto, and F. Tirado. 2D-DWT on GPUs: Filter-Bank versus Lifting. *IEEE Transaction on Parallel and Distributed Systems*, pages 299–309, 2008

[SPTT08] J. Setoain, M. Prieto, C. Tenllado, and F. Tirado. Real-Time On-Board Hyperspectral Image Processing Using GPUs. In *High Performance Computing in Remote Sensing*, chapter 18, pages 411–451. Chapman And Hall, 2008

[STG+08] J. Setoain, C. Tenllado, J. I. Gomez, M. Arenaz, M. Prieto, and J. Tourino. Towards Automatic Code Generation for GPU Architectures. In *PARA'08: Workshop on State-of-the-Art in Scientific and Parallel Computing*, 2008

Como conclusión general, encontramos las unidades de procesamiento gráfico como plataformas de gran interés para el procesamiento paralelo de grandes cantidades de datos, así como para nuestro caso particular: el análisis de imágenes hiperespectrales. Con un rendimiento por Vatio muy superior a los procesadores de propósito general, pero un volumen de mercado que permite precios similares, son una solución que hay tener muy en cuenta para el cómputo científico tanto en la actualidad como en el futuro.

5.1. Trabajo futuro

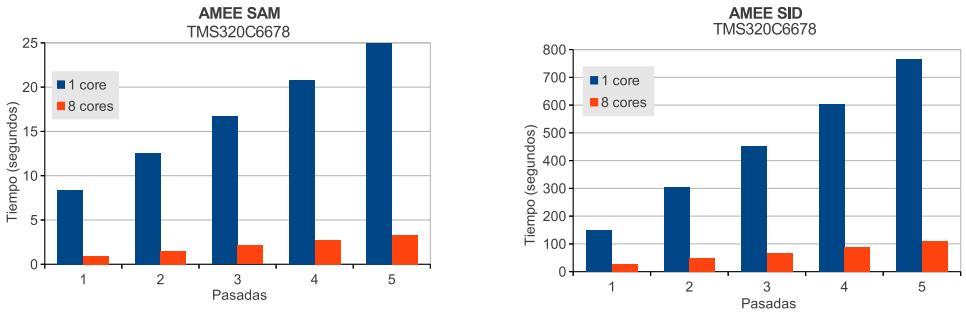
A lo largo del desarrollo de este estudio se ha hecho evidente la velocidad a la que cambian tanto las arquitecturas de las *GPUs* como los entornos de desarrollo destinados a su explotación para cómputo de altas prestaciones.

Estudios genéricos para aplicaciones específicas se hacen inviables, ya que con cada cambio en la arquitectura y con cada nueva versión del entorno, las posibilidades aumentan y los resultados varían.

Por el lado del entorno de desarrollo, dada la imposibilidad actual de apoyarnos en compiladores de manera eficaz, una posibilidad más interesante es el desarrollo de bibliotecas de primitivas auto-optimizables, tal y como comentábamos al final del capítulo 4. Aunque el entorno *CUDA* presente una *API* que cambia con cada nueva arquitectura, cambiando los parámetros del rendimiento, también está disponible el entorno *OpenCL* [Gro08], con una interfaz de programación mucho más estable. Esto facilita la posibilidad de utilizar una aproximación similar a *ATLAS*, en la que se ofrezcan una serie de primitivas de cómputo que busquen de manera automática, y específica para cada sistema, los parámetros óptimos de funcionamiento.

Por el lado de la plataforma, en el caso específico del análisis de imágenes hiperespectrales y dadas las restricciones que existen para poder embarcar *hardware* en un satélite –tanto de energía como certificaciones de resistencia al entorno– los *DSP* multi-núcleo que han aparecido, como el *TMS320C6678* [TI12], abren todo un abanico de posibilidades para el procesamiento de imágenes hiperespectrales en satélite. Además, gracias a que entornos de desarrollo como *OpenCL* se están portando a todo tipo de dispositivos multi-núcleo, cualquier esfuerzo hecho en la implementación de aplicaciones en *GPUs* usando primitivas auto-optimizables sería directamente aprovechable en este tipo de dispositivos multi-núcleo.

A modo de prueba, hemos implementado una versión preliminar de *AMEE* sobre esta plataforma.



(a) (b)

FIGURA 5.1: Resultados de *AMEE* sobre el *DSP TMS320C6678* de Texas Instruments para una imagen hiperespectral de 64 MiB.

En la figura 5.1 mostramos los resultados para *AMEE* con las distancias *SAM* (5.1(a)) y *SID* (5.1(b)) sobre el *DSP TMS320C6678* de Texas Instruments. Para paralelizar el algoritmo hemos utilizado el soporte para *OpenMP* que nos proporciona el compilador de *Texas Instruments*, y para dar una idea de la escalabilidad se muestran resultados utilizando un único núcleo y los ocho disponibles en el *DSP*. Como podemos observar, incluso utilizando herramientas estándar de paralelización como *OpenMP* se alcanza una buena escalabilidad del algoritmo en esta plataforma. Es esperable que con un esfuerzo más dirigido a la plataforma obtengamos resultados significativamente mejores.

Este tipo de dispositivos posee dos ventajas frente a las *GPUs*. Por un lado, su consumo es mucho más limitado; por otro lado, estos *DSP* están certificados para el espacio. Estas dos ventajas lo convierten en una solución ideal para explorar aplicaciones de análisis de imágenes hiperespectrales en satélites, dado que pueden ser utilizados de manera inmediata en estos entornos.

En resumen, usar los nuevos entornos de desarrollo como *OpenCL*, que no

son específicos para *GPUs*, y enfocar los esfuerzos de optimización en primitivas de alto nivel con las que luego construir las aplicaciones concretas, resulta una mejor opción usar que entornos específicos para *GPUs* y dedicar los esfuerzos a optimizar cada aplicación. De cara a la plataforma, los *DSP* multi-núcleo como *TMS320C6678*, gracias a su consumo reducido y la certificación para el espacio, son una opción muy prometedora como procesadores de altas prestaciones para entornos embarcados.

Apéndice A

Hyperspectral Imaging on GPUs: A Summary in English

This chapter is a summary of the PhD. thesis titled "*Hyperspectral Imaging on GPUs*". The summary is structured as follows. In Section A.1 we introduce this thesis, presenting the motivation and objectives, as well as a small summary of the state of the art. In Section A.2 we summarize what hyperspectral imaging is about and we describe the algorithms and problems we have been working on along this thesis. Section A.3 is a description of how we exploit *GPUs* through *OpenGL* to solve our hyperspectral imaging problems, together with the results we obtained. In Section A.4 we do the same, but using *CUDA* instead of *OpenGL* to exploit *GPUs* for hyperspectral imaging. To conclude, in Section A.5 we present the main conclusions of this work as well as the main publications supporting it.

A.1. Introduction

We start by introducing the main concepts we have been dealing with in this thesis along with the motivations that drove our work. In order to set it all in context, we start with a summary of the state of the art.

Evolution of Parallel Architectures

In 1965, Gordon Moore, one of Intel's co-founder, pointed out that thanks to the advances achieved in integration technology, they were capable of doubling the number of transistors integrated per chip every 18 months. This statement has been know as "Moore's Law" [Moo65], and it has been carried out almost strictly until today, as shown in figure A.1.

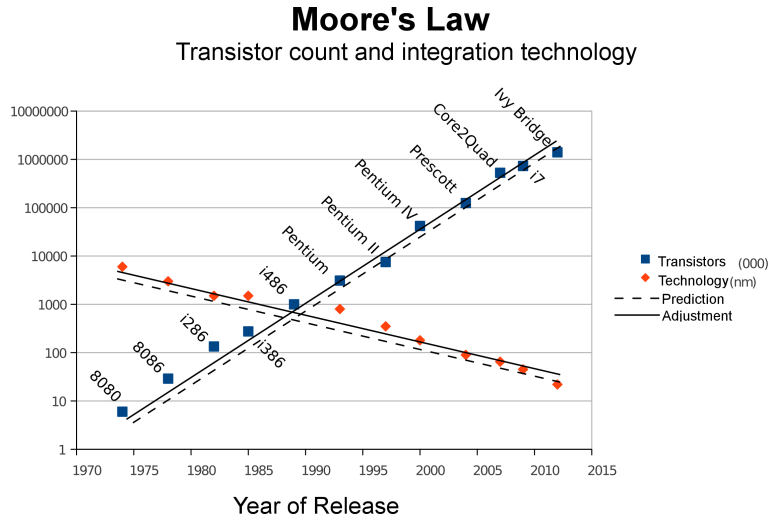


FIGURA A.1: Prediction and adjustment of transistor count and integration scale to Moore's Law.

Processors designers have been using the increasing number of transistors trying to comply with the following idea: a computer consists of a single processor executing a sequential flow of instructions that is connected to a monolithic

memory containing all data and instructions. This abstract computer model is known as a Von Neumann architecture (see figure A.2).

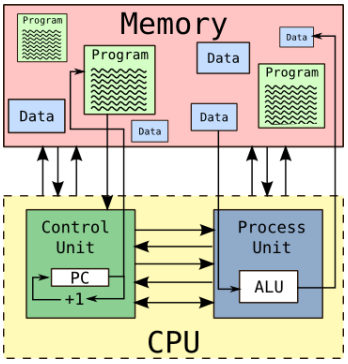


FIGURA A.2: In the architecture proposed by John Von Neumann, a computer consists of a memory containing data as well as programs, and a CPU in charge of running the programs on that data. A CPU is made of one control unit, which manages the execution flow of programs, and one processing unit, which manages data processing. In this figure, *PC* is the *Program Counter*, in charge of following the instruction flow of a program. The *ALU* (*Arithmetic-Logic Unit*) performs arithmetic and logic operations on data.

This way we avoid the need for retraining programmers and rewriting programs, and so computer architects have developed multiple techniques that allowed them to use the ever increasing number of transistors on chip to increase processor performance as well. Unfortunately, these techniques have met their limits, and further exploitation does not result in a much better performance.

In order to maintain the same rate of growth in performance as in the number of transistors per chip, the need for a paradigm shift became apparent. Processor designers could no longer observe the Von Neumann abstraction and so they developed the concept of *Chip Multi-Processors* (CMP). In these designs several processing cores are integrated in the same chip. They are also known as *multicore* or, in case we are speaking of a lot of small simple

processing cores, *manycore* architectures.

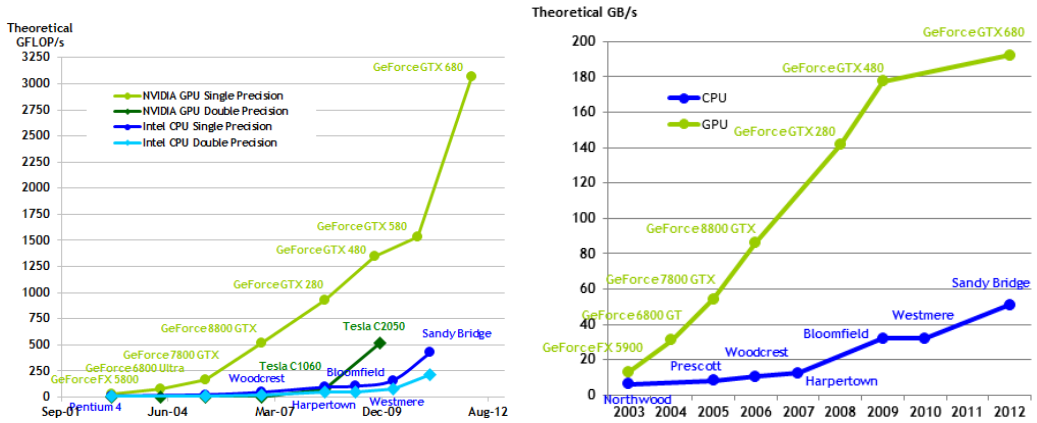
For processor designers, this change implies dealing with a greater pressure on memory and I/O buses as the number of processing cores increases. On the bright side, scaling through the upcoming technologies becomes easier and cheaper just by adding more cores to the *CMP*, and market economy can be satisfied by varying clock frequencies as well as the number of cores integrated: high-end products integrate a higher number of faster processing cores while low-end products integrate fewer and slower processing cores.

The essential problem with this new paradigm have to be faced by programmers, which are now in charge of exposing the implicit parallelism in their applications. Ideally, development tools (IDEs, compilers, CASE tools, etc) should be able to expose this parallelism all by themselves but, to this day, technology is far from that. Programmers need training in parallel programming and its associated problems and they have to rewrite their algorithms, otherwise the performance in their applications will stall.

This is also a problem for processor vendors as they will see their market shrink if they fail to provide the adequate solutions so that developers can take advantage of the upcoming processors. In order to ease the task of facing these new platforms, both industry and academia have proposed several solutions. Such are the cases of *CUDA* and *Brook+* [BFH+03] for general purpose computing on graphics processing units, *CellSs* [PBBL07] for *Cell BE* programming or *OpenCL* [Gro08] and, more recently, *OpenACC* [Ope11] for heterogeneous parallel architectures in general.

GPUs for general purpose and scientific computing

Graphics processing units (or *GPUs*) are among the most widely adopted *CMPs*. Originally conceived for real-time computer graphics synthesis, *GPUs* have become ubiquitous as the videogame industry grew to surpass the rest of the entertainment industry. This growth has become a powerful stimulus for investment in *GPU* architecture development, to the point of making *GPUs* raw processing power growing faster than that of *CPUs* themselves, as shown in figure A.3.



(a) (b)

FIGURA A.3: The difference in computing power and bandwidth between *GPUs* and *CPUs* has been increasing throughout the years (figures from *CUDA* programming manual [nvi12]).

Nowadays, there are *GPUs* available in the market that reach peak theoretical performances in the order of teraflops. That, together with their ubiquity, make *GPUs* a very interesting opportunity towards high performance application development.

From graphics synthesis to scientific computing

At first, *GPUs* were designed to accelerate real-time 2D image synthesis from scenes represented by 3D geometry. In *Computer Graphics*, a 3D scene comprises geometry, lights and a viewpoint or camera from which we will synthesize a 2D image of that scene (like taking a photograph of the scene).

There are several ways to transform a 3D scene in a 2D image, like *ray casting* [Rot82], *ray tracing* [App68] or *radiosity* [GTGB84]. But if our intention is generating those images in real time, the usual method is the one known as *scanline rendering and rasterization*, in which geometry is represented by groups of vertices building primitives (like triangles, squares or other polygons). These vertices are projected from the viewpoint and the area of the image enclosed by a projected primitive is filled as a function of certain properties contained in its vertices.

Graphics processing units implement what is known as a "graphics pipeline", which receives the geometry of the scene in one end and, stage by stage, transforms it into the synthetic image. The graphics pipeline usually implemented by graphics cards follows the schematic shown in figure A.4.

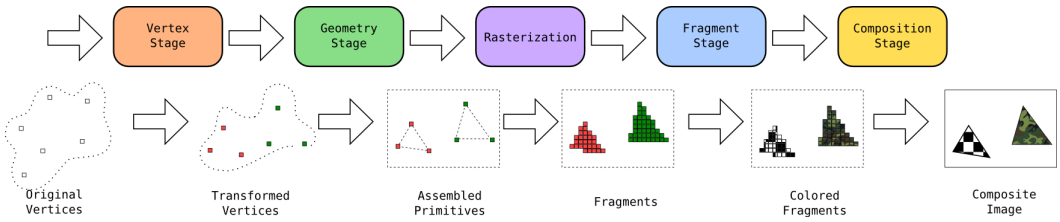


FIGURA A.4: Graphics pipeline. It takes as input a set of vertices and their attributes, and transforms them through consecutive stages to obtain an image of the scene represented by that geometry.

We can find five stages in a graphics pipeline:

1. *Vertex stage*. In this stage, the vertices of the scene to be represented come in along with their associated attributes. Generally, these attributes contain information about the geometry of the object they make up (normals, tangents, etc) and information about the material the object is made of (such as color and texture coordinates). Attributes related to geometric information are used to determine how lights in the scene illuminate the object. Attributes related to object's material are used to obtain the final color. Vertices are transformed one by one, independently, to place them in the scene. Their lighting is computed in relation to the lights in the scene, and then they are projected from the viewpoint. Generally, we modify the vertex attributes in relation to scene information, such as lights, camera and so on.

2. *Geometry stage*. In this stage, vertices will be grouped by primitives (segments, triangles, squares and all kinds of polygons), cropped on the frame of our visualization area (viewport) and assembled forming rasterizable geometry (usually triangles).

3. *Rasterization*. It's the key stage and the one that names this graphics synthesis technique. From primitives assembled in the previous stage, we determine the points in the screen enclosed by them, and we interpolate the vertex attributes for every one of those points, known as fragments. A fragment is associated with the corresponding (interpolated) attributes from the vertices of the polygon containing that fragment.

4. *Fragment stage*. In this stage, fragments generated by rasterization are colored according to their attributes, determining their raw aspect. As

in the case of vertices, this process is performed on each fragment independently.

5. *Composition stage*. In this stage, colored fragments are combined among them and with those that might happen to be in the visualization memory (frame buffer), in order to give the 2D image its final aspect. Typical effects performed by this stage include *Alpha Blending* (useful to simulate translucent materials) and *Anti-Aliasing* (to conceal sub-sampling artifacts, such as jagged edges, very common in a rasterized polygon).

In their beginnings, *GPUs* implemented the graphics pipeline as a series of fixed functions but, in the following generations, different stages increased their flexibility allowing users to replace the fixed functions by customized ones. The first one to be programmable was the vertex stage and, with the release of what's been know as fourth generation *GPUs* [FK03], the fragment stage became programmable as well. This custom functions are known as *vertex shaders* and *fragment shaders*.

Figure A.5 shows a generic block diagram for fourth generation *GPUs*. First, vertices are sent to vertex processors, which are in charge of the vertex stage. Having no dependencies between individual vertices, we can easily improve performance by just adding more processors of this kind. Once the geometry stage has assembled them in primitives the rasterizer finds the enclosed fragments and sends them to the fragment processors. Again, the processing of the fragments is completely independent from one another, and therefore we can increase fragment processing performance by adding more fragment processors. Usually, this kind of *GPUs*, where fragment and ver-

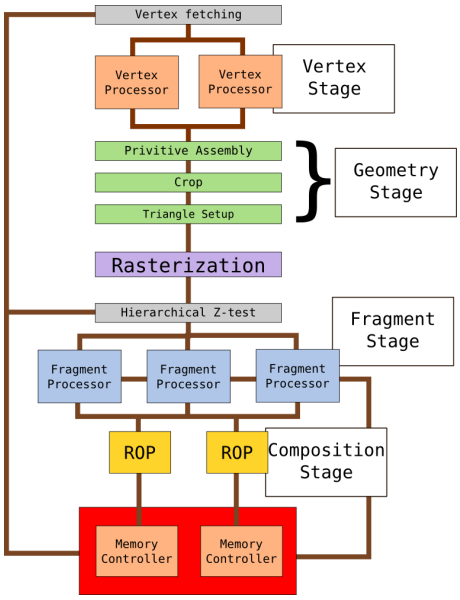
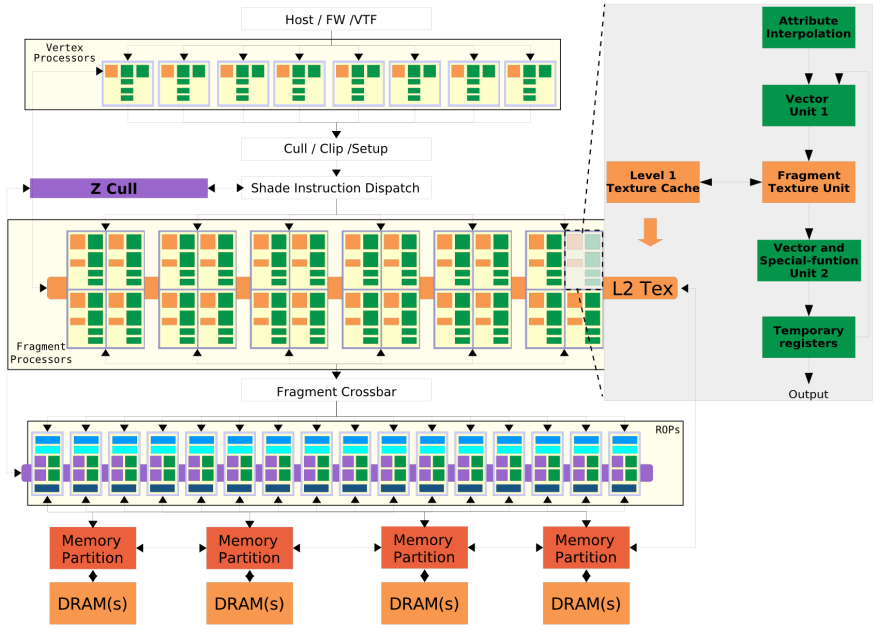


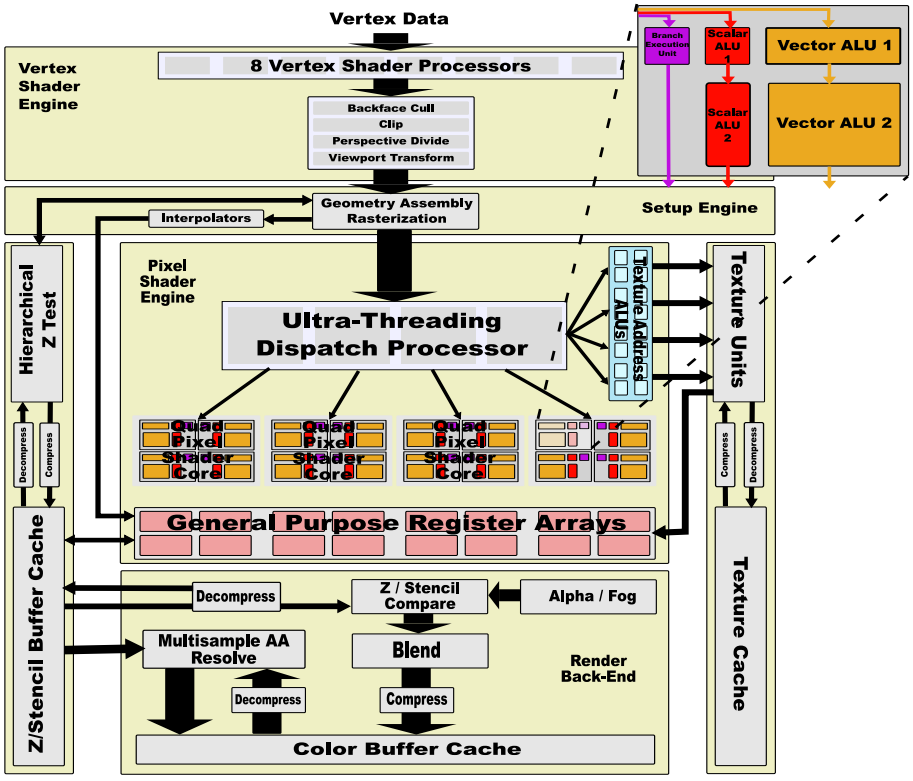
FIGURA A.5: Block diagram of a 4th generation GPU.

tex processors are differentiated (pack different functionality and have access to different resources), include a higher number of fragment processors than vertex processors. This is due to the fact that, typically, a small number of vertices will generate a larger number of fragment. So, in order to maintain a balanced load on both fragment and vertex processors, we need a higher number of the former. At last, in the composition stage, colored fragments are composed to make the final image. *GPUs* have specialized logic available for that purpose, the Raster Operation Processors or *ROPs*. We can configure the *ROPs* to combine the color of incoming fragments with that of those already in memory in order to obtain the final pixels of the image.

We can see the block diagrams of two fourth generation *GPUs*, a *NVIDIA G70* and a *ATI R520*, in figure A.6. We can see how both *GPUs* offer similar designs, with programmable vertex and fragment processors, a rasterizer and dedicated hardware for the geometry stage (*Setup Engine* in *R520* and *Cull/-*



(a)



(b)

FIGURA A.6: Block diagram of (a) NVIDIA G70 and (b) ATI-Radeon R520 GPUs.

Clip/Setup in *G70*) and, finally, hardware for the composition stage (*Render Back-End* in *R520* and *ROP* in *NVIDIA*'s *GPU*).

From fifth generation on, the internal structure of *GPUs* undergoes a dramatic change: unified architectures. Differentiation between vertex and fragment processors has vanished, and vector units have been replaced by a higher number of scalar units grouped forming a special kind of *VLIW* processor (*Very Long Instruction Word*). These processors are capable of executing the same instruction from different threads simultaneously. This capability is known as *Single Instruction, Multiple-Threads* or *SIMT*, and it allows having a higher number of threads running simultaneously which helps to hide the high latencies of memory accesses. Moreover, as they have a more generalized architecture, they can run all kinds of programs, including those of the geometry stage, making it effectively programmable in these *GPUs*.

Figure A.7 shows the diagram of a *GPU* with unified architecture. In the figure, we see 8 blocks of two multicore processors with 8 cores each, each one of them coupled with a texture fetch unit set and its own first level *cache* memory. This accounts for a total of 128 processors integrated on chip.

Usually, *GPUs* make computations on fixed point and single precision floating point numbers, but later evolutions of unified architectures added double precision floating point capabilities as well as several atomic instructions.

The generalization from a very specific architecture for computer graphics to a *SIMT* architecture with certain specific units for graphics synthesis (rasterizer, texture units, *ROPs*, etc), eases the exploitation of *GPUs* for non-graphics computing. When *GPU* manufacturers recognized the opportunity that *GPUs* as high performance computers present, they begun to distribute

FIGURA A.7: Block diagram of an NVIDIA G80 GPU.

tools and solutions for this emerging market. Such is the case of *NVIDIA CUDA* (*Compute Unified Device Architecture*), whose possibilities and peculiarities are explored in Section A.4.

Hyperspectral imaging

Hyperspectral images are those with a spectral resolution much higher than that of regular images. In a regular color image we will see three spectral bands: red (wavelength of $625-740nm$), green (wavelength of $520-570nm$) and blue (wavelength of $440-490nm$). On the other hand, in a hyperspectral image we can find from hundreds to thousands of spectral bands, with wavelengths ranging from ultraviolet (less than $400nm$) to infrared (more than $750nm$). Usually, bands will be uniformly distributed throughout the spectrum covered by the sensors, a fundamental difference with multispectral images, which usually target fewer and more specific wavelengths [VGC+93, KLC04] (for instance: red, green, blue, near infrared, medium infrared and radar).

Such a greater spectral resolution compared to regular color images implies that pixels in a hyperspectral image contain hundreds of components instead of just the three of regular images. That's why, when we refer to pixels in hyperspectral images, we call them pixel-vectors or, on a more general way, the spectral signature of a point in the image. This is the terminology we will follow from now on.

Thanks to the high spectral resolution of hyperspectral images, we can perform spectrographic analysis on pixel-vectors. This could be useful for many different applications both military and civil: crop growth control, environmental monitoring, open-air mining, detection and identification of man-made structures, surveillance, and so on [SM02, SB03].

Behind all these applications lies the fact that every material reflect electromagnetic radiation in specific wavelengths, following distinctive patterns according to their compositions. With the adequate spectral information we can distinguish and identify those materials (see figure A.8).

Motivations

Despite of all the potential applications for hyperspectral images, the massive amount of data that one of these images can contain limits their scope to applications without real-time requirements nor storage capacity limitations. If, for instance, we work with images of $1m^2$ per pixel of spacial resolution and 256 bands of spectral resolution, a single hyperspectral image of a $4km^2$ region would weight almost $2GiB$. Besides, processing all that information requires a significant amount of computing power which, in turn, would require

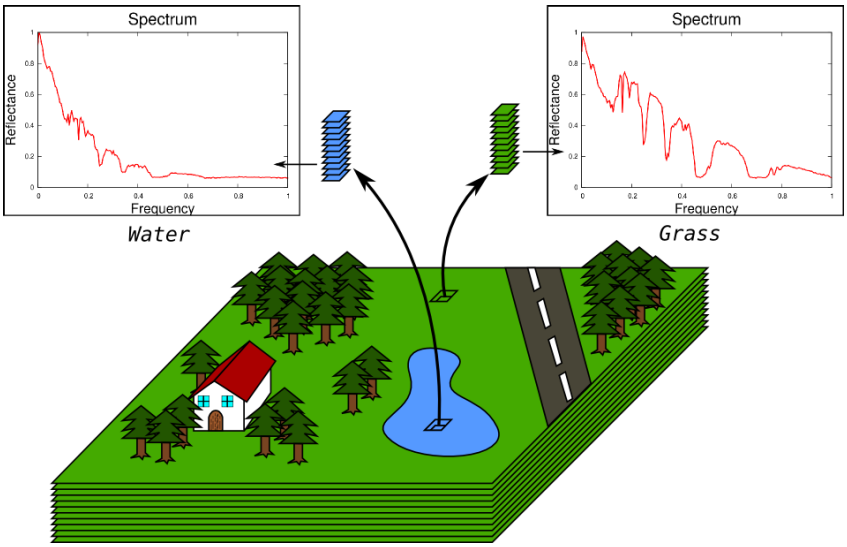


FIGURA A.8: The spectral resolution in a hyperspectral image allows us to identify materials in the image from their spectra.

big computers with a great power consumption and/or a significant amount of computing time.

Nevertheless, there are certain applications which could benefit from hyperspectral imaging if we get rid of these limitations. One example would be oil spills in the ocean. The spill floats a few centimeters under the water, which makes it invisible to regular cameras but perfectly visible to hyperspectral sensors. Another example would be ground level forest fires, where tree tops cover the fire and make it hard to follow at sight, but easy to see in a hyperspectral image. In both cases we need a quick response, a processed image that'd show us the state and evolution of these kind of fast changing phenomena. Hyperspectral imaging would be useless if processing the information takes several ours (or even days).

Another example for which the great size of hyperspectral images could be a problem is satellite remote sensing. Satellites have limited storage space and

data transmission bandwidth, as well as a limited power source available. If every single image weights in the order of gigabytes, a satellite couldn't storage many of them at a time. This could also be a problem for airborne remote sensing on small planes or *UAVs* (*Unmanned Aerial Vehicles*). Albeit, an already processed hyperspectral use several orders of magnitude less of space, which would speed up transmissions and allow for longer mission durations.

In oder to avoid this problems we need not only high computing power (for real time processing) but also that this power fits within certain space and energy restrictions.

Here is where *GPUs* arise as a promising solution. *GPUs* provide a quite significant computing power in a reduce volume and having relatively low energy requirements compared to more classical solutions such as computer clusters. In addition, thanks to their ubiquity due to great market situation that videogame industry is enjoying, we have these platforms available at prices significantly lower than almost any other high performance solution.

Objectives

Our main objective is to assess the viability of these platforms as boardable high performance computing solutions that might provide real-time hyperspectral imaging, creating a wide range of possible solutions unfeasible nowadays.

The objectives of this thesis are:

- Analyze the existing parallelism in the automated morphological *endmember* extraction algorithm [PMPP02] using two different distance functions: *SID* and *SAM*.

- Developing a framework to program *GPUs* following the *stream processing model* using *OpenGL* as an underlying *GPU* access library.

- Port the hyperspectral imaging algorithms to the *stream processing model* and implementing those algorithms efficiently on *GPUs* using the framework we developed for that purpose.

- Port the hyperspectral imaging algorithms to *CUDA* development environment and analyze the best approach for doing so.

- Analyze the performance of those algorithms on both platforms, *CUDA* and our *GPGPU framework*, and assess their suitability as real-time hyperspectral imaging solution.

The rest of this summary is organized as follows:

In Section A.2 we introduce the algorithms we have been working with. In Section A.3 we describe our framework for *GPGPU* on *OpenGL* and discuss the mapping of hyperspectral imaging algorithms on it. In Section A.4 we analyze the implementation of our algorithms on *CUDA*. Contributions of the thesis, possible future directions for this research, as well as publication work of the thesis are finally presented in Section A.5.

A.2. Hyperspectral Imaging

As we mention above, hyperspectral images are useful for spectrographic analysis. Thanks to their high spectral resolution, we can determine what materials are the pixels in our images made of. One of the essential task in

most hyperspectral imaging applications is finding the pixel-vectors with the purest spectra, known as *end-members*.

The main applications in hyperspectral imaging are the following three:

1. *Identification:* We compare the spectral signature of *end-members* against a database containing the spectra of different materials, identifying the specific material for each *end-member*.

2. *Un-mixing:* We decompose every pixel-vector in the image in a (usually) linear combination of the *end-members*.

3. *Classification:* We localize regions in the image that belong to the class of one *end-member*, delimiting areas dominated by a specific material.

Finding the *end-members* is not only central to almost any hyperspectral imaging application, but also the most time consuming task. In order to find the purest pixel-vector we have to, somehow, compare them with each other. Given the size of hyperspectral images, this takes a great deal of mathematical operations.

We can classify the methods for finding *end-members* in two groups:

1. Supervised: The application requires of some kind external or user intervention.

2. Unsupervised: The application performs the analysis with data from the sensors, without any intervention from the user.

For real-time hyperspectral imaging, supervised algorithms would be of no use; we need an autonomous algorithm, that is, an unsupervised *end-member* finding algorithm.

The *AMEE* algorithm (*Automated Morphological End-member Extraction*), besides meeting the requirement of being unsupervised, was designed for computational efficiency [PMPP02]. It is a highly parallel algorithm so it is able to take advantage of the massive parallelism available in our platforms of study.

Another distinctive feature of this algorithm is the use of mathematical morphology to perform its analysis. There are methods that perform the analysis based on the spectral information in the pixel-vectors, while other methods perform the analysis based on spacial information. However, *AMEE* uses mathematical morphology to correlate spacial and spectral information, taking advantage of both in order to find the *end-members*.

Regarding accuracy, in Section A.2.4 we show that *AMEE* is superior to other *end-member* finding algorithms in most cases.

Besides, with some modifications (explained in Section A.2.2) we can tune certain parameters that allow us to choose a trade-off between performance and accuracy, which is very useful for real-time scenarios.

Because of all these features: efficiency, parallelism, accuracy and the possibility of adjusting the trade-off between accuracy and performance; we decided to focus on the implementation and optimization on *GPUs* of the *AMEE* algorithm for *end-member* finding.

A.2.1. The *AMEE* algorithm

We define $\boldsymbol{f}$ as a set of hyperspectral data in an n-dimensional space ($\mathbb{K}^n$) where n is the number of channels or spectral bands:

$$\boldsymbol{f}: \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{K}^n$$

The main concept behind the *AMEE* algorithm is defining an order relation, in terms of spectral purity, on the set of pixel-vectors inside of a spacial searching window -or structuring element- around every pixel-vector in the image [Soi03].

With that purpose in mind, we begin by defining the cumulative distance between one specific pixel $\boldsymbol{f}(x, y)$, i.e.: a vector $\mathbb{K}^n$ in the discrete spacial coordinates (x, y), and every pixel-vector in the spacial neighborhood defined by B (B-neighborhood) as the following equation [PMPP02]:

$$D_B(\boldsymbol{f}(x, y)) = \sum_{(i,j)\in\mathbb{Z}^2(B)} Dist(\boldsymbol{f}(x, y), \boldsymbol{f}(i, j)) \tag{A.1}$$

where (i, j) are the spacial coordinates in the discrete domain of the B-neighborhood, represented by $\mathbb{Z}^2(B)$, and *Dist* is the measure of the distance between two vectors $\mathbb{K}^n$. Later on, in Section A.2.3, we discuss the election of the *Dist* function, as it is a key element in establishing the order relation and, therefore, the algorithm's accuracy.

We use the cumulative distance, inside the structuring element, as a measurement of how representative of the set defined by the structuring element is a pixel-vector. This way, we can choose the most representative and the less representative (those with the highest and lowest cumulative distance, respectively) in order to compute the *Morphological Eccentricity Index* or *MEI* for the pixel-vector in the center of the structuring element.

From the cumulative distance, we can define the morphological erosion as:

$$(f \ominus B)(x, y) = argmin_{(i,j)\in\mathbb{Z}^2(B)}\{D_B[f(x + i, y + j)]\} \tag{A.2}$$

where operator $argmin_{(i,j)\mathbb{Z}^2(B)}$ chooses the pixel-vector with the lowest cumulative distance inside the structuring element.

Likewise, we can define the morphological dilation as:

$$(f \oplus B)(x,y) = argmax_{(i,j)\in\mathbb{Z}^2(B)}\{D_B[f(x+i,y+j)]\} \tag{A.3}$$

where operator $argmax_{(i,j)\in\mathbb{Z}^2(B)}$ chooses the pixel-vector with the highest cumulative distance inside the structuring element.

And so, from equations A.2 and A.3, and relying on the distance function *Dist*, we can define the *morphological eccentricity index* (*MEI*) of a pixel-vector as the extended morphological gradient of the image in the position of that pixel-vector:

$$\begin{aligned} MEI_B(f(x,y)) &= (f \oplus B)(x,y) - (f \ominus B)(x,y) \\ &= Dist((f \ominus B)(x,y), (f \oplus B)(x,y)) \\ &= Dist(argmin_{(i,j)\in\mathbb{Z}^2(B)}\{D_B[f(x+i,y+j)]\}, \\ &\quad argmax_{(i,j)\in\mathbb{Z}^2(B)}\{D_B[f(x+i,y+j)]\}) \end{aligned} \tag{A.4}$$

The *MEI* of a pixel-vector represents the spectral purity of that pixel-vector, and we can use it to find the pixel-vector with the purest spectrum in the image; i.e.: our first *end-member*.

The remaining *end-members* can be found from there on, choosing the pixel-vector with the highest distance to the first one, then the one with the highest cumulative distance to those two, and so on and so forth until we find all the *end-members*.

At this point, the *AMEE* algorithm for finding the first *end-member* would be:

First, for every pixel-vector (x, y) in the spacial domain of $\boldsymbol{f}$:

1. Center the structuring element $\mathbb{Z}^2(B)$ at (x, y).

2. As shown in figure A.9, compute the cumulative distance of every pixel-vector with each other inside the B-neighborhood.

3. Find the pixel-vectors with highest and lowest cumulative distance inside the structuring element. That is, the erosion (equation A.2) and dilation (equation A.3) of the image.

4. Compute the *MEI* score of the image in that point; that is, the distance between the pixel-vector with the highest cumulative distance and the pixel-vector with the lowest cumulative distance (see figure A.10), which is equivalent to the extended morphological gradient (equation A.4) of the image at (x, y).

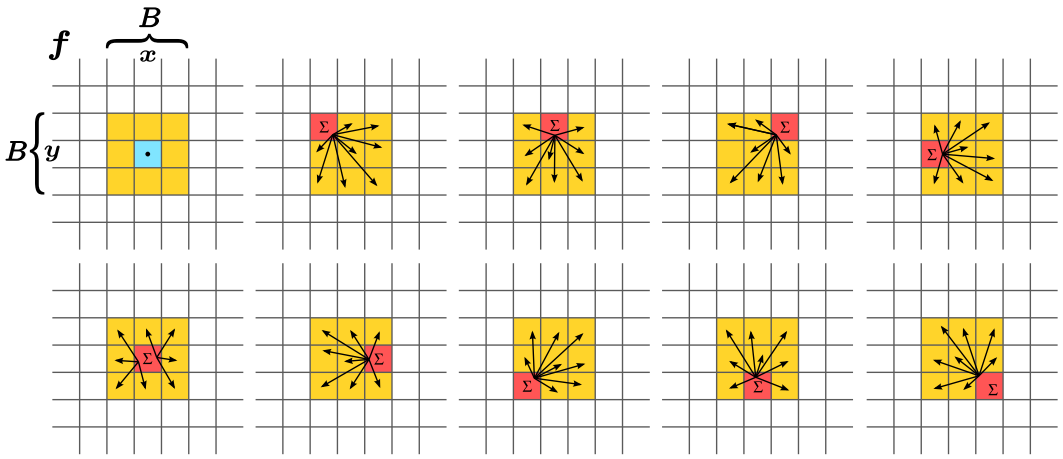


FIGURA A.9: Cumulative distance of the pixel-vectors given a structuring element $\mathbb{Z}^2(B)$ centered in (x, y).

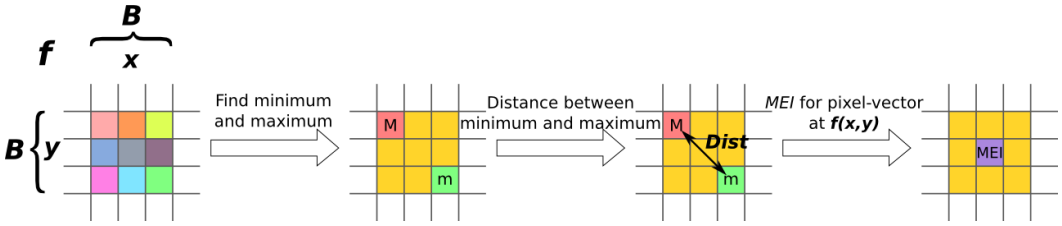


FIGURA A.10: Finding the pixel-vectors with maximum and minimum cumulative distance, and *MEI* computation for the pixel-vector $\boldsymbol{f}(x, y)$.

In order to obtain the *end-members* from a *MEI* map, among the pixel-vectors with the highest associated *MEI* score we can find the q most orthogonal to one another. We start by adding the pixel-vector with the highest associated *MEI* score of all to our set of *end-members*, and then, iteratively, we continue adding the next pixel-vector with highest *Dist* to the current set. In the end, we get a unique spectral set of $e_{i\,i=1}^{q}$ pixel-vectors as *end-members*.

From the algorithm description we draw two conclusions:

1. The size of the structuring element establishes a trade-off between accuracy and performance. The bigger the size the higher the accuracy, as each pixel-vector is compared with more of them during *MEI* map computation. The smaller the size, the higher the performance, as the number of computations needed to build the *MEI* map grows in the order of the fourth power with the size of the chosen B-neighborhood. Given a cost of K_λ for the computation of *Dist* between two pixel-vectors with λ spectral bands, computing the *MEI* map for an image of $W \times H$ has a cost in the order of $K_\lambda \cdot W \cdot H \cdot (B^4 - B^2 + 1)$.

2. *MEI* map computation and so finding the first *end-member* is the most time consuming part of the algorithm. In case of an image of $W \times H$,

finding λ-*end-members* has a cost in the order of $K_\lambda \cdot W \cdot H \cdot (B^4 - B^2 + 1)$ for the first *end-members*, and $(n-1) \cdot K_\lambda \cdot W \cdot H$ for the rest of them. Generally, even for the lowest values of B, $(B^4 - B^2 + 1) \gg (n-1)$ so the time needed to find the first *end-member* is much higher than the time needed to find the rest.

Because of the second conclusion, in this thesis we have focused on the *MEI* map computation given that, algorithmically, it shows the highest complexity and consumes most of the time needed to find the *end-members*.

A.2.2. *AMEE*: Multi-pass *MEI* map

As we have noted above, choosing the structuring element size ultimately influences the performance as well as the accuracy, and that is why it is interesting to consider some improvements over the original algorithm.

Experimentally, we can observe that the MEI_B map, obtained using a structuring element $\mathbb{Z}^2(B)$, provides an accuracy similar to the one obtained by accumulating on the MEI_3 map the $\frac{B-3}{2}$ successive MEI_3 maps obtained from the $\frac{B-3}{2}$ successive morphological dilations of the original image.

So, if we define the i-th dilation of $\boldsymbol{f}$ as:

$$(f \oplus B)^i = f \overbrace{\oplus B \cdots \oplus B}^{i}$$

We can define multi-pass *MEI* map as:

$$MEI_B(\boldsymbol{f}) \approxeq \sum_{i=0}^{\frac{B-3}{2}} MEI_3((f \oplus 3)^i) \tag{A.5}$$

Therefore, after computing an initial MEI_3, we shall add to the algorithm a stage that computes the morphological dilation on the image, accumulating its MEI_3 and repeating this operation until the $\frac{B-3}{2}$-th dilation. In the end, we replace an algorithm to compute one MEI_B map by an algorithm to compute $\frac{B-3}{2}+1$ MEI_3 maps.

Analyzing the performance of both version we see that the cost of computing one MEI_B is in the order of:

$$K_\lambda \cdot W \cdot H \cdot (B^4 - B^2 + 1)$$

Being K_λ the cost of computing $Dist$ between two pixel-vectors with λ spectral bands, and $W \times H$ the dimensions of the image.

On the other hand, the cost of computing a MEI map equivalent to that MEI_B map by means of accumulating several MEI_3 maps is in the order of:

$$(\frac{B-3}{2}+1) \cdot K_\lambda \cdot W \cdot H \cdot (3^4 - 3^2 + 1)$$

Figure A.11 shows the *speed-up* obtained for different values of B. As we can see, it increases at a cubic rate with the size of the structuring element.

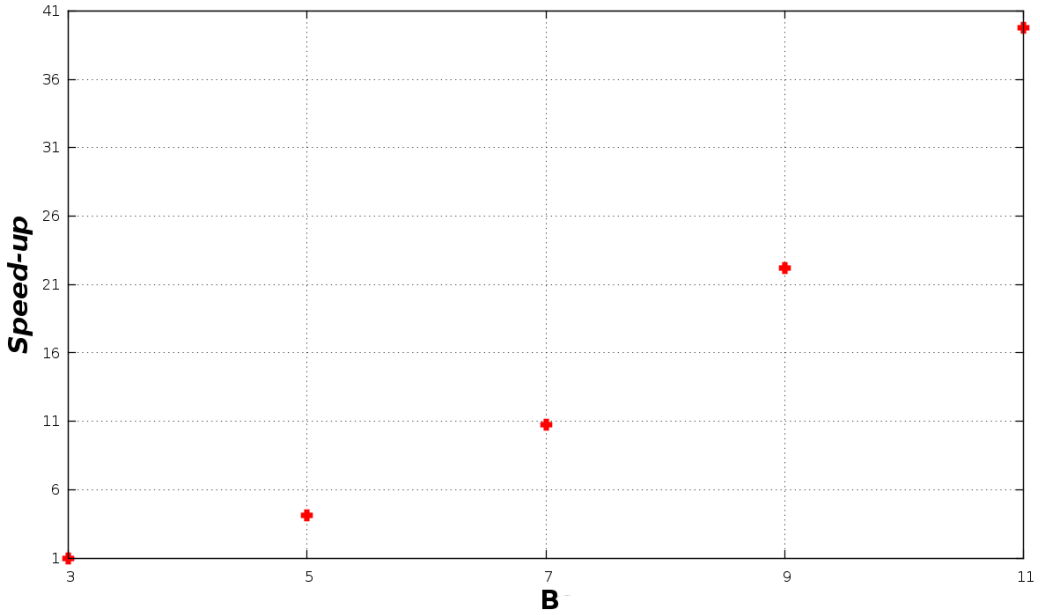


FIGURA A.11: Performance comparison between multi-pass *AMEE* and the original *AMEE*.

A.2.3. Distance function

Choosing the *Dist* function is a key issue for the order relation obtained. The *AMEE* algorithm uses the angular distance or *SAM* (*spectral angle mapper*) and the spectral information divergence or *SID*, both common measurements in hyperspectral imaging [Cha03].

SAM: Angular distance as distance measurement

In order to simplify, lets assume that $\boldsymbol{s}_i = (s_{i1}, s_{i2}, \cdots, s_{in})^T$ and $\boldsymbol{s}_j = (s_{j1}, s_{j2}, \cdots, s_{jn})^T$ are two spectral signatures in $\mathbb{K}^n$. In this case "spectral signature" is not necessarily the same as "pixel-vector" and that is why we leave out the spacial coordinates of previous examples, although the same

reasoning applies to pixel-vectors. The *SAM* between $\boldsymbol{s}_i$ and $\boldsymbol{s}_j$ would be:

$$SAM(s_i, s_j) = cos^{-1}(\frac{s_i \cdot s_j}{\|s_i\| \, \|s_j\|}) \tag{A.6}$$

Notice that *SAM* is invariant to constant multiplicative factors in the input vectors, and so it will be invariant to unknown scale factors that might arise from differences in illumination or the angle of the sensor. *SAM* gives us the angular distance between two spectral signatures in $\mathbb{K}^n$.

***SID*: Spectral information divergence as distance measurement**

SID distance is based on the concept of divergence and it measures the discrepancy of probabilistic behaviours between two spectral signatures. Given our two signatures $\boldsymbol{s}_i = (s_{i1}, s_{i2}, \cdots, s_{in})^T$ and $\boldsymbol{s}_j = (s_{j1}, s_{j2}, \cdots, s_{jn})^T$ in $\mathbb{K}^n$, assuming non-negative $\boldsymbol{s}_i$ and $\boldsymbol{s}_j$, we can define the following two probabilistic measurements:

$$\begin{aligned} M[s_{ik}] = p_k &= \frac{s_{ik}}{\sum_{l=1}^{n} s_{il}} \\ M[s_{jk}] = q_k &= \frac{s_{jk}}{\sum_{l=1}^{n} s_{jl}} \end{aligned} \tag{A.7}$$

From this definition, the self-information provided by s_j for the band l is given by $I_l(s_j) = -log\ q_l$. And so we can define the entropy of s_j with respect to s_i as:

$$\begin{aligned} D(s_i \| s_j) &= \sum_{l=1}^{n} p_l \; D_l(s_i \| s_j) \\ &= \sum_{l=1}^{n} p_l (I_l(s_j) - I_l(s_i)) \\ &= \sum_{l=1}^{n} p_l \; log \frac{p_l}{q_l} \end{aligned} \tag{A.8}$$

Using this equation (A.8), we define *SID* as:

$$\begin{aligned} SID(s_i, s_j) &= D(s_i \| s_j) + D(s_j \| s_i) \\ &= \sum_{l=1}^{n} p_l \; log \frac{p_l}{q_l} + \sum_{l=1}^{n} q_l \; log \frac{q_l}{p_l} \end{aligned} \tag{A.9}$$

Intuitively, *SID* gives us a measurement for the entropy between two pixel-vectors. In other words, the difference in the "amount" of information of one vector with respect to another. The greater difference the more "new" information one pixel-vector provides with respect to the other. The lower the difference the more representative of them both is any.

A.2.4. Accuracy

As a reference for the accuracy test, we take the image "*cuprite*", available from *AVIRIS* website [NASA].

TABLA A.1: Similarity scores for *AMEE SAM* and *AMEE SID* with different number of passes. Comparison with mineral spectral and *end-members* provided by *USGS*

	SAM AMEE			**SID AMEE**		
	$I_{max} = 1$	$I_{max} = 3$	$I_{max} = 5$	$I_{max} = 1$	$I_{max} = 3$	$I_{max} = 5$
Alunite	0.084	0.081	0.079	0.081	0.081	0.079
Buddingtonite	0.112	0.086	0.081	0.103	0.084	0.082
Calcite	0.106	0.102	0.093	0.101	0.095	0.090
Chlorite	0.122	0.110	0.096	0.112	0.106	0.084
Kaolinite	0.136	0.136	0.106	0.136	0.136	0.102
Jarosite	0.115	0.103	0.094	0.108	0.103	0.094
Montmorillonite	0.108	0.105	0.101	0.102	0.099	0.092
Muscovite	0.109	0.099	0.092	0.109	0.095	0.078
Nontronite	0.101	0.095	0.090	0.101	0.092	0.085
Pyrophilite	0.098	0.092	0.079	0.095	0.086	0.071

TABLA A.2: Similarity scores for different methods using mineral spectral and *end-members* provided by *USGS* (*United States Geological Survey*)

	PPI	**N-FINDR**	**VCA**	**IEA**
Alunite	0.084	0.081	0.084	0.084
Buddingtonite	0.106	0.084	0.112	0.094
Calcite	0.105	0.105	0.093	0.110
Chlorite	0.125	0.136	0.096	0.096
Kaolinite	0.136	0.152	0.134	0.134
Jarosite	0.112	0.102	0.112	0.108
Montmorillonite	0.106	0.089	0.120	0.096
Muscovite	0.108	0.094	0.105	0.106
Nontronite	0.102	0.099	0.099	0.099
Pyrophilite	0.094	0.090	0.112	0.090

Table A.1 shows similarity results obtained using multi-pass *AMEE* with *SID* and *SAM* as distance functions. On the other hand, table A.2 shows similarity results using four other methods (*PPI*, *N-FINDR*, *VCA* and *IEA*) on the same scene. For *SAM* distance we appreciate how after the fifth pass we stay behind *N-FINDER* only in the detection of *montmorillonite*, and not by far. *AMEE* shows greater accuracy for all the other materials. This happens because all the other methods use spectral information exclusively whereas *AMEE* takes advantage of the spacial correlation.

For *SID* distance, though not very different to those obtained for *SAM*, we appreciate a small improvement in the results. This is most probably because the physical meaning of the spectral information divergence is a better fit for the distance problem we are dealing with. And even if the angular distance is a reasonable measurement too, its physical meaning does not exactly fit in what we are trying to measure. For instance, when we measure cumulative distances, one pixel-vector might have a cumulative distance of 360 degrees while another might have 720 degrees. In this case, we interpret 720 degrees as greater than 360 degrees. Even if it is not a valid metric space, we can experimentally verify that, interpreted this way, it is indeed a valid distance.

A.3. Hyperspectral Imaging on *OpenGL*

OpenGL is a programming library designed for graphics synthesis. A good way to take advantage of *GPU* processing power for general purpose computing (also known as *General Purpose computing on Graphics Processing Units* or *GPGPU*), is thinking of a *GPU* as a set of stream processors and building a stream processing framework on *OpenGL*.

In this section we explain what is the stream processing model and how can we use the facilities provided by *OpenGL* in order to build a *GPGPU* framework for stream processing. Based on this, we describe the implementation of *AMEE* with this *SDK* (using *SID* as well as *SAM* as distance functions) and discuss its performance.

Stream Processing Model

In the *stream processing model* the data is stored in ordered sequences of elements known as *streams* and the algorithms are defined by a special type of functions known as *kernels*, which take as input individual elements from *streams*.

A *kernel* function might have one or more input *streams* and one or more output *streams*. The *kernel* processes the individual elements in the *streams* by tuples, so we find an implicit loop executing the *kernel* in successive instances, each one taking one input tuple and producing one output tuple, until the input *streams* are finished.

The main benefit in this processing model is that, as feedback is not allowed, all *kernel* instances are free of interdependencies. This way, we can speed processing up by just adding more stream processors. The data parallelism is exposed between instances of the same *kernel*. The only drawback is that you have to put an effort into exposing the implicit parallelism in an algorithm by transforming it to the stream processing model.

In order to make it easier to understand, we describe a program in stream processing model using arrows to represent streams and boxes to represent *kernels*. In figure A.12 we can see an example of this representation. Although very simple, it might get as complicated as needed, with loops and conditions in the stream paths, for instance.

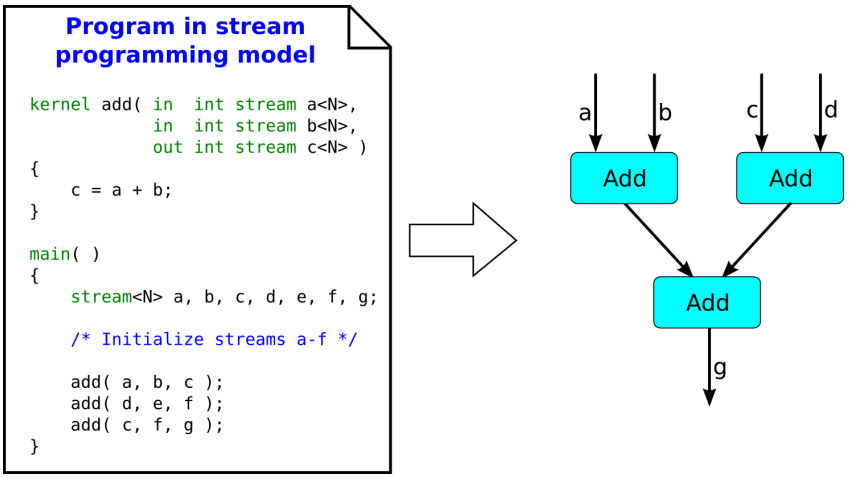


FIGURA A.12: Graphic representation of a program in the stream processing model.

GPGPU SDK on *OpenGL*

Our objective is building an abstraction layer to hide the details of the graphics library and the graphics subsystem of the operating system. This layer provides means to access the computational resources without need for advanced graphics synthesis knowledge. Figure A.13 shows a diagram of our *GPGPU SDK*.

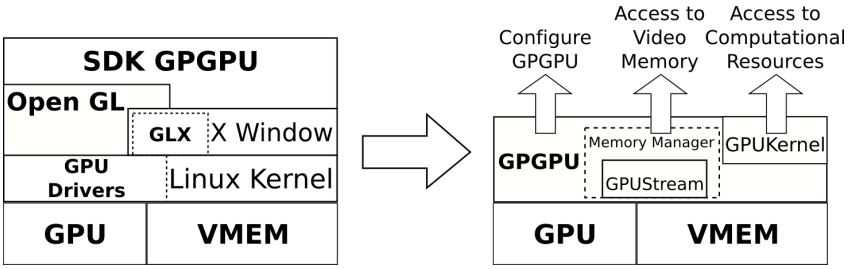


FIGURA A.13: Our *SDK* abstracts the complexity of the operating system and *OpenGL* with a development framework based on streams and *kernels*.

Our *GPGPU SDK* is a basic infrastructure that allow developers to build their applications in the stream processing model. To do so, we provide classes and methods to manage the following three functions needed:

- Environment management: Initialize, access to resources, synchronize and finalize.

- Memory management: Allocate, manipulate and free *streams*.

- Computational resources management: Build and run *kernels*.

Limitations

When porting algorithms to the stream processing model we need to take into account the following limitations in order to use our *GPGPU SDK*:

- *Kernels* are programmed in *Cg* [FK03]. Although most of the work related to graphics is transparent to programmers, a few concepts are necessary to program *kernels*. Fortunately, we can do this mechanically so we do not need to learn the whole discipline of computer graphics in depth.

- *Kernels* are limited to a single output *stream* and, at most, eight input *streams*. Due to the way the underlying *OpenGL* library utilizes memory in the *GPU*, we have to divide our program in *kernels* taking at most eight input streams and a single output stream.

- Data in *GPU* memory is stored as single precision floating point values (*float*). If we want to work with different types, we have to convert them to floating point and then, inside the *kernel* function, cast them back to the original type.

AMEE on *OpenGL*

The first issue we have to address is the mapping of the hyperspectral image onto the GPU memory. In order to fit chunks of processable data in *GPU* memory, we need to partition the image taking advantage of *AMEE's* spacial and spectral parallelism. The partition, shown in figure A.14, is as follows:

- In the spacial dimension: We divide our image in *"spacial regions"* (S.R.), so that a whole spacial region (with all its spectral information) fits in the *GPU* memory.
- In the spectral dimension: Every spacial region is divided in "blocks" holding four contiguous spectral bands each. This means that we have to rearrange the data before copying it into streams, but in turn we will be able to take advantage of vector operations.

For the purpose of dealing with frontiers, the spacial division has to consider the additional pixel-vectors needed to compute the *MEI* score of edge pixels. For multi-pass *AMEE*, we need two extra rows and two extra columns for each pass. Every pass discards the outermost edge of the region, but this operation can be performed at zero cost with our *SDK*.

If we transform the algorithm to get rid of redundant distance computations, we can reduce the number of distances per pixel-vector to 12. Numbering them from left to right and top to bottom, centered in $f(x, y)$, we see those distances in figure A.16. Computing those distances for every pixel-vector $f(x, y)$ is enough to obtain the 9 different cumulative distances inside an structuring element.

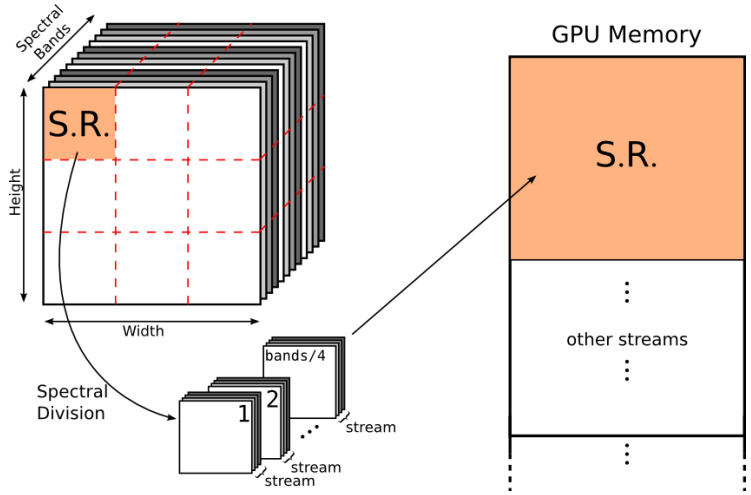


FIGURA A.14: The image is divided into spacial regions (S.R.) to make it fit inside the available video memory. In order to take advantage of the vector operations provided by the *GPU*, each spacial region is further divided into streams with four spectral bands each.

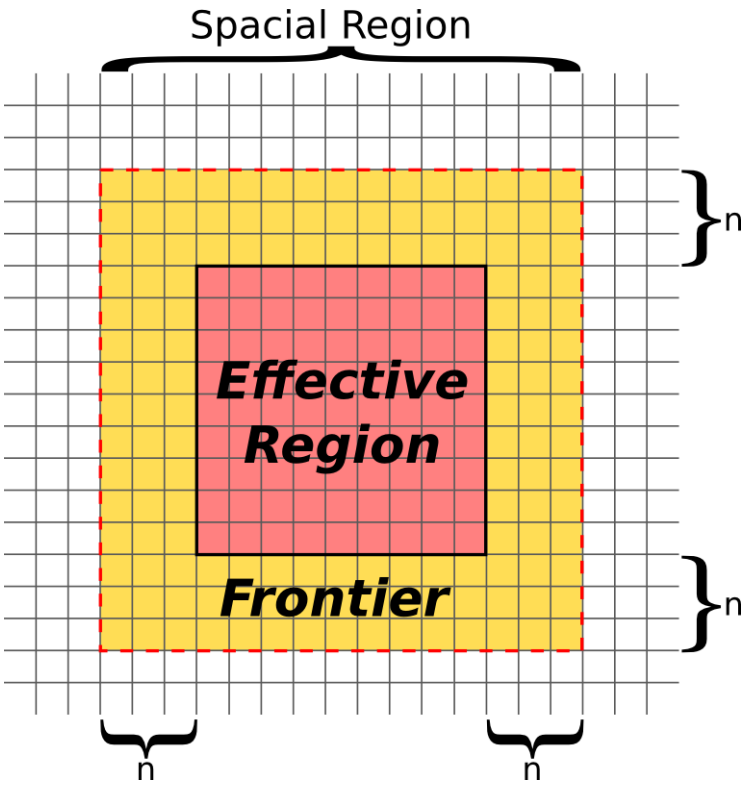


FIGURA A.15: Spacial region for n-pass *AMEE*

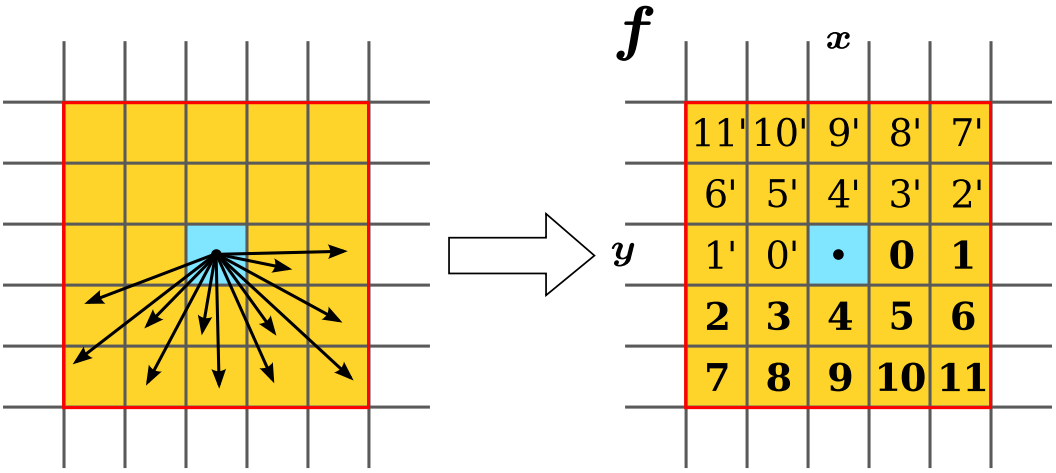


FIGURA A.16: By symmetry, we only need to compute 12 distances per pixel-vector.

SID AMEE

The stream algorithm for *AMEE* using *SID* distance consists of the following chained stages:

1. Setup and upload S.R.: We rearrange the data in streams as described and upload them into video memory.

2. Normalization: We compute the coefficients p_k and q_k as described by equation A.7. We reduce in the spectral dimension and divide the pixel-vector components by the value obtained. This operation needs three *kernels*:

   a) Accumulate: Vector addition and accumulation of streams.

   b) Reduce: Add the four components in a stream into one.

   c) Divide: Divide each component in the streams by the reduced value.

3. *SID* distance: We compute the 12 *SID* distances shown in figure A.16 for every pixel-vector. Thanks to the ability of referencing displaced

streams provided by our *SDK*, two *kernels* are enough for implementing this stage:

   a) SID^*: Compute the equation A.9 partially taking advantage of the associativity in l.

   b) Reduce: Again, we have to reduce the four components in the stream to one, completing the computation of the *SID* distance.

4. Cumulative distance: Again, we can take advantage of the possibility of referencing displaced streams to compute all distances with a single *kernel*.

5. Min/Max: We find the relative position to the minimum and maximum element inside the structuring element. We use the four components in a stream to hold the four relative coordinates, minimum and maximum (x, y).

6. MEI_3: We select the distance between the minimum and maximum.

7. Dilation: If we need to do more passes, we will select the pixel-vector with the greatest cumulative distance in the structuring element. Performing the dilation on the normalize image allows us to save redundant computations.

Figure A.17 shows the stream processing model scheme of the described algorithm.

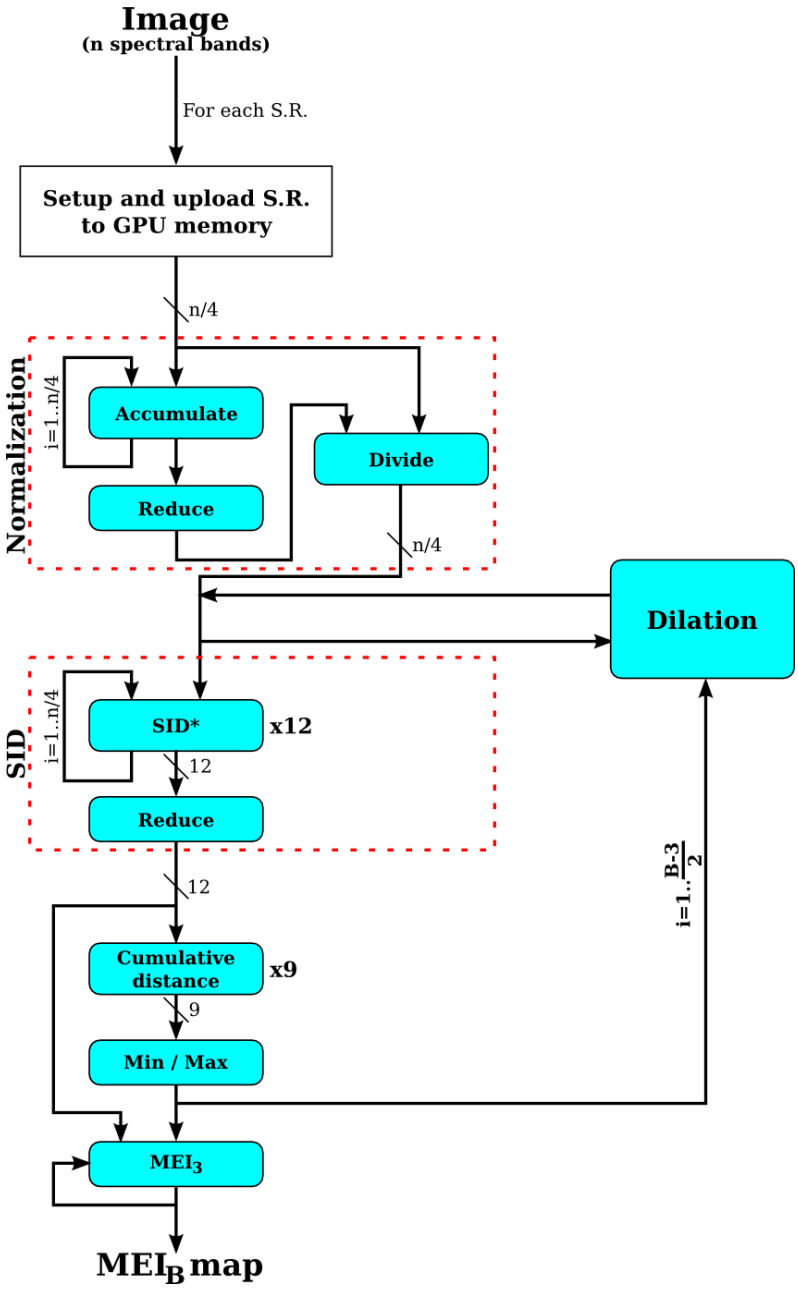


FIGURA A.17: Graphics representation of the stream model for *SID AMEE*. Computing the approximated MEI_B of an image in $\mathbb{K}^n$ using multi-pass *AMEE*

.

SAM AMEE

Following the same strategy, these are the stages of a stream processing version of the *SAM AMEE* algorithm:

1. Setup and upload S.R.: As in the previous case, we rearrange the image data and upload the streams to *GPU* memory.

2. Inner product: We compute the thirteen inner products needed to compute the distances. We use the two following *kernels*:

   a) Multiply and accumulate: Compute the partial inner products using displaced streams. In the end, we get four partial inner products.

   b) Reduce: We add the four partial inner products to obtain the final product.

3. Norm: We take the square root of the stream containing the products of each pixel-vector with itself.

4. *SAM* distance: From the inner products and norms we can easily obtain the twelve different distances with one *kernel*.

5. Cumulative distance: We compute the nine cumulative distances from the twelve computed before.

6. Min/Max: We find the relative position of the pixel-vectors with minimum and maximum cumulative distance in the structuring element.

7. MEI_3: Again, we select the distance between the maximum and minimum we found in the previous stage.

8. Dilation: If we have to do more pases, we select the pixel-vector with maximum cumulative distance.

Notice that in *SAM AMEE* from the inner product on we use scalar operations, which means that we cannot take advantage of the vector units as much. Figure A.18 shows the stream processing model for *SAM AMEE*.

Experimental results

In our evaluation the *CPU* versions of the algorithms follow a stream processing model too because they are also faster on *CPU*. Despite this behavior surprised us at first, it is not so unexpected and has been observed in other studies as well [GR05]. Essentially, beside removing redundant computations, the structure of these versions is more friendly to memory optimization techniques such as *tiling*.

As for the experimental platforms, we have used two different generations of *GPUs* and their respective contemporary generations of *CPUs*. Table A.3 shows the features of the *CPUs* used in the experiments, and table A.4 does the same for the *GPUs*.

Tabla A.3: *CPU* features

	Pentium 4 (Northwood C)	**Prescott (6x2)**
Year	2003	2005
FSB	800 MHz, 6.4 GB/s	800 MHz, 6.4 GB/s
ICache L1	12KiB	12KiB
DCache L1	8KiB	16KiB
L2 Cache	512KiB	2MiB
Memory	1GiB	2 GiB
Clock	2.8 GHz	3.4 GHz

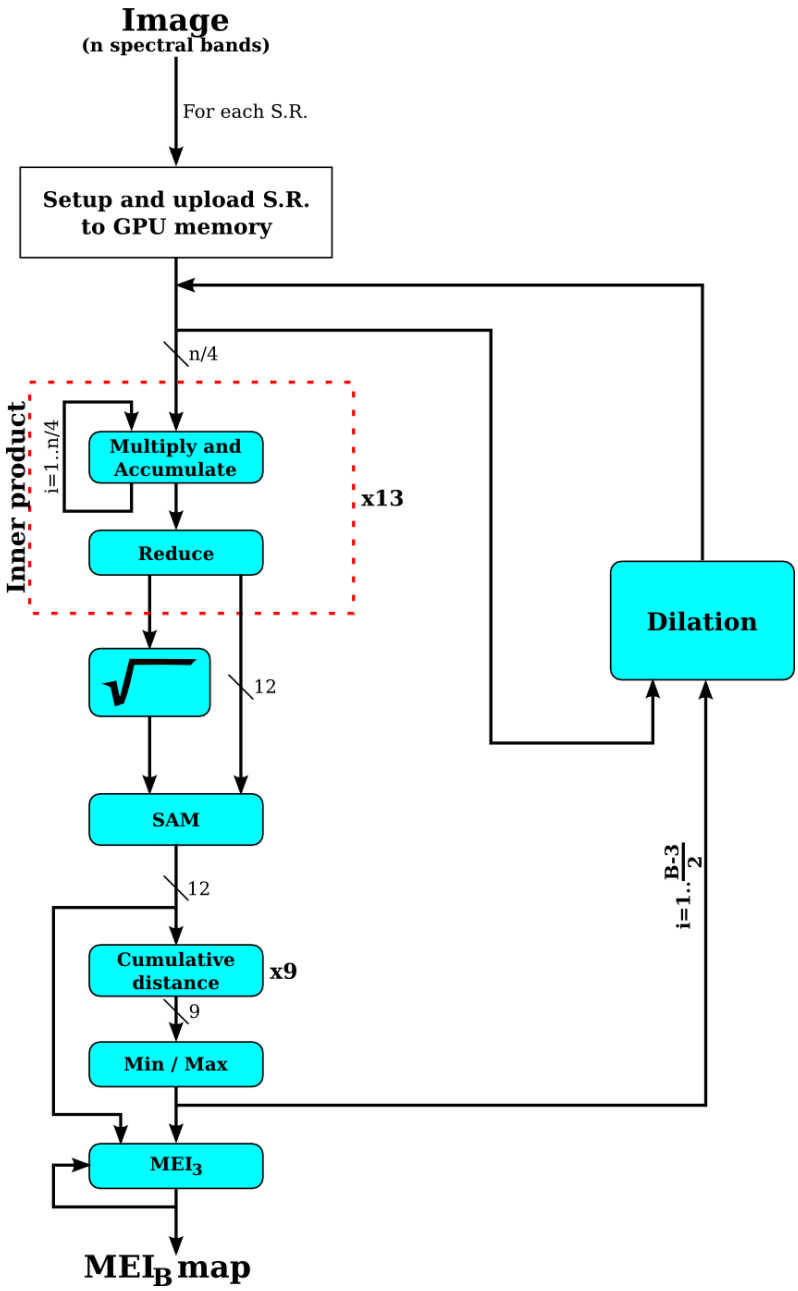


FIGURA A.18: Graphics representation of the stream model for *SAM AMEE*. Computing the approximated MEI_B of an image in $\mathbb{K}^n$ using multi-pass *AMEE*.

We chose two *Intel CPUs*, a *Pentium IV Northwood* (P4N) released in 2003, and a *Pentium IV Prescott* (P4P) released in 2005. Similarly, we chose two *NVIDIA GPUs*, a *GeForce FX 5950 Ultra* [BK03] (GPU *NV38*) as a 2003 representative, and a *Geforce 7800 GTX* [Cor05] (GPU *G70*) as a 2005 representative.

TABLA A.4: *GPU* features

	NV38	**G70**
Year	2003	2005
Core	NV38	G70
Bus	AGPx8	PCI Express
Memory	256MiB	256MiB
Core clock	475 MHz	430 MHz
Memory clock	950 MHz	1.2 GHz GDDR3
Memory bus	256-bit	256-bit
Memory bandwidth	30.4 GB/s	38.4 GB/s
Processor count	4	24
Texture fill rate	3800 MTexels/s	10320 MTexels/s

As input data, we use a common reference hyperspectral photograph taken with an *AVIRIS* sensor over a mining sector in Nevada (USA), dedicated to open air cuprite extraction. Hyperspectral data and ground-truth information of this terrain are available at *AVIRIS* website [NASA].

In order to test the scalability of our algorithm, we cut different sections of the image to obtain 16, 32, 64, 128, 256 and 512 *MiB* images. Notice that source data uses two byte integer coefficients but we cast and upload single precision floating point values, which require twice as much space in *GPU* memory.

Tables A.5 and A.6 show execution times obtained for 5-pass *AMEE* on *CPUs* and *GPUs* respectively. The *GPU* times include all the data prepro-

cessing and uploading time, as well as the transference from *GPU* to main memory of the obtained *MEI* map.

The *CPU* version has been compiled with automatic optimization flags using the *Intel ICC* compiler (*icc -O3 -ttp7 -restrict -xP*).

TABLA A.5: Execution time (in miliseconds) for *CPU* implementations on a Pentium IV Northwood (P4N) and a Pentium IV Prescott (P4P).

	SAM AMEE		**SID AMEE**	
Size (MiB)	P4N	P4P	P4N	P4P
16	6588.76	4133.57	22369.8	16667.2
32	13200.3	8259.66	45928	33826.7
64	26405.6	16526.7	92566.6	68185
128	52991.8	33274.9	187760	137412
256	106287	66733.7	377530	277331
512	212738	133436	756982	557923

TABLA A.6: Execution time (in miliseconds) for *GPU* implementations.

	SAM AMEE		**SID AMEE**	
Size (MiB)	NV38	G70	NV38	G70
16	898.36	457.37	1923.63	513
32	1817.52	905.93	3909.91	1034.42
64	3714.86	1781.58	7873.9	2035.01
128	7364.12	3573.3	15963.1	4144.82
256	14877.2	7311.05	31854.5	8299.07
512	29794.8	14616	63983.9	16692.2

We can see that *GPU* versions are significantly faster that *CPU* counterparts, specially for *SID AMEE*, due to its greater arithmetic complexity and a more extensive use of vector units. We also see that our algorithm scales perfectly with the image size, if we double the image size computing time doubles too.

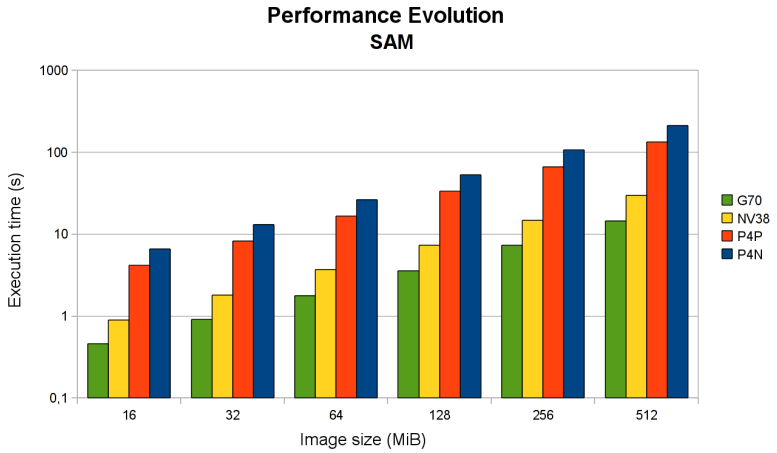


FIGURA A.19: Performance of *SAM AMEE* on *CPU* and *GPU* for different image sizes and five passes ($I_{max} = 5$).

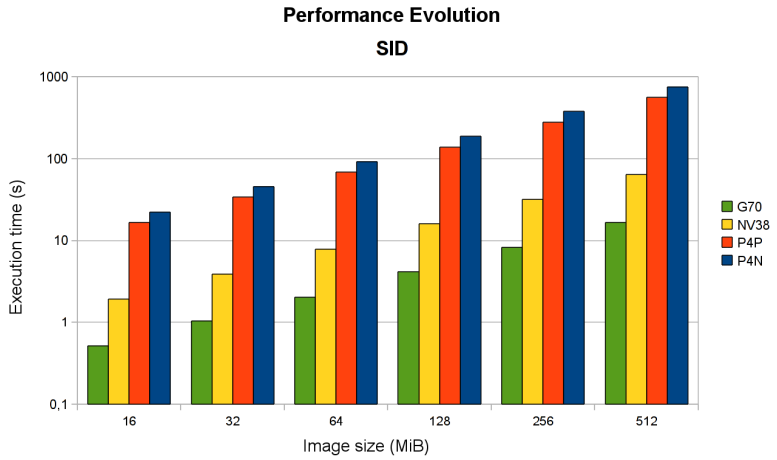


FIGURA A.20: Performance of *SID AMEE* on *CPU* and *GPU* for different image sizes and five pases ($I_{max} = 5$).

From the diagrams (in logarithmic scale) in figures A.19 and A.20 we observe those results and the evolution with the image size. Processing a $512MiB$ hyperspectral image on a *CPU* takes in the order of 10 minutes while the same image can be processed on a *CPU* takes around 15 seconds.

Figure A.21 shows the speedup obtained on *GPU* respect to *CPU* using

an image of $512MiB$ and different passes. We observe that the more passes we do the higher the speedup. As we do not need to transmit data from *CPU* to *GPU* between passes, the initial transfer time is compensated by a higher number of computations. The *SAM AMEE* version reaches a speedup of up to 10, while the *SID AMEE* version goes up to 35 and 40 times faster than the equivalent *CPU* version.

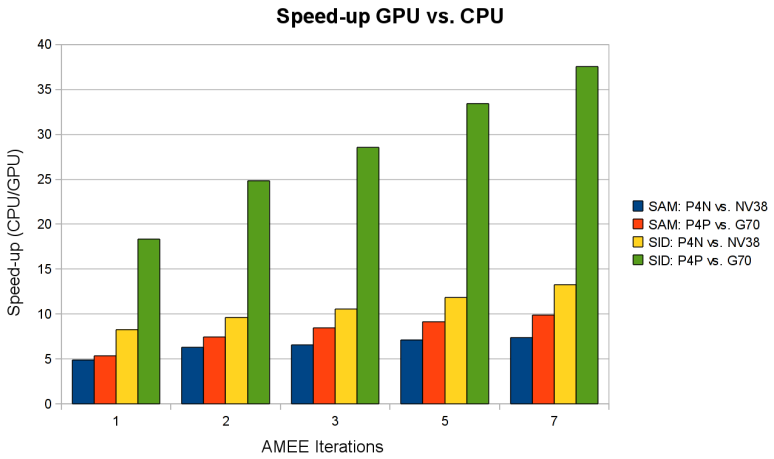


FIGURA A.21: Speedup of *GPU* implementations compared to *CPU* implementations.

Regarding the comparison between both versions of *AMEE*, figure A.22 shows the speedup of *SAM AMEE* compared to *SID AMEE*. We see that the speedup is consistent independently of the image size. Also, although the difference between versions is considerable on *CPU*, being *SAM AMEE* up to four times faster than *SID AMEE*; *GPU* versions, though noticeable, do not show such a steep difference.

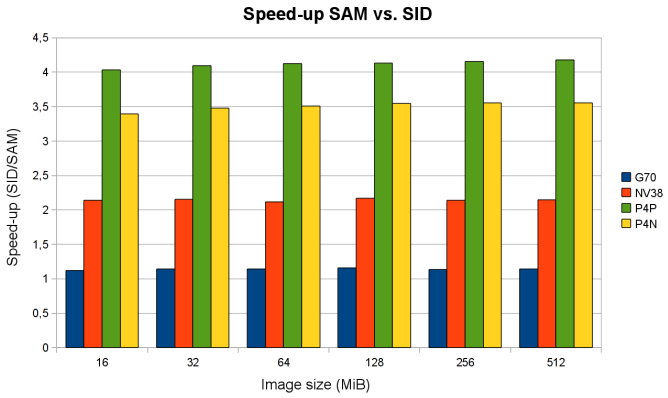


FIGURA A.22: Speedup of *SAM AMEE* compared to *SID AMEE* on all the experimental platforms ($I_{max} = 5$).

In oder to see how architectures have evolved for both, *CPU* and *GPU*, figure A.23 compares the speedups of the 2003 generation with the 2005 generation. We observe that performance on *CPU* have increased around 30 %-50 %, while *GPUs* have practically doubled their performance in the worst case, and quadrupled it in the case of *SID AMEE*.

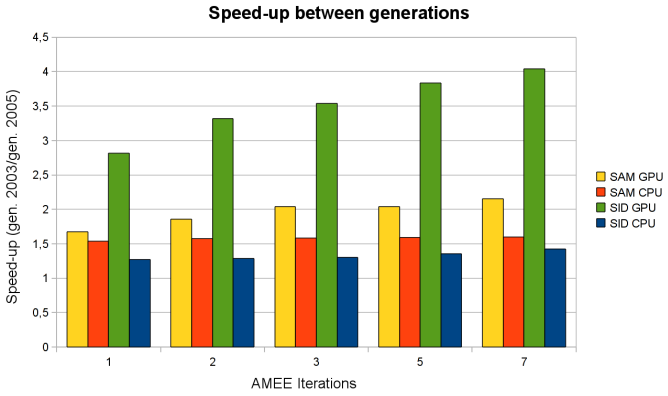


FIGURA A.23: Speedup comparison between two *CPU* generations (2003's *Pentium IV Northwood* and 2005's *Pentium IV Prescott*) and two *GPU* generations (2003's *NV GeForce 5950 Ultra* and 2005's *NV GeForce 7800 GTX* del 2005).

Conclusions

Our experiments prove that we can make the most of *GPUs* for hyperspectral images. We observed a significant improvement in the execution of *AMEE* on *GPUs* compared to contemporary *CPUs*. From the obtained data we can draw the following conclusions:

- We can dramatically reduce the time needed for hyperspectral imaging using a *GPU*, going from several minutes to a few seconds.
- Our algorithm's performance increases much more between *GPU* generations than *CPU* generations, proving that it scales well in manycore architectures.
- The improvement in *GPU* respect to *CPU* is more significant for *SID AMEE* than *SAM AMEE*. This is due to *SID* having a greater arithmetic density and making extensive use of transcendental operations which *GPUs* implement in hardware while *CPUs* use complex software functions.
- Finally, although *SAM AMEE* is a little bit less accurate than *SID AMEE* (see section A.2.4), it is considerably faster.

As a general conclusion, our *GPGPU SDK* has proven to be a valid solution to make the most of *GPUs* while hiding the complexities of *OpenGL* behind the stream processing model. We also proved that graphics processing units are an ideal platform for hyperspectral image processing, opening the possibility of real time processing.

A.4. Hyperspectral Imaging on *CUDA*

This section is dedicated to analysing hyperspectral imaging on *NVIDIA's CUDA*, a development environment targeted to massively parallel architectures. For the sake of brevity we leave out all the details about the development environment, which can be found inside *CUDA* programming manuals [nvi12]. From now on, we assume the reader is well versed in *CUDA*.

This section is divided in two well differentiated parts. First, in Section A.4.1 we perform different analysis to try and determine which *CUDA* architectural elements influence performance the most. The second part, in Section A.4.2, we get into details of porting *AMEE* to *CUDA*.

A.4.1. Performance analysis

CUDA allows programmers a greater control over how the *GPU* behave. In order to make the increased complexity manageable, we start by performing a few test to find out where to focus our optimization efforts and how. There are different elements we can control as programmers:

- Memory hierarchy exploitation. *CUDA* gives us a direct control over global and shared memory, and we can choose to access the global memory through texture units and take advantage of their *cache* memory. It is in the hands of the programmer to make good use of the memory hierarchy.

- Memory access pattern. A *GPU* runs thousands of threads in parallel and accessing the memory concurrently, which poses a great pressure on

the system bus. To relieve that pressure, *GPUs* incorporate advanced features in their memory controllers. Many of those features depend on the memory access pattern.

- *Warp* divergence and concurrency. Conditional statements can become a problem, given that they might provoke divergence and thread serialization. This imply a loss in parallelism and therefore a significant performance degradation.

- Occupancy. Our threads are distributed among all the available multiprocessors. Depending on the resources required per each thread and the number of threads per block we pack, the number of threads run in the same multiprocessor might vary, thus varying the effective parallelism.

In oder to delimit our performance study and generalize it within reason, we will focus on the elements that are easier to control.

As starting point we have used a simplified version of *AMEE*. Because of its properties (compute density and parallelism) it is an ideal case of study and it makes our conclusions easier to generalize for our target: *AMEE*.

This simplification, shown in figure A.24, essentially consists of reducing the spacial dimensionality of *AMEE*.

```
for ij=1,n {
    for b1=1,m {
        acc[b1]=0
        for b2=1,l {
            dp=0
            for li=1,S {
                dp+=f1(I,ij,b1,b2,li)
            }
            acc[b1]+=f2(dp)
        }
    }
    max=-Inf
    for b3=1,m {
        if acc[b3]>max {
            max=acc[b3]
            maxpos=b3
        }
    }
    np=0
    for li=1,S {
        np=np+f3(I,ij,maxpos,li)
    }
    res[ij]=f4(np)
    respos[ij]=maxpos
}
```

FIGURA A.24: Simplified algorithm. I is the input image; f_1, f_2, f_3 and f_4 are functions that depend on their input parameters.

The two outer loops are merged into one (index ij) and, likewise, the loops that computes the cumulative distances inside a window go from two spacial dimensions to one, reducing four inner loops to two (b_1 and b_2).

Besides, we no longer search for the positions of the maximum and minimum cumulative distances inside a window, only the maximum. We replace the function that computes the *MEI* score by another that only depends on the position of the maximum.

We have replaced mathematical operations by functions to make it easier to read. In the code of figure A.24, f_1 computes the partial inner product between two positions (b_1 and b_2) around ij; similarly, f_2 computes the distance using

that product. The definition of f_1 and f_2 depends on the distance we are using (*SID* or *SAM*). Necessarily f_4 will do the same as f_2, and f_4 the same as f_2 only in this case the partial inner product is computed between ij and $maxpos$.

This simplified algorithm presents the same kind of parallelism in the same places as *AMEE*. This allow us to make better assumptions when generalizing our conclusions to *AMEE*.

In order to map it to *CUDA*, we parallelize the outermost loop because it is the one with the greatest degree of parallelism (most number of independent iterations). As experimental platform we have used the *NVIDIA GeForce 8800 GTX* with a *G80* core which supports *CUDA 1.0*.

Exploiting occupancy

In order to study the influence of processor occupancy we have tried two common code transformations: loop splitting and loop unrolling.

In loop splitting, the loop is divided in several loops which go through the same range of indices but only do a part of the work each. We can split the outermost loop (ij) to obtain several loops with a lower work load, reducing the need for resources of each one and potentially increasing the maximum occupancy. In return, we have to launch a higher number of *kernels* which imply a certain extra load. Moreover, loop splitting might lead to variable expansion and a greater memory usage. Also, we might lose some locality accessing to memory.

Loop unrolling also makes an impact in resource occupancy. What we do is replicating the body in a loop to build a sequence of statements that reduces the number of iterations. This way, pressure on the register file in-

creases, reducing the maximum number of concurrent threads. According to a study [RRB+08], loop unrolling (in the context of matrix-matrix multiplication) is a worthwhile technique in spite of it all, due to the increase in instruction level parallelism. Nevertheless, sometimes the increase in instruction level parallelism cannot compensate the decrease in occupancy.

A last way to control the occupancy is limiting the number of registers used by *kernels*. *CUDA* compiler allows us to restrict the maximum number of registers available to a *kernel*, giving us another option to trade-off occupancy, this time at the expense of increasing register spilling and so a loss in performance.

First of all, in order to study the influence of processor occupancy we have run two different versions of our algorithm using a different number of threads per block (from 16 to 384). One version, *simple*, containing the whole algorithm in one *kernel*; and another version, *split*, where the outermost loop (ij) is split into two parts as shown in figure A.25. This loop splitting implies the expansion of the variable *acc*, local to loop ij, due to a *read after write* dependency crossing the split point. In both cases, we implement the algorithm in *CUDA* by mapping each iteration of ij loop to a different thread.

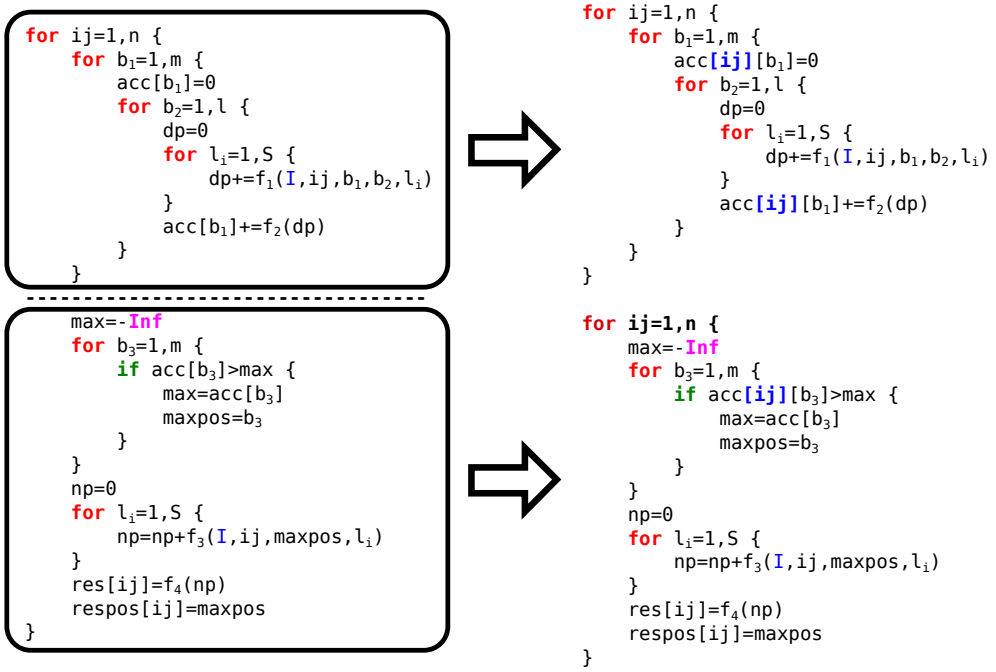
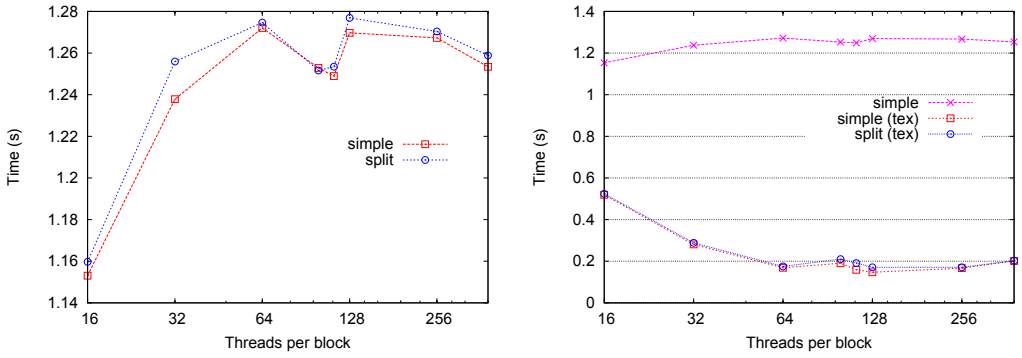


FIGURA A.25: Loop splitting. We divide the algorithm from figure A.24 in two, expanding the variable *acc* to solve the dependency inside the original loop.

When we run the first experiments we find that, even though the differences are small (less than 10 %), the best results are obtained for 16 and 32 threads per block, which implies an occupancy as low as 33 %. Apparently, another factor is limiting performance much more than occupancy.

As shown in figure A.26(b) our intuition was right. In a second set of experiments, we modify the algorithms to access the main memory through the texture *cache* to lower the pressure on the memory bus, taking advantage of locality and reducing problems with uncoalesced accesses. Once we remove the bottleneck in the memory access, we achieve optimal performance with 128 threads per block, which is expected because that block size allows an occupancy of 83 % which is the maximum that is possible for our code.

Another interesting result we can observe is that the split version nev-



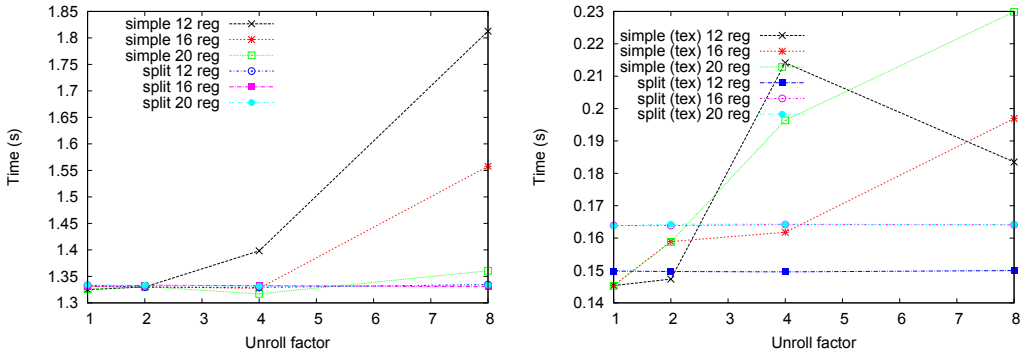
(a) (b)

FIGURA A.26: Execution time depending on the number of threads per block. We compare a single *kernel* algorithm with a two *kernel* version in which the main loop's been split into two. (a) Accessing directly to main memory. (b) Accessing through texture *cache*.

er performs better than the single *kernel* version (*simple*). The difference is less pronounced when we use texture *cache* to alleviate our memory access issues, but the overload of launching two *kernels* instead of one still exceed any advantage we could get from a lower resources requirement.

Regarding loop unrolling, we prepared several experiments unrolling the innermost loop (l_i) in our algorithm (see figure A.24). In order to exert some degree of control on the occupancy due to the additional resources requirement, we use facilities provided by the *CUDA* compiler that allow us to put a limit to the maximum number of registers used by a *kernel*. We run tests with 12, 16 and 20 registers, forcing a maximum occupancy of 83 %, 67 % and 50 % respectively, being 20 the number of registers the compiler assigns by default to our specific *kernel*.

Figure A.27(a) shows the results when we access to main memory directly and figure A.27(b) shows the results when using the texture *cache* to alleviate memory access issues. As we can see, loop unrolling does not lead to better



(a) (b)

FIGURA A.27: Loop unrolling impact on performance. We force a compilation with 12, 16 and 20 register to set the occupancy at 83 %, 67 % and 50 % respectively. All test are launched with 128 threads per block.

performance in any case. We do appreciate that limiting the number of registers (increasing maximum occupancy) provides a slight improvement in the *split* version.

Exploiting the memory hierarchy

Given that the performance of our *split* algorithm and the single *kernel* version (*simple*) is quite similar, we describe our experiments with the memory hierarchy using the *simple* version, which is slightly faster than *split*.

When trying to make good use of the memory hierarchy, the first thing to look at is where the algorithm access the input data. In our case, we access the input image I from f_1 and f_3. The first function is placed inside two nested loops, b_1 and b_2, which imply that there will be some reuse around the outermost loop index (ij) throughout the loop l_i (see figure A.24). Those two loops are not present in case of f_3, so the access to the input image I from there will not be as redundant as in f_1 case. Nonetheless, some reuse by

different threads (index ij) is to be expected.

Because of that difference in memory access from those two locations, we have considered five different strategies:

- [1] (simple tex). As a base line, we see how the algorithm behaves when all the accesses are made through texture *cache*.

- [2] (simple sh). In this implementation, data is preloaded to shared memory. This implies a pre-load phase, in which every thread in a block fetches a specific block of data, and then a synchronization barrier guarantee that all data is already in shared memory before we begin to use it.

- [3] (simple tex+sh). In this version, we try to alleviate the uncoalesced memory accesses by making use of the texture *cache*. We capture locality in the shared memory, faster than *cache* but with the overload of a synchronization barrier, but take advantage of the way the texture *cache* fetches whole *cache* lines avoiding uncoalesced accesses to main memory.

- [4] (L1 sh L2 tex). Given that f_3 reuses less data than f_1, it is interesting to explore an asymmetry in how we fetch the data. In this case, we try to capture locality from f_1 using shared memory but we use the texture *cache* for f_3, removing one synchronization barrier.

- [5] (L1 tex+sh L2 tex). Lastly, trying to take advantage of synergy between texture *cache* and shared memory, we pre-load the data into shared memory through texture *cache* with the intent on avoid uncoalesced accesses, while exploiting data reuse through the faster shared memory.

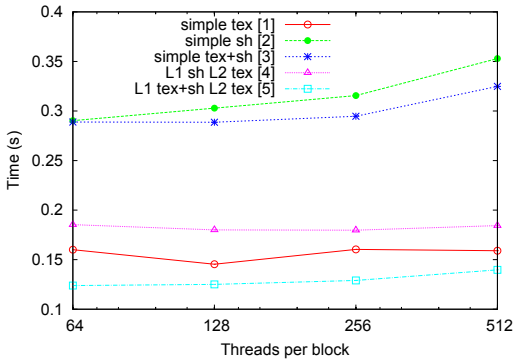


FIGURA A.28: Effects of using the different levels of the memory hierarchy. Depending on the method used to fetch image data, the results are: [1] Access through texture *cache*; [2] Pre-load in shared memory; [3] Pre-load in shared memory through texture *cache*; [4] Pre-load in shared memory in the first loop, fetch data through texture *cache* in the second one; [5] Pre-load in shared memory through texture *cache* in the first loop, fetch data through texture *cache* in the second one.

Figure A.28 shows the execution time of the different implementations. We can see how the faster version is [5], which is similar to [1] but uses shared memory to capture data reuse, instead of relying on the texture *cache*. We can also see, by the difference between [3] and [5], that removing the second synchronization barrier provides better results even though data reuse is slower. This is because, as we explained before, we do not reuse data in f_3 as much as we do in f_1. We can also appreciate the synergy between texture *cache* and shared memory from the differences between [4] and [5].

From our experiments, we can draw the following conclusions:

- Making good use of the bandwidth between *GPU*'s main memory and processing cores is the ultimate performance factor in our *CUDA* applications.
- We can use the texture *cache* to alleviate the issues of uncoalesced memory accesses.

- Shared memory is not always the best option. If we cannot make sure there's enough data reuse, the overload caused by the necessary synchronization barriers and bank conflicts will be higher than the benefit obtained from a faster memory.

It is interesting to notice the synergy obtained from the combined use of texture *cache* and shared memory. The former is in charge of alleviating the setback of uncoalesced memory accesses while the second, being faster, provides a better option to take advantage of data reuse.

Once we have established a series of design principles, we can explore the implementation of *AMEE* on *CUDA*.

A.4.2. *AMEE* on *CUDA*

In this section we have explored the different possibilities that arise when implementing *AMEE* on *CUDA*. From previous sections we already know that *GPUs* run *AMEE* faster than *CPUs*, so we have focused on finding out what is the best implementation strategy given the new possibilities that *CUDA* provides.

We have implemented and tested five different versions that are discussed in detail in further sections:

- $V1$: As a base line we have implemented a version analogous to the one we used on *OpenGL*. It comprises the exact same number of *kernels*, each one in charge of analogous tasks. This version only takes advantage of the shared memory where it makes sense (as we learnt from Section A.4.1).

- $V2$: Given that we do not have the *kernel* size nor stream count limits imposed by *OpenGL* anymore, we have tried to take advantage of merging *kernels*. As shown above, reducing the number of *kernels* launched might be a good idea.

- $V3$: In this version, we use memory *padding* so that the image in memory is aligned by rows. We have used this version as a reference for the next two, given that memory *padding* is mandatory for the next two techniques and we want to separate the benefits of *padding* from the benefits of the strategy.

- $V4$: We try to capture the $2D$ locality present in our algorithm replacing one-dimensional textures by two-dimensional textures.

- $V5$: With the same intention than $V4$, this time we have tried to take advantage of the $3D$ locality using three-dimensional textures.

All the experiments has been run on an *NVIDIA GeForce 8800 GTX* with a *G80* core. This is the first generation of *GPUs* with *CUDA* capability. The technical specifications can be found in the *CUDA* development manual ([nvi12]).

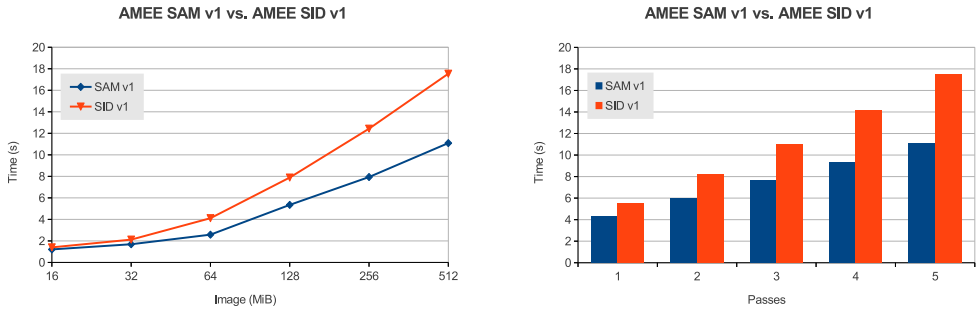
From *OpenGL* to *CUDA*

In this version we have ported the algorithmic structure we used in *OpenGL* directly to *CUDA*. This is our base line, from here on we have tried different strategies to take advantage of the benefits that *CUDA* provides. This version of the algorithm has the following features:

- We use shared memory pre-loading data through texture *cache*.

- One *kernel* per distance to compute.
- One *kernel* per inner product.
- One *kernel* per cumulative distance.
- One *kernel* to find minimum and maximum cumulative distances.
- One *kernel* to compute the *MEI* score.
- One *kernel* to make the morphological dilation.

Figure A.29 shows the results of this version. We can see that the time for 16, 32 and 64 MiB is almost the same, even though the image size is doubling. This indicates that setup times for our version of *CUDA* environment are significantly greater than data transfer and processing for small images. We should take into account that we used a "beta" *CUDA* driver, the very first to support *CUDA*, and setup time is expected to have been significantly improved in later versions. Looking at the performance with different number of passes we see a 20 % to 30 % improvement respect to *OpenGL*, which is to be expected due to the technological upgrade. Another difference we should notice regarding to *OpenGL*, is that we are using one-dimensional textures while *OpenGL* uses two-dimensional ones.



(a) (b)

FIGURA A.29: Results for a direct port from *OpenGL* to *CUDA*. (a) Depending on the image size and making 5 passes. (b) Depending on the number of passes over a 512 MiB image.

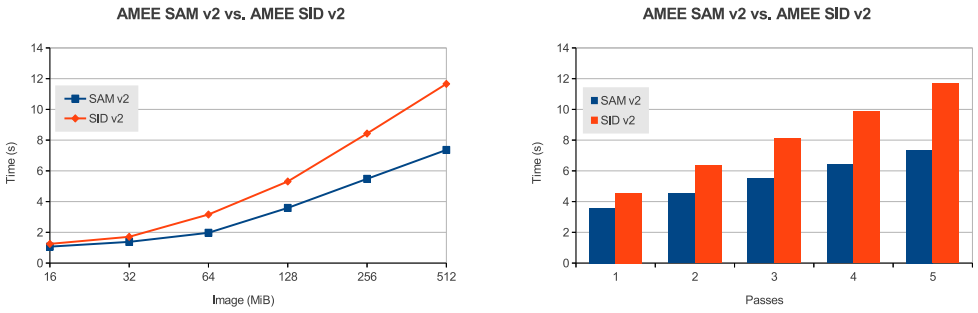
Minimizing the number of *kernels*

The first advantage of *CUDA* over *OpenGL* that we have exploited is the absence of a restriction on the number of inputs and outputs per *kernel*. We can dramatically reduce the number of *kernels*, which results in a more efficient use of *GPU* resources, as well as a reduction in the overload due to *CPU* tasks.

In this version, the several dozens of *kernels* of the previous version are reduced to four:

- One *kernel* to normalize the image for *SID* or compute the norm of the pixel-vectors for *SAM*.
- One single *kernel* to compute the twelve distances for each pixel-vector.
- One single *kernel* to compute the nine cumulative distances, find the maximum and minimum, and compute the *MEI* score.
- One *kernel* to make the morphological dilation.

Figure A.30 shows the results of this version. As we expected, given the results of the analysis in section A.4.1, we have significantly improved the performance in both, *SID AMEE* and *SAM AMEE*.



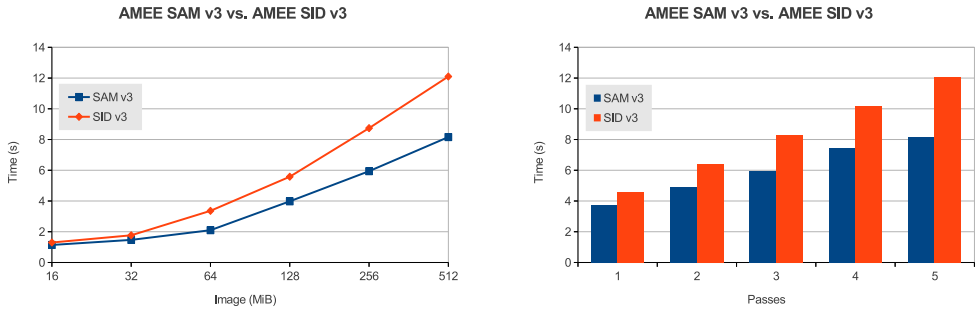
(a) (b)

FIGURA A.30: Results of minimizing the number of *kernels*. (a) Depending on the image size and making 5 passes. (b) Depending on the number of passes over a 512 MiB image.

Improving memory access

The next obvious improvement is making use of the texture units in *2D* mode as well as *3D* mode. This would require allocating *GPU* memory using special function that allocate padded memory, which aligns the image to optimize access in two and three dimensiones (see *cudaMallocPitchr* in *CUDA* development manual [nvi12]).

In order to be able to differentiate the improvement due to $2D$ and $3D$ texture units from the improvement due to a better memory alignment, this version still uses the one-dimensional cache but allocates better aligned memory.



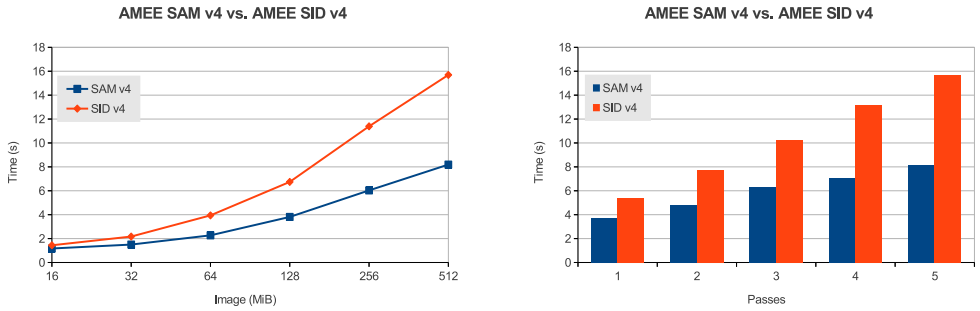
(a) (b)

FIGURA A.31: Results using padded memory. (a) Depending on the image size and making 5 passes. (b) Depending on the number of passes over a 512 MiB image.

In figure A.31 we can observe that, even though clearly superior to the first version, there is a performance loss compared to the previous version. This makes it clear that the use of padded memory by itself presents no advantage with one-dimensional textures, quite the opposite.

Using two-dimensional texture *cache*

In this version, we bind the memory allocated to texture units in *2D* mode. It implies almost no change with respect to the previous version, but it allows us to better capture the *2D* locality present in our algorithm.



(a) (b)

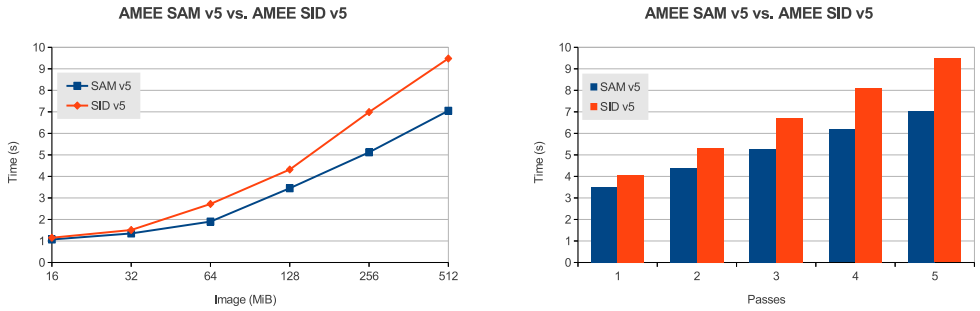
FIGURA A.32: Results using two-dimensional textures. (a) Depending on the image size and making 5 passes. (b) Depending on the number of passes over a 512 MiB image.

As we can see in figure A.32, we obtain a slight improvement without reaching the performance achieved in the second version. Even if our algorithm present some degree of *2D* locality, hyperspectral images are of three-dimensional nature.

Using three-dimensional textures

In the *CUDA* version of *AMEE*, as we no longer divide the image in spectral slices, we can easily see how we are actually working with three-dimensional data (two spacial dimensions and one spectral dimension). At this point we should ask how *AMEE* would perform if we exploit three-dimensional locality taking advantage of the facilities provided by *CUDA* in this regard.

We can see the results in figure A.33 and, as we expected, the results are clearly superior to any of the previous versions.

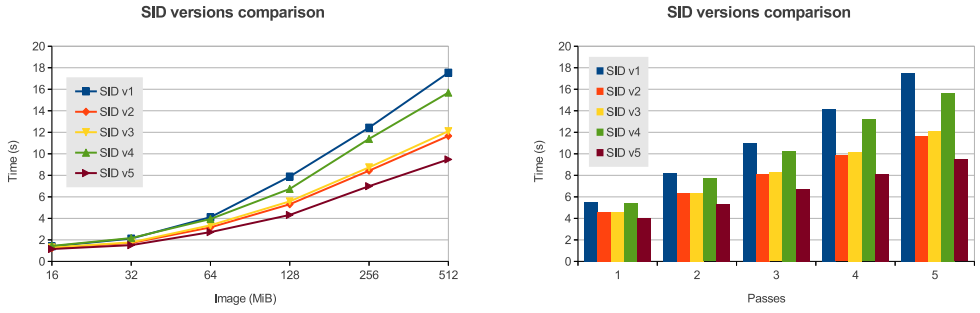


(a) (b)

FIGURA A.33: Results using three-dimensional textures. (a) Depending on the image size and making 5 passes. (b) Depending on the number of passes over a 512 MiB image.

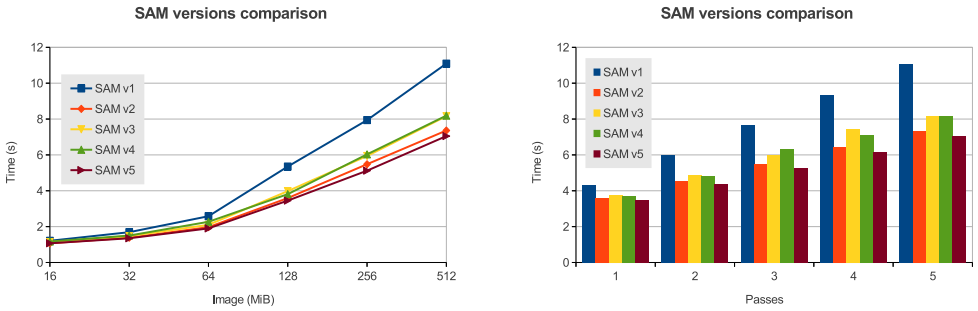
Comparing the different versions

Finally, we shall compare the *AMEE* implementations we have presented between them. Figure A.34 shows the results of *AMEE* when using *SID* as a distance function while figure A.35 shows the results of *AMEE* using *SAM* as distance function.



(a) (b)

FIGURA A.34: Comparison of the different versions of *SID AMEE*. (a) Depending on the image size and making 5 passes. (b) Depending on the number of passes over a 512 MiB image.



(a) (b)

FIGURA A.35: Comparison of the different versions of *SAM AMEE*. (a) Depending on the image size and making 5 passes. (b) Depending on the number of passes over a 512 MiB image.

As we can see, the implementation using *3D* textures is the best in all, while the direct translation from *OpenGL* to *CUDA* is well behind the rest.

A.4.3. Conclusions

In this section we have made an analysis of *CUDA* performance factors which we used to guide our implementation of *AMEE*, using both *SID* and *SAM* distances, on *GPUs* using *CUDA* as a development environment. From the results obtained we can draw the following main conclusions:

1. The good use of the different levels in the memory hierarchy is the ultimate factor when developing applications on *CUDA*.

2. We can get good results following a scheme similar to the one used when we are developing on *OpenGL*, but with relatively little effort we can get significant improvements using *CUDA*.

3. Using texture units to fetch data from main memory to shared memo-

ry can lead to performance improvements when the memory controller cannot coalesce the memory accesses.

4. Even though bigger *kernels* imply lower occupancy due to higher resources requirement, the overload of a higher number of smaller *kernels* does not compensate the improvements in occupancy.

In order to keep this work delimited, we have performed our study on the first architectonic family supporting *CUDA*. Nevertheless, during this time new *GPU* architectures arose which eliminate several limitations present in *CUDA 1.0*. They also add new capabilities which imply new opportunities to exploit parallelism. At the same time, manufactures began to provide *GPUs* targeted to different market segments and aimed to meet other necessities, such as power restrictions, which might be useful in some specific scenarios like Earth observation satellites. In these environments, delivering enough performance to make on-board hyperspectral imaging feasible will not be enough. Among other things, we also must comply with the satellite power restrictions, usually a couple hundred Watts shared among all satellite sub-systems.

New architectures

As a representative of these new architectures and their possibilities, in the area of power optimization we have at our disposal *Carma* [SEC13] boards designed to deliver high performance based on *GPUs* while optimizing the performance per Watt. Its objective is to reduce power consumption replacing the general purpose *GPU* by a low power processor.

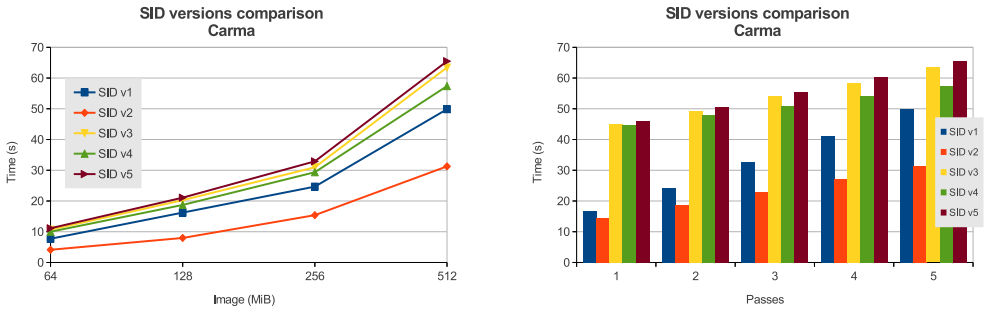
The idea behind *Carma* is, if the burden of high demanding tasks is dele-

gated to a *GPU*, which provides high performance per Watt, we can replace the *CPU* by a low power processor, even if its performance is relatively low. In the case of *Carma*, the main processor is a *Tegra 3*, an *MPSoC* that integrates an *ARM Cortex A9* quad-core *CPU* and a *ULP GeForce* (*Ultra Low Power*) *GPU*. Also, as a coprocessor aimed at intensive computation, *Carma* includes a *NVIDIA Quadro 1000M* (*GF108GLM* core) *GPU* with *CUDA 2.1* support.

This *GPU*, which includes 96 scalar cores, delivers 270 GFlops with a power consumption of 45 Watts only. In turn, a *G80 GPU* delivers 518 GFlops with a power consumption of 185 Watts. This means that even if *Carma* does not match the raw power of a *G80*, with a performance per Watt of $6GFlops/Watt$ versus $2{,}8GFlops/Watt$ is significantly more power-efficient. If our objective is to complete a task as fast as possible *Carma* is not the best solution, but if we have power restrictions (as is the case in satellite boarded solutions) or power efficiency is a main concern this kind of board is a solution that we must take into account.

As a reference, we have run our five version of *AMEE* on *CUDA* using a *Carma* board, and we can see the obtained results in figures A.36 and A.37.

We notice how the versions performing memory rearrangement (v3, v4 and v5) become significantly slower. If we look at the difference in time between passes, we see that the differences among versions is not really that significant. The differences between 1 and 5 passes are in the range of 7-8 seconds. *CPU* tasks (transferring data from main memory to *GPU* memory and related tasks) are taking most of the time. In versions with memory rearrangement this task lead to a significant overload. The rearrangement cost on general purpose systems was compensated by more efficient memory access. On *Carma*, which

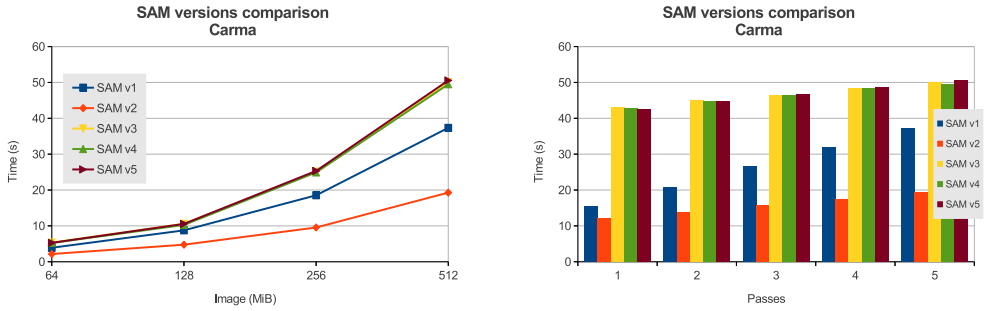


(a) (b)

FIGURA A.36: Results for the five different *SID AMEE* implementations on *Carma*. (a) Performing 5 passes on images with different sizes; (b) Using a 512MiB image and performing a different number of passes.

uses a low power and, consequently, low performance *CPU*, the arrangement overload is much more significant. Also, as *Carma*'s *GPU* architecture has more relaxed conditions for memory access coalescing, we are not taking as much advantage of the *cache*. In conclusion, we find a much greater overload and a smaller advantage when using $2D$ and $3D$ caches, and so we obtain quite different relative results in comparison to those obtained on a *G80 GPU*.

As a general conclusion, it becomes self-evident that specific results for each implementation will depend, to a great extent, on the *GPU* architecture and the specific platform mounting them. If we want an optimal solution, taking into account the rate at which *GPU* architectures change and new systems incorporating them appear, we will need a specific study for each platform. In order to make the problem approachable, a possible solution is designing a library that provides auto-tuned primitives in the same way *ATLAS* (*Automatically Tuned Linear Algebra Software*) provides linear algebra primitives in which the inner parameters are automatically determined for



(a) (b)

FIGURA A.37: Results for the five different *SAM AMEE* implementations on *Carma*. (a) Performing 5 passes on images with different sizes; (b) Using a 512MiB image and performing a different number of passes.

every specific platform.

A.5. Main Contributions

The main objective of this work has been studying hyperspectral imaging on graphics processing units, specifically *end-member* extraction using the *AMEE* algorithm, an unsupervised *end-member* finding algorithm that takes advantage of the correlation between spacial and spectral information relying on extended morphological operations.

We divided our study in two parts attending to the two great architectonic families: the one present at the beginning of this work, when there were no frameworks to use *GPUs* as general purpose processors; and the one release during the development of this work, shipped together with development frameworks provided by manufactures and directed to ease the use of *GPUs* as high-performance computing systems.

In the first part, given that the only way to take advantage of a *GPU*'s processing power was using a graphics synthesis library –*OpenGL* in our case– we began by designing and implementing a framework to abstract the complexity of graphics synthesis exposing a stream processing model. We have transformed two versions of *AMEE* –one based on *SID* distance and the other based on *SAM* distance– to the stream processing model, extracting all the implicit parallelism and removing the redundant computations present in the naive version of the algorithm, in oder to implement them on *GPU* taking advantage of our framework.

In the second part, thanks to the new architectures that emerged, we moved our study to the *CUDA* environment, with much less limitations than our framework and, at the same time, a lot more challenges to face when exploiting

performance. With that in mind, we began by making a performance analysis trying to pinpoint the most influential parameters in our case of study, locate potential problems and evaluate possible solutions. We used the conclusions drawn from this first analysis to guide the implementations of the two *AMEE* versions on *CUDA*, which led to the analysis of five different approaches.

This work produced the following publications:

[STP+06] J. Setoain, C. Tenllado, M. Prieto, D. Valencia, A. Plaza, and J. Plaza. Parallel Hyperspectral Image Processing on Commodity Graphics Hardware. In *ICPPW '06: Proceedings of the 2006 International Conference Workshops on Parallel Processing*, pages 465–472, Washington, DC, USA, 2006. IEEE Computer Society

[SPT+07] J. Setoain, M. Prieto, C. Tenllado, A. Plaza, and F. Tirado. Parallel Morphological Endmember Extraction Using GPUs. *IEEE Geoscience and Remote Sensing Letters*, pages 441–445, 2007

[SPTP08] J. Setoain, M. Prieto, C. Tenllado, and A. Plaza. GPUs for Parallel on-board Hyperspectral Image Processing. *International Journal of High Performance Computing Applications*, pages 424–437, 2008

[TSPT08] C. Tenllado, J. Setoain, M. Prieto, and F. Tirado. 2D-DWT on GPUs: Filter-Bank versus Lifting. *IEEE Transaction on Parallel and Distributed Systems*, pages 299–309, 2008

[SPTT08] J. Setoain, M. Prieto, C. Tenllado, and F. Tirado. Real-Time On-Board Hyperspectral Image Processing Using GPUs. In *High Performance Computing in Remote Sensing*, chapter 18, pages 411–451. Chapman And Hall, 2008

[STG+08] J. Setoain, C. Tenllado, J. I. Gomez, M. Arenaz, M. Prieto, and J. Tourino. Towards Automatic Code Generation for GPU Architectures. In *PARA'08: Workshop on State-of-the-Art in Scientific and Parallel Computing*, 2008

As a general conclusion, we find graphics processing units platforms of great interests to processing massive quantities of data, as well as to our case: hyperspectral imaging. With a performance per Watt significantly superior to that of general purpose processors but a market volume that allows for similar prices, they arise as a solution to necessarily take into account nowadays as well as in the future.

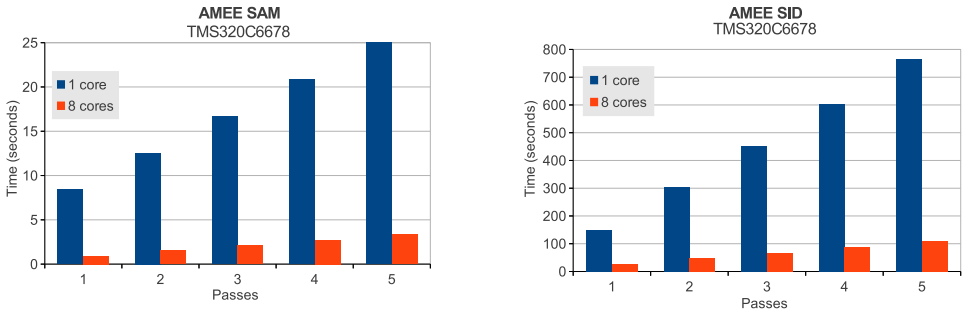
A.6. Future work

Throughout the development of our study it has become evident the fast changing rate of *GPU* architectures as well as development environments aimed at exploiting *GPUs* as general purpose high performance platforms. Given that with every architectural change and every new release the possibilities increase and results may vary, generic studies for specific applications become unfeasible.

As of today, the compiler technology is far from being effective enough to rely on them and so a much more interesting possibility would be the development of auto-tuned primitive libraries, as we already suggested at the end of Section A.4.

Although the *CUDA* environment presents an *API* that changes with every new architecture, and with it the performance parameters, we have also got *OpenCL* [Gro08] available, which presents a much more stable programming interface. This eases the possibility of using an approach similar to that of *ALTAS*, so we can offer a set of computing primitives that tune themselves for specific systems, finding the optimal operating parameters automatically.

Specifically in the case of hyperspectral imaging –given the restrictions on power consumption and tolerance to the environment imposed upon satellite hardware– the recently appeared multi-core *DSPs* such as *TMS320C6678* [TI12] offer a variety of opportunities for satellite on-board hyperspectral imaging. We have implemented, as a test, a preliminary version of *AMEE* on this platform.



(a) (b)

FIGURA A.38: *AMEE* on Texas Instrument *TMS320C6678* multi-core *DSP* with a 64 MiB hyperspectral image.

Figure A.38 shows the results of *AMEE* using *SAM* (A.38(a)) and *SID* (A.38(b)) distances on a Texas Instruments *TMS320C6678 DSP*. In order to parallelize the algorithm we have taken advantage of the *OpenMP* support provided by Texas Instruments' compiler. To give a sense of the scalability we show results on one single core and using the eight available on the *DSP*.

This kind of device presents two advantages over *GPUs*. On one hand, the power required is much more restricted; on the other hand, these *DSP* are space-certified. These two advantages make them an ideal solution to explore hyperspectral imaging applications on satellites, given that they can be used in these environments directly.

In summary, development environments such as *OpenCL*, which is not specific to *GPUs*, and focusing on the optimization of high-level primitives to build specific application, emerge as a better choice than *GPU*-specific *APIs* and optimizing on a per-application basis. As for the platform, multi-core *DSPs* such as *TMS320C6678* arise as a promising option for on-board high performance computing.

Apéndice B

Introducción a OpenGL

OpenGL es una interfaz software para el hardware gráfico, destinada a la programación de aplicaciones gráficas interactivas en tres dimensiones. *OpenGL* es una *API* (interfaz para la programación de aplicaciones, de sus siglas en inglés *Application Programming Interface*) procedimental que manipula una máquina de estados. El estado de dicha máquina define la forma en la que la geometría que enviamos a representación será transformada en la imagen sintética final.

OpenGL define varios tipos de procedimientos según su funcionalidad:

1. Procedimientos para componer la escena:

   a) Luces: Habilitar/deshabilitar luces, establecer su tipo (direccional/omnidireccional), su color y su posición en la escena.

   b) Cámara: La posición, proyección y el volumen de vista de la cámara. La posición y proyección se establecen mediante una matriz de estado que define el cambio de sistema de coordenadas y la proyección.

También se establece la forma de la imagen resultante y el volumen que proyectará la cámara mediante otras llamadas.

c) Objetos: La posición, rotación y escalado de los objetos también se define mediante una matriz de cambio de base, la matriz de modelado y vista, que transformará en la etapa de vértices toda la geometría enviada a representación.

2. Procedimientos para establecer el aspecto de los objetos:

   a) Propiedades del material.

   b) Textura.

   c) Color.

   Todas estas propiedades se establecen como un estado de la máquina de estados de *OpenGL*, de forma que todo lo que se envíe a representar se verá afectado por las propiedades del material, el color y la textura que esté establecida en ese momento en la máquina de estados.

3. Procedimientos para enviar primitivas a representar. Estos procedimientos no actúan sobre el estado de la máquina, son los que inician el dibujado de la escena y, por lo tanto, se verán afectados por el estado establecido por los procedimientos de composición y de aspecto. *OpenGL* envía los objetos a representación en forma de primitivas geométricas:

   a) Puntos.

   b) Segmentos.

   c) Polígonos (de 3 lados o más).

```
/* Crea un contexto de representación GL para X11 (Unix/Linux) */
void create_rendering_context( )
{
    /* Conectamos con el servidor X */
    Display* dpy = XOpenDisplay( NULL );

    /* Atributos del contexto */
    int fbAttr[] = {
        GLX_RGBA,           /* Sintetizamos imágenes RGBA */
        GLX_RED_SIZE, 8,    /* 8 bits para el rojo */
        GLX_GREEN_SIZE, 8, /* 8 bits para el verde */
        GLX_BLUE_SIZE, 8, /* 8 bits para el azul */
        GLX_ALPHA_SIZE, 8, /* 8 bits para la transparencia */
        GLX_DOUBLEBUFFER, True, /* Queremos dos framebuffers */
        None
    };
    XVisualInfo* vi = glXChooseVisual( dpy, DefaultScreen(dpy), fbAttr );

    /* Creamos el contexto */
    GLXContext cx = glXCreateContext( dpy, vi, 0, GL_TRUE );

    /* Creamos una ventana de 300x300 en (0,0) */
    XSetWindowAttributes swa;
    swa.colormap = XCreateColormap( dpy, RootWindow(dpy, vi->screen),
            vi->visual, AllocNone );
    swa.border_pixel = 0;
    swa.event_mask = StructureNotifyMask;
    Window win = XCreateWindow( dpy, RootWindow(dpy, vi->screen),
            0, 0, 300, 300, 0, vi->depth,InputOutput, vi->visual,
            CWBorderPixel|CWColormap|CWEventMask, &swa );
    XMapWindow( dpy, win );

    /* Asociamos nuestro contexto de representación a la ventana */
    glXMakeCurrent( dpy, win, cx );
}
```

FIGURA B.1: Código que crea un contexto de representación de *OpenGL* en una ventana de un sistema gráfico X11, típico en sistemas Unix y *GNU/Linux*.

```
void render_scene_loop( Window win, Display* dpy )
{
    /* Creamos una textura para nuestros objetos */
    /* Habilitamos el uso de texturas 2D */
    glEnable( GL_TEXTURE_2D );
    int tex;
    /* Textura rojo/azul como un tablero de damas */
    unsigned char tdata[] = {
        255,0,0, 255,0,0, 0,0,255, 0,0,255,
        255,0,0, 255,0,0, 0,0,255, 0,0,255,
        0,0,255, 0,0,255, 255,0,0, 255,0,0,
        0,0,255, 0,0,255, 255,0,0, 255,0,0 };
    glGenTextures( 1, &tex );
    glBindTexture( GL_TEXTURE_2D, tex );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
    glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA, 4, 4, 0, GL_RGB, GL_UNSIGNED_BYTE,
            tdata );
    while( !terminar ) {
        /* Borramos los buffers */
        glClearColor( 0.5, 0.5, 0.5, 1 );
        glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

        /* Configuramos la escena */
        /* Luces */
        float l0_pos[] = { 2.0, 0.0, 2.0 };
        glEnable( GL_LIGHTING );
        glEnable( GL_LIGHT0 );
        glLightfv( GL_LIGHT0, GL_POSITION, l0_pos );
        /* Cámara */
        glViewport( 0, 0, 300, 300 );
        glMatrixMode( GL_PROJECTION );
        glLoadIdentity( );
        glFrustum( -1.0, 1.0, -1.0, 1.0, 0.2, 1000.0 );
        glTranslatef( 0.0, 0.0, -1.5 );
```

FIGURA B.2: Aplicación simple en *OpenGL* (I). En esta parte del código se ve la creación de una textura, establecimiento del estado de la máquina *OpenGL* y preparación de una luz y la cámara.

```
        /* Objetos */
        glMatrixMode( GL_MODELVIEW );
        glLoadIdentity( );
        /* Rotado 45º en el eje (0,1,0) y trasladado a (0,0,-1.25) */
        glTranslatef( 0.0, 0.0, −1.25 );
        glRotatef( 45.0, 0.0, 1.0, 0.0 );
        glBindTexture( GL_TEXTURE_2D, tex );
        /* Empieza el dibujado de primitivas: triángulos */
        /* Pintaremos 2 triángulos formando un cuadrado de 6x6 */
        glBegin( GL_TRIANGLES );
            glTexCoord2f( 0.0, 0.0 ); /* Propiedades del vértice */
            glNormal3f( 0.0, 0.0, 1.0 );
            glVertex2f( −3.0, −3.0 );
            glTexCoord2f( 1.0, 0.0 );
            glNormal3f( 0.0, 0.0, 1.0 );
            glVertex2f( 3.0, −3.0 );
            glTexCoord2f( 0.0, 1.0 );
            glNormal3f( 0.0, 0.0, 1.0 );
            glVertex2f( −3.0, 3.0 ); /* Primer triángulo completo */
            glTexCoord2f( 0.0, 1.0 );
            glNormal3f( 0.0, 0.0, 1.0 );
            glVertex2f( −3.0, 3.0 );
            glTexCoord2f( 1.0, 0.0 );
            glNormal3f( 0.0, 0.0, 1.0 );
            glVertex2f( 3.0, −3.0 );
            glTexCoord2f( 1.0, 1.0 );
            glNormal3f( 0.0, 0.0, 1.0 );
            glVertex2f( 3.0, 3.0 ); /* Ya tenemos el cuadrado */
        glEnd( );
        glFlush( );
        /* Estamos usando un doble framebuffer: pintas en uno y muestras otro */
        /* Cambiamos el buffer que se muestra por el que estamos pintando */
        /* Es dependiente del sistema operativo también */
        glXSwapBuffers( dpy, win );
    }
}
```

FIGURA B.3: Aplicación simple en *OpenGL* (II). En esta parte del código se ve el dibujado de un cuadrado utilizando dos triángulos como primitivas.

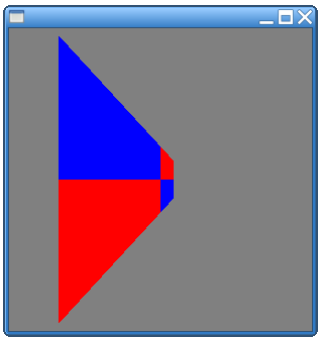


FIGURA B.4: Resultado de la ejecución del código de las figuras B.1, B.2 y B.3

OpenGL también ofrece procedimientos para realizar composición de escenas en post-producción y para la manipulación de diferentes *buffers* típicos en aplicaciones gráficas, como el *framebuffer*, que contiene la información de lo que vemos en la pantalla, el *Z-buffer*, que contiene información de la profundidad espacial de lo que hay en el *framebuffer* (útil para resolver la oclusión entre distintos objetos), u otros con un propósito más específico como los *buffers* de acumulación o el *stencil buffer*. No vamos a entrar a detallar esta parte por ser muy específica de aplicaciones puramente gráficas y estar fuera del alcance de este trabajo.

En una aplicación típica de *OpenGL*, encontraremos una serie de comandos específicos del sistema operativo que serán los encargados de crear un contexto de representación. Establecerán el tamaño de la ventana de dibujado y sus propiedades, como la profundidad del color, el tipo de *buffers* que tendrá asociados u otras más relacionadas con el sistema operativo (como si la ventana será o no redimensionable). Podemos ver un ejemplo de un código sencillo para hacer esto en un sistema *GNU/Linux* corriendo un servidor X en la figura B.1.

Tras establecer el contexto de representación, entraremos en un bucle en el

que se borrarán los *buffers*, se compondrá una escena (configurando las luces y la cámara) y, para cada objeto, configuraremos el estado de la máquina con el aspecto deseado de dicho objeto, lo situaremos en la escena con la matriz de modelado y vista, y lo enviaremos a representar primitiva a primitiva. Podemos ver un un trozo de una aplicación *OpenGL* sencilla en las figuras B.2 y B.3. La aplicación dibuja un cuadrado con una textura sencilla superpuesta, y el resultado se puede ver en la figura B.4.

Para aplicaciones gráficas más complejas, también disponemos de procedimientos para cargar programas de vértices y fragmentos personalizados, y sustituir las respectivas etapas por los programas externos. Esto es útil para conseguir imágenes sintéticas de aspecto fotorrealista, y es lo que utilizamos en el capítulo 3 para explotar las tarjetas gráficas como sistemas *CMP* de propósito general.

Apéndice C

Biblioteca de desarrollo para GPGPU

En arquitecturas anteriores a la quinta generación (*G70* y posteriores), así como en arquitecturas de *GPUs* empotradas (la serie *Tegra* de *NVIDIA*, por ejemplo), la única forma que disponemos para acceder al hardware son las bibliotecas de *OpenGL*. *OpenGL* alimenta los recursos de cómputo con distintos tipos de datos: los procesadores de vértices se alimentan con información de vértices y los de fragmentos con la salida del *rasterizador*, que depende de dichos vértices.

Como se puede ver en la figura 1.6 de la introducción, las *GPUs* suelen incorporar un número de procesadores de fragmentos bastante superiores a los procesadores de vértices. Esto es debido a que, por lo general, un pequeño grupo de vértices generará un número significativamente mayor de fragmentos. Por este motivo decidimos centrarnos en aprovechar los procesadores de fragmentos que, no solo son más numerosos, sino que en los casos en los que

son diferentes, suelen presentar una funcionalidad más útil en el cómputo de propósito general.

Lo básico para poder hacer esto es que la salida de esos procesadores pueda ser utilizada posteriormente como entrada. Esto no es obvio ya que, por su naturaleza dedicada a la síntesis de gráficos, la salida de los procesadores de fragmentos se dirige a una porción especial de la memoria de vídeo, el *framebuffer*, que es la que contiene la información de lo que se muestra en pantalla. A su vez, los procesadores de fragmentos solo tienen acceso a unas regiones de memoria conocidas como *texturas*. Afortunadamente, como podemos observar en la figura C.1, con el tiempo se han ido añadiendo una serie de extensiones que habilitan la técnica que se conoce como *render-to-texture* o *síntesis en una textura*, que permite que la salida de los procesadores de fragmentos vaya a esas regiones de memoria en lugar de ir a dicho *framebuffer*.

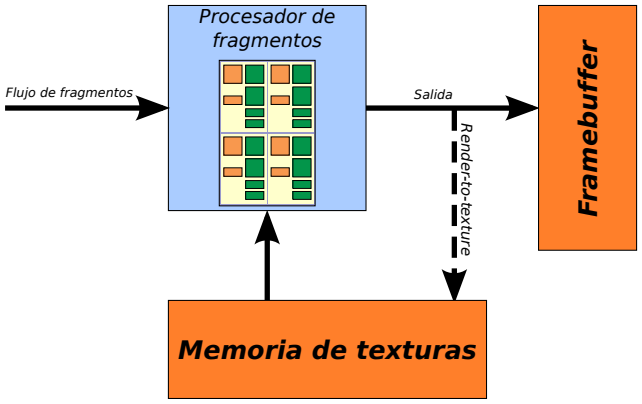


FIGURA C.1: Con el mecanismo de *render-to-texture* podemos realimentar los procesadores de fragmentos con su salida de forma eficiente.

Retomando el esquema presentado el capítulo 3.2, en la figura C.2 podemos ver el esquema general de la abstracción hecha sobre *OpenGL*.

Como ya se ha comentado, la funcionalidad de la que deberemos disponer

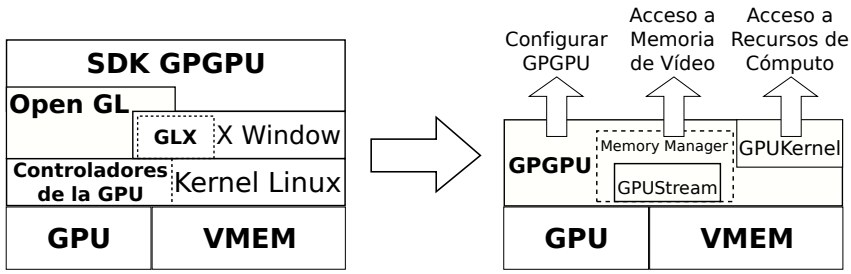


FIGURA C.2: Esquema de la abstracción de nuestra biblioteca de desarrollo para *GPGPU*.

será la siguiente:

- Gestión del entorno: Inicialización, acceso a los recursos, sincronización y finalización.
- Gestión de memoria: Reserva y liberación de flujos (*streams*).
- Acceso a los recursos de cómputo: Construcción e invocación de *kernels*.

En nuestra biblioteca hemos encapsulado toda la funcionalidad en tres clases principales: *GPGPU*, *GPUStream* y *GPUKernel*.

La clase *GPGPU*

GPGPU es la clase principal y la encargada tanto de la gestión del entorno como de la gestión de memoria. Es una clase estática con los siguientes métodos:

- *Initialize*: Inicializa el entorno de *GPGPU* y el gestor de memoria. Toma ciertos parámetros que permiten optimizar la gestión de memoria para nuestra aplicación.
- *getGPUStream*: Reserva espacio en la memoria para un flujo de las dimensiones solicitadas y lo devuelve para su uso.

- *freeGPUStream*: Libera los recursos ocupados por un flujo.
- *waitForCompletion*: Bloquea la ejecución hasta que se completa la ejecución del último *kernel* invocado.

La clase *GPUStream*

La clase *GPUStream* encapsula nuestros flujos de datos. Se solicitan y liberan a través de la interfaz de la clase *GPGPU* descrita con anterioridad. Los métodos principales de este tipo de objetos son los siguientes:

- *write/writeAll*: Rellena (parcial o totalmente) el contenido del flujo a partir de un vector.
- *read/readAll*: Lee el contenido del flujo (o una parte) de vuelta a la memoria principal.
- *setValue*: Rellena todo el flujo con un valor constante.
- *getSubStream*: Devuelve una referencia a una parte del flujo contigua en dos dimensiones. Dicha referencia puede ser tratada como un flujo; es decir, la podemos utilizar para leer y escribir esa región del flujo o como parámetro para los *kernel*.

La clase *GPUKernel*

La clase *GPUKernel* representa nuestros *kernels*, funciones especiales que toman flujos como parámetros. Los objetos de la clase *GPUKernel* se construyen a partir de un fichero externo que contiene el código de nuestro *kernel*

(más adelante se darán más detalles), y disponen de métodos para dar valor a parámetros que no sean objetos de la clase *GPUStream* a partir de su nombre.

Para invocar un *kernel*, podemos utilizar directamente la clase que lo contiene como si fuese una función con los flujos que toma como parámetro en orden, primero los de entrada y luego los de salida. De este modo, si queremos invocar un *kernel* que haga la suma de dos vectores (como el de la figura 3.2), haríamos lo siguiente:

GPUKernel suma(*"suma.fp"*);
suma(a, b, c);

Donde el fichero *"suma.fp"* contendría el código del *kernel* de suma vectorial, y *a*, *b* y *c* son tres objetos de la clase *GPUStream* que hemos reservado e inicializado con anterioridad.

La gestión de memoria

Como ya se ha comentado, el acceso a la memoria de una *GPU* se realiza a través de unas entidades, enteramente relacionadas con la síntesis de gráficos, conocidas como *texturas*. Una *textura* es una región en la memoria de vídeo que contiene información sobre el aspecto de la geometría que queremos dibujar. Esto es importante porque de su uso se derivan ciertas particularidades sobre su utilización:

1. Los elementos de una textura (o *texel*, contracción del inglés *texture element*) son colores en el espacio de color *RGBA*. Eso quiere decir que cada componente de una textura contiene cuatro valores.

2. En síntesis de gráficos las texturas presentan localidad espacial y temporal, con la particularidad de que dicha localidad es bidimensional. Esto hace que las memorias *cache* que se utilizan son diseñadas para aprovechar localidad en dos dimensiones.

Esto quiere decir que lo óptimo es tomar como referencia texturas bidimensionales con elementos vectoriales de anchura cuatro. Podríamos haber utilizado texturas de una sola dimensión y no vectoriales, pero por un lado apenas aprovecharíamos memoria *cache* y por otro no podríamos beneficiarnos de las unidades vectoriales que poseen los procesadores de una *GPU*.

Con esto en mente, pasamos a nuestra siguiente decisión de diseño: Cómo mapear los flujos en texturas.

Se nos presentan dos opciones válidas:

1. Utilizar una textura para cada flujo.

2. Utilizar una única textura y que los flujos sean referencias a distintos puntos de la textura.

En el primer caso la implementación sería mucho más sencilla, pero como cada flujo estaría en su propio espacio de direcciones, no podríamos disponer de referencias a un flujo indeterminado dentro de otro dado (punteros). Esto es debido a que cada textura se direcciona de manera individual y es técnicamente imposible hacer una referencia a una textura no determinada en tiempo de compilación.

También descartamos una solución mixta donde podamos pedir flujos que soporten referencias (reservados como secciones de una misma textura) y flu-

jos con direccionamiento único debido a que complicaría aún más la implementación y, al romperse la homogeneidad, complicaría también la programación de *kernels*.

Así pues, optamos por el espacio de direccionamiento unificado, y utilizaremos una única textura donde los flujos serán desplazamientos dentro de esa textura. De esta forma, nuestros flujos pueden contener referencias a otros flujos simplemente almacenando desplazamientos.

Por último, para gestionar dicha memoria necesitaremos de un gestor con una política de emplazamiento. Nosotros hemos optado por una política en la gestión de memoria de primer encaje (*first fit*), de forma que cuando se solicite un flujo de unas determinadas dimensiones, se buscará el primer hueco libre que pueda albergarlo. La gestión sería similar a como la realiza un sistema operativo moderno solo que con un espacio de direcciones bidimensional, lo cual supone unas ciertas particularidades a la hora de reservar y liberar memoria, y hace que el problema de la fragmentación sea más grave.

Programación de *kernels*

Para el desarrollo de *kernels*, vamos a utilizar el lenguaje de programación *Cg* (*C for graphics*)[FK03]. Originalmente desarrollado por *NVIDIA*, tiene soporte en todas las plataformas principales, tanto *software* como *hardware*. Hereda mucha de su sintaxis de *C*, y añade una serie de características relacionadas con la síntesis de gráficos.

En nuestra biblioteca de desarrollo esos elementos relacionados con gráficos tienen un impacto mínimo. El desarrollador solo tiene que saber que hay una

forma particular de declarar los parámetros de entrada de tipo flujo, y que el acceso al contenido se hace mediante funciones específicas.

```
void main(  in  float2 a_address : TEXCOORD0,
            in  float2 b_address : TEXCOORD1,
            out float4 c_value   : COLOR,
            const uniform samplerRECT mem )
{
    float4 a = texRECT( mem, a_address );
    float4 b = texRECT( mem, b_address );
    c = a + b;
}
```

FIGURA C.3: *Kernel* sencillo escrito en *Cg*.

En el código de la figura C.3 podemos ver estos elementos para el caso de una suma de vectores. Declaramos una función principal al estilo *C* cuyos parámetros deberán ser especificados como de entrada, de salida o uniformes. La función *kernel* siempre deberá incluir un parámetro uniforme del tipo *samplerRECT* que será la referencia a nuestra memoria. Los flujos de entrada se pasan al *kernel* como direcciones dentro del espacio de direcciones de dicha memoria, que al ser bidimensional, requerirá dos coordenadas; de ahí que el tipo de los flujos de entrada sea el tipo especial *float2*, que contiene dos valores en coma flotante. A la declaración de cada flujo hay que añadir lo que se conoce como un *"vínculo"* (*binding*), que le dice a *Cg* a qué flujo de entrada corresponde. El *"vínculo"* se especifica escribiendo *":"* seguidos de la palabra clave *TEXCOORDn*, donde *"n"* es el orden del flujo de entrada (*TEXTCOORD0* lo vincula al primer flujo de entrada, y así sucesivamente).

Para acceder al elemento correspondiente de dicho flujo, se utiliza la función especial de direccionamiento *"texRECT"*, que toma como parámetros el espacio de direcciones y las coordenadas. En nuestro caso el espacio de direcciones será el único disponible, y las coordenadas serán las que se pasan de

entrada por cada flujo a nuestra función *kernel*. La función *"texRECT"* nos devolverá un elemento del tipo *float4* que contiene los cuatro valores correspondientes a ese elemento del flujo.

Para los flujos de salida hay que tener en cuenta que nuestra biblioteca de desarrollo soporta un único flujo de salida por *kernel*. En las *GPUs* más modernas es posible tener más, pero ni es especialmente eficiente ni está soportado por todas las tarjetas que permiten hacer *GPGPU*, así que lo limitamos a un único flujo.

Ese flujo se especifica como un elemento *float4* (cuatro valores en coma flotante) cuyo *"vínculo"* es *COLOR*. A diferencia de los flujos de entrada, el flujo de salida es un valor, no unas coordenadas dentro de un flujo. Si se quiere leer un valor del flujo de salida, podemos especificar un flujo adicional de entrada con el *"vínculo" WPOS*. De ese modo podemos utilizar el flujo de salida también como una entrada sin necesidad de especificar el flujo dos veces, una como entrada y otra como salida.

Como se puede observar, a pesar de que hay elementos específicos de gráficos, al desarrollador no le es necesario tener un conocimiento específico ya que su uso es mecánico. Sería sencillo construir un compilador que transformase de una sintaxis de flujos a una función *Cg* compatible con nuestra biblioteca de desarrollo. Para elementos más específicos del lenguaje *Cg* se puede consultar el libro *The Cg Tutorial*[FK03], donde podremos encontrar listas con las funciones matemáticas y las distintas operaciones vectoriales disponibles.

Bibliografía

[App68] A. Apple. Some Techniques for Machine Rendering of Solids. In *Proceedings of the AFIPS Conference*, volume 32, pages 37–45, 1968.

[BFH+03] I. Buck, T. Foley, D. Horn, J. Sugerman, P. Hanrahan, M. Houston, and K. Fatahalian. BrookGPU web site. http://graphics.stanford.edu/projects/brookgpu/, 2003.

[BK03] P. Brown and M. J. Kilgard. GeForce FX 5900 product. Disponible en http://www.nvidia.com/page/fx_5900.html, 2003.

[Cha03] C.-I Chang. *Hyperspectral Imaging: Techniques for Spectral Detection and Classification*. Kluwer: New York, 2003.

[Cor05] NVIDIA Corporation. GeForce 7800 GTX. Disponible en http://www.nvidia.com/page/geforce_7800.html, 2005.

[FK03] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[GR05] J. Gummaraju and M. Rosenblum. Stream Programming on General-Purpose Processors. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–354. IEEE, 2005.

[Gro08] Khronos Group. OpenCL web site. http://www.khronos.org/opencl/, 2008.

[GTGB84] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennet Bataille. Modeling the Interaction of Light Between Diffuse Surfaces. *Computer Graphics*, 18(3):213–222, 1984.

[KLC04] R. Kiefer, T. Lillesand, and J. Chipman. *Remote Sensing and Image Interpretation*. John Wiley and Sons, Inc., Hoboken, NJ, USA, 2004.

[LE10] Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended openmp programming and tuning for gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[Moo65] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, 1965.

[NASA] National Aeronautics and Space Administration. Airborne Visible/InfraRed Imaging Spectrometer. http://aviris.jpl.nasa.gov/html/aviris.freedata.html.

[NJK11] G. Noaje, C. Jaillet, and M. Krajecki. Source-to-Source Code Translator: OpenMP c to cuda. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 512–519, 2011.

[nvi12] NVIDIA CUDA programming manual. Available at. http://developer.nvidia.com, 2012.

[Ope11] OpenACC. OpenACC web site. http://www.openacc.org, 2011.

[PBBL07] Josep M. Perez, Pieter Bellens, Rosa M. Badia, and Jesus Labarta. CellSs: Making it Easier to Program the Cell Broadband Engine Processor. *IBM Journal of R&D*, 51(5):593–604, 2007.

[PMPP02] A. Plaza, P. Martinez, R. Perez, and J. Plaza. Spatial/spectral Endmember Extraction by Multidimensional Morphological Operations. *IEEE Transactions in Geoscience and Remote Sensing*, 40(9):2025–2041, September 2002.

[Rot82] Scott D. Roth. Ray Casting for Modeling Solids. *Computer Graphics and Image Processing*, 18(2):109–144, 1982.

[RRB+08] S. Ryoo, C.I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and Wen mei Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[SB03] G. Shaw and H. Burke. Spectral Imaging for Remote Sensing. *Lincoln Laboratory Journal*, 14(1):3–28, 2003.

[SEC13] SECO. Carma web site. http://www.seco.com/carma, 2013.

[SM02] G. Shaw and D. Manolakis. Signal Processing for Hyperspectral Image Exploitation. *IEEE Signal Processing Magazine*, 19(1):12–16, 2002.

[Soi03] P. Soille. *Morphological Image Analysis: Principles and Applications, 2nd ed.* Berlin: Springer, 2003.

[SPT+07] J. Setoain, M. Prieto, C. Tenllado, A. Plaza, and F. Tirado. Parallel Morphological Endmember Extraction Using GPUs. *IEEE Geoscience and Remote Sensing Letters*, pages 441–445, 2007.

[SPTP08] J. Setoain, M. Prieto, C. Tenllado, and A. Plaza. GPUs for Parallel on-board Hyperspectral Image Processing. *International Journal of High Performance Computing Applications*, pages 424–437, 2008.

[SPTT08] J. Setoain, M. Prieto, C. Tenllado, and F. Tirado. Real-Time On-Board Hyperspectral Image Processing Using GPUs. In *High Performance Computing in Remote Sensing*, chapter 18, pages 411–451. Chapman And Hall, 2008.

[STG+08] J. Setoain, C. Tenllado, J. I. Gomez, M. Arenaz, M. Prieto, and J. Tourino. Towards Automatic Code Generation for GPU Archi-

tectures. In *PARA'08: Workshop on State-of-the-Art in Scientific and Parallel Computing*, 2008.

[STP+06] J. Setoain, C. Tenllado, M. Prieto, D. Valencia, A. Plaza, and J. Plaza. Parallel Hyperspectral Image Processing on Commodity Graphics Hardware. In *ICPPW '06: Proceedings of the 2006 International Conference Workshops on Parallel Processing*, pages 465–472, Washington, DC, USA, 2006. IEEE Computer Society.

[TI12] Texas Instruments. Tms320c6678. http://www.ti.com/product/tms320c6678, 2012.

[TSPT08] C. Tenllado, J. Setoain, M. Prieto, and F. Tirado. 2D-DWT on GPUs: Filter-Bank versus Lifting. *IEEE Transaction on Parallel and Distributed Systems*, pages 299–309, 2008.

[VGC+93] G. Vane, R. Green, T. Chrien, H. Enmark, E. Hansen, and W. Porter. The Airborn Visible/Infrared Imaging Spectrometer. *Remote Sensing of Environment*, 44:127–1243, 1993.

[Wal91] David W. Wall. Limits of Instruction-Level Parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 26, pages 176–189, New York, NY, 1991. ACM Press.

Índice de figuras

1.1. Predicción y ajuste a la Ley de Moore del número de transistores y la escala de integración. 2

1.2. En la arquitectura propuesta por John Von Neumann, un computador consiste en una memoria que contiene tanto los datos como los programas, y en una CPU que es la encargada de ejecutar los programas que procesan dichos datos. Una CPU está compuesta por una unidad de control, que gestiona el flujo de ejecución de los programas, y una unidad de proceso, que gestiona el procesamiento de los datos. En la figura, *PC* (*Program Counter*) es el contador de programa, encargado de seguir el flujo de ejecución de las instrucciones de un programa. *ALU* (*Arithmetic Logic Unit*) es la unidad aritmético-lógica, que realiza operaciones aritméticas y lógicas sobre datos. 3

1.3. Predicción y ajuste a la Ley de Moore del número de transistores y la frecuencia de procesamiento. 7

1.4. Tubería gráfica. Como entrada toma un conjunto de vértices y sus atributos, que se transforman en las sucesivas etapas en la imagen de la escena representada por dicha geometría. 12

1.5. Diagrama de bloques de una GPU de 4ª generación. 15

1.6. Diagrama de bloques de las GPUs (a) NVIDIA G70 y (b) ATI-RADEON R520. 17

1.7. Diagrama de bloques de una GPU NVIDIA G80. 19

1.8. *Rage*, de *ID Software*. 21

1.9. *Bioshock*, de *Irrational Games*. 22

1.10. La distancia en potencia de cálculo pico y ancho de banda entre *GPUs* y *CPUs* se ha ido incrementando notablemente con los años (gráficas obtenidas del manual de programación de *CUDA* [nvi12]). 23

1.11. La resolución espectral en una imagen hiperespectral nos permite identificar los materiales de la imagen en base a su espectro. 26

2.1. Cómputo de la distancia acumulada de los vectores-píxel para un elemento estructural $\mathbb{Z}^2(B)$ centrado en (x, y). 37

2.2. Búsqueda de los vectores-píxel con la máxima y mínima distancia acumulada, y cómputo del valor *MEI* para el vector-píxel $\boldsymbol{f}(x, y)$. 37

2.3. Comparación del rendimiento de *AMEE* multi-pasada con *AMEE* original 40

3.1. Unidad de procesamiento de flujos genérica. 49

3.2. Suma secuencial de vectores y su transformación al modelo de procesamiento de flujos. 50
3.3. Representación gráfica de un programa en el modelo de procesamiento de flujos. 51
3.4. Nuestra biblioteca de desarrollo (*SDK*) abstrae toda la complejidad del sistema del sistema operativo y *OpenGL* con un modelo de desarrollo basado en flujos y *kernels*. 52
3.5. Código para la creación de un contexto de síntesis en *OpenGL* que soporte "síntesis en una textura", "síntesis fuera de la pantalla" y texturas con componentes reales. 54
3.6. Código para configurar la máquina de estados de *OpenGL* para que el espacio de proyección coincida con el espacio de la pantalla. 55
3.7. Preparar la síntesis sobre una textura. En nuestro caso, la textura $tex[2]$ es la que guarda los resultados del cómputo. 56
3.8. Cargamos nuestro programa de fragmentos y lo enviamos a la *GPU* 57
3.9. Cargamos en $tex[0]$ y $tex[1]$ nuestros datos de entrada y dibujamos un cuadrado del tamaño de la pantalla, asignando a sus vértices dichas texturas. 58
3.10. Utilizando nuestra biblioteca de desarrollo. 59
3.11. La imagen se divide en regiones espaciales (R.E.) para acomodarla a la cantidad de memoria disponible en la *GPU*. Cada región espacial se divide espectralmente en flujos con cuatro bandas espectrales cada uno para aprovechar las operaciones vectoriales de la *GPU* 62

3.12. Región espacial para n pasadas. 62

3.13. Las regiones fronterizas de diferentes regiones espaciales se solapan entre ellas y con las regiones efectivas de la región espacial contigua. 63

3.14. Cada vector-píxel tiene una región de influencia de 5×5 en el cálculo de distancias acumuladas. 64

3.15. El número de distancias distintas se reduce a 24 si simplificamos la redundancia por conmutatividad. 64

3.16. Por simetría, el número de distancias que es necesario calcular para cada píxel se reduce a 12. 66

3.17. Representación gráfica del modelo de flujos para *SID AMEE*. Cómputo de la aproximación multi-pasada al MEI_B de una imagen en $\mathbb{K}^n$ (con n bandas espectrales). 72

3.18. Representación gráfica del modelo de flujos para *SAM AMEE*. Cómputo de la aproximación multi-pasada al MEI_B de una imagen en $\mathbb{K}^n$ (con n bandas espectrales). 73

3.19. Rendimiento de la implementación de *SAM AMEE* en *CPU* y *GPU* para diferentes tamaños de imagen y cinco pasadas ($I_{max} = 5$). 78

3.20. Rendimiento de la implementación de *SID AMEE* en *CPU* y *GPU* para diferentes tamaños de imagen y cinco pasadas ($I_{max} = 5$). 78

3.21. Aceleración de las implementaciones en *GPU* respecto a las implementaciones en *CPU* para distinto número de pasadas. . 79

3.22. Aceleración de la implementación de *SAM AMEE* respecto a *SID AMEE* en las diferentes plataformas experimentales, para distintos tamaños de imagen y con cinco pasadas ($I_{max} = 5$). . 80

3.23. Comparación de la aceleración dos generaciones de *CPU* (*Pentium IV Northwood* del 2003 y *Pentium IV Prescott* del 2005), y dos generaciones de tarjetas gráficas (*NV GeForce 5950 Ultra* del 2003 y *NV GeForce 7800 GTX* del 2005). 81

4.1. Una rejilla de $3x2$ bloques de $4x3$ hilos. 87

4.2. Esquema de las distintas memorias y su ámbito. 89

4.3. Reparto de bloques entre los multiprocesadores de flujo en función del número de ellos disponibles en el dispositivo. 91

4.4. Implementación en hardware del modelo SIMT de *CUDA*. . . 92

4.5. Accesos fusionados a datos tipo *float*. A la derecha, acceso fusionado con divergencia en el *warp*. 97

4.6. Accesos no fusionados a datos tipo *float*. A la izquierda, por acceso no secuencial. A la derecha, por accesos no alineados. . 98

4.7. A la izquierda, acceso a datos de tipo *float* no fusionados por acceso a memoria no contigua. A la derecha, acceso a datos de tipo *float3* no fusionados. 99

4.8. Algoritmo simplificado para estudio del rendimiento. La imagen de entrada es I; f_1, f_2, f_3 y f_4 son funciones matemáticas dependientes de los parámetros que reciben. 102

4.9. Transformación de fisión de bucles. Dividimos el algoritmo de la figura 4.8 en dos, expandiendo la variable *acc* debido a la dependencia dentro del bucle. 106

4.10. Tiempo de ejecución en función del número de hilos por bloque. Comparamos una versión del algoritmo en un único *kernel* con una versión en la que hemos realizado una fisión del bucle principal. (a) Con acceso directo a la memoria. (b) Con acceso a través de *cache* de texturas. 107

4.11. Efecto del desenrollado de bucles en el rendimiento. Forzamos la compilación con 12, 16 y 20 registros para establecer la ocupación de los procesadores al 83 %, 67 % y 50 % respectivamente. Las pruebas se han realizado con 128 hilos por bloque. 108

4.12. Efecto del uso de las distintas memorias en el rendimiento. Los resultados, en función del método de acceso a la imagen, son: [1] Acceso a través de la *cache* de texturas; [2] Precarga en memoria compartida; [3] Precarga en memoria compartida a través de la *cache* de texturas; [4] Precarga en memoria compartida en el primer acceso y uso de *cache* de texturas en el segundo; [5] Precarga en memoria compartida a través de la *cache* de texturas en el primer acceso y uso de la *cache* de texturas en el segundo. 112

4.13. Resultados de la conversión directa de *OpenGL* a *CUDA*. (a) Con 5 pasadas y en función del tamaño de la imagen; (b) Con la imagen de 512MiB y en función del número de pasadas. . . 118

4.14. Resultados minimizando el número de *kernels*. (a) Con 5 pasadas y en función del tamaño de la imagen; (b) Con la imagen de 512MiB y en función del número de pasadas. 119

4.15. Resultados optimizando el acceso a la memoria. (a) Con 5 pasadas y en función del tamaño de la imagen; (b) Con la imagen de 512MiB y en función del número de pasadas. 121

4.16. Resultados con texturas bidimensionales. (a) Con 5 pasadas y en función del tamaño de la imagen; (b) Con la imagen de 512MiB y en función del número de pasadas. 122

4.17. Resultados con texturas tridimensionales. (a) Con 5 pasadas y en función del tamaño de la imagen; (b) Con la imagen de 512MiB y en función del número de pasadas. 123

4.18. Resultados de las distintas implementaciones para la versión de *AMEE* utilizando la distancia *SID*. (a) Con 5 pasadas y en función del tamaño de la imagen; (b) Con la imagen de 512MiB y en función del número de pasadas. 124

4.19. Resultados de las distintas implementaciones para la versión de *AMEE* utilizando la distancia *SAM*. (a) Con 5 pasadas y en función del tamaño de la imagen; (b) Con la imagen de 512MiB y en función del número de pasadas. 124

4.20. Resultados de las distintas implementaciones para la versión de *AMEE* en Carma utilizando la distancia *SID*. (a) Con 5 pasadas y en función del tamaño de la imagen; (b) Con la imagen de 512MiB y en función del número de pasadas. 127

4.21. Resultados de las distintas implementaciones para la versión de *AMEE* en Carma utilizando la distancia *SAM*. (a) Con 5 pasadas y en función del tamaño de la imagen; (b) Con la imagen de 512MiB y en función del número de pasadas. 128

5.1. Resultados de *AMEE* sobre el *DSP TMS320C6678* de Texas Instruments para una imagen hiperespectral de 64 MiB. 135

A.1. Prediction and adjustment of transistor count and integration scale to Moore's Law. II

A.2. In the architecture proposed by John Von Neumann, a computer consists of a memory containing data as well as programs, and a CPU in charge of running the programs on that data. A CPU is made of one control unit, which manages the execution flow of programs, and one processing unit, which manages data processing. In this figure, *PC* is the *Program Counter*, in charge of following the instruction flow of a program. The *ALU* (*Arithmetic-Logic Unit*) performs arithmetic and logic operations on data. III

A.3. The difference in computing power and bandwidth between *GPUs* and *CPUs* has been increasing throughout the years (figures from *CUDA* programming manual [nvi12]). V

A.4. Graphics pipeline. It takes as input a set of vertices and their attributes, and transforms them through consecutive stages to obtain an image of the scene represented by that geometry. . . VI

A.5. Block diagram of a 4th generation GPU. IX

A.6. Block diagram of (a) NVIDIA G70 and (b) ATI-Radeon R520 GPUs. x

A.7. Block diagram of an NVIDIA G80 GPU. xii

A.8. The spectral resolution in a hyperspectral image allows us to identify materials in the image from their spectra. xiv

A.9. Cumulative distance of the pixel-vectors given a structuring element $\mathbb{Z}^2(B)$ centered in (x, y). xxi

A.10. Finding the pixel-vectors with maximum and minimum cumulative distance, and *MEI* computation for the pixel-vector $\boldsymbol{f}(x, y)$. xxii

A.11. Performance comparison between multi-pass *AMEE* and the original *AMEE*. xxv

A.12. Graphic representation of a program in the stream processing model. xxxi

A.13. Our *SDK* abstracts the complexity of the operating system and *OpenGL* with a development framework based on streams and *kernels*. xxxi

A.14. The image is divided into spacial regions (S.R.) to make it fit inside the available video memory. In order to take advantage of the vector operations provided by the *GPU*, each spacial region is further divided into streams with four spectral bands each. . xxxiv

A.15. Spacial region for n-pass *AMEE* xxxiv

A.16. By symmetry, we only need to compute 12 distances per pixel-vector. xxxv

A.17.Graphics representation of the stream model for *SID AMEE*. Computing the approximated MEI_B of an image in $\mathbb{K}^n$ using multi-pass *AMEE* XXXVII

A.18.Graphics representation of the stream model for *SAM AMEE*. Computing the approximated MEI_B of an image in $\mathbb{K}^n$ using multi-pass *AMEE*. XL

A.19.Performance of *SAM AMEE* on *CPU* and *GPU* for different image sizes and five passes ($I_{max} = 5$). XLIII

A.20.Performance of *SID AMEE* on *CPU* and *GPU* for different image sizes and five pases ($I_{max} = 5$). XLIII

A.21.Speedup of *GPU* implementations compared to *CPU* implementations. XLIV

A.22.Speedup of *SAM AMEE* compared to *SID AMEE* on all the experimental platforms ($I_{max} = 5$). XLV

A.23.Speedup comparison between two *CPU* generations (2003's *Pentium IV Northwood* and 2005's *Pentium IV Prescott*) and two *GPU* generations (2003's *NV GeForce 5950 Ultra* and 2005's *NV GeForce 7800 GTX* del 2005). XLV

A.24.Simplified algorithm. I is the input image; f_1, f_2, f_3 and f_4 are functions that depend on their input parameters. XLIX

A.25.Loop splitting. We divide the algorithm from figure A.24 in two, expanding the variable *acc* to solve the dependency inside the original loop. LII

A.26.Execution time depending on the number of threads per block. We compare a single *kernel* algorithm with a two *kernel* version in which the main loop's been split into two. (a) Accessing directly to main memory. (b) Accessing through texture *cache*. LIII

A.27.Loop unrolling impact on performance. We force a compilation with 12, 16 and 20 register to set the occupancy at 83 %, 67 % and 50 % respectively. All test are launched with 128 threads per block. LIV

A.28.Effects of using the different levels of the memory hierarchy. Depending on the method used to fetch image data, the results are: [1] Access through texture *cache*; [2] Pre-load in shared memory; [3] Pre-load in shared memory through texture *cache*; [4] Pre-load in shared memory in the first loop, fetch data through texture *cache* in the second one; [5] Pre-load in shared memory through texture *cache* in the first loop, fetch data through texture *cache* in the second one. LVI

A.29.Results for a direct port from *OpenGL* to *CUDA*. (a) Depending on the image size and making 5 passes. (b) Depending on the number of passes over a 512 MiB image. LX

A.30.Results of minimizing the number of *kernels*. (a) Depending on the image size and making 5 passes. (b) Depending on the number of passes over a 512 MiB image. LXI

A.31.Results using padded memory. (a) Depending on the image size and making 5 passes. (b) Depending on the number of passes over a 512 MiB image. LXII

A.32.Results using two-dimensional textures. (a) Depending on the image size and making 5 passes. (b) Depending on the number of passes over a 512 MiB image. LXIII

A.33.Results using three-dimensional textures. (a) Depending on the image size and making 5 passes. (b) Depending on the number of passes over a 512 MiB image. LXIV

A.34.Comparison of the different versions of *SID AMEE*. (a) Depending on the image size and making 5 passes. (b) Depending on the number of passes over a 512 MiB image. LXIV

A.35.Comparison of the different versions of *SAM AMEE*. (a) Depending on the image size and making 5 passes. (b) Depending on the number of passes over a 512 MiB image. LXV

A.36.Results for the five different *SID AMEE* implementations on *Carma*. (a) Performing 5 passes on images with different sizes; (b) Using a 512MiB image and performing a different number of passes. LXVIII

A.37.Results for the five different *SAM AMEE* implementations on *Carma*. (a) Performing 5 passes on images with different sizes; (b) Using a 512MiB image and performing a different number of passes. LXIX

A.38.*AMEE* on Texas Instrument *TMS320C6678* multi-core *DSP* with a 64 MiB hyperspectral image. LXXIII

B.1. Código que crea un contexto de representación de *OpenGL* en una ventana de un sistema gráfico X11, típico en sistemas Unix y *GNU/Linux*. LXXVII

B.2. Aplicación simple en *OpenGL* (I). En esta parte del código se ve la creación de una textura, establecimiento del estado de la máquina *OpenGL* y preparación de una luz y la cámara. . . . LXXVIII

B.3. Aplicación simple en *OpenGL* (II). En esta parte del código se ve el dibujado de un cuadrado utilizando dos triángulos como primitivas. LXXIX

B.4. Resultado de la ejecución del código de las figuras B.1, B.2 y B.3LXXX

C.1. Con el mecanismo de *render-to-texture* podemos realimentar los procesadores de fragmentos con su salida de forma eficiente. . LXXXIV

C.2. Esquema de la abstracción de nuestra biblioteca de desarrollo para *GPGPU*. LXXXV

C.3. *Kernel* sencillo escrito en *Cg*. XC

Índice de tablas

2.1. Puntuaciones de similaridad para *AMEE SAM* y *AMEE SID* con diferente número de pasadas. Comparación con los espectros de minerales y *end-members* proporcionados por el *USGS* 43

2.2. Puntuaciones de similaridad de distintos métodos con respecto a los espectros de minerales y *end-members* proporcionados por la *USGS* (Servicio Geológico de los EEUU, del inglés *United States Geological Survey*) 44

3.1. Tabla de punteros a las distancias para el cálculo del *MEI*. X significa que la distancia debe ser 0 y S que es la simétrica. Los subíndices indican la dirección de qué distancia hay que almacenar, y entre paréntesis el desplazamiento en (x, y) necesario. Por ejemplo: Si centrando el elemento estructural en (x, y) el máximo está en la posición 5 y el mínimo en la 7, el $MEI_3(x, y) = Dist_2(x + 2, y + 1)$. 69

3.2. Características de las *CPUs* 75

3.3. Características de las *GPUs* 75

3.4. Tiempo de ejecución (en milisegundos) para la implementación en *CPU* sobre un Pentium IV Northwood (P4N) y un Pentium IV Prescott (P4P). 77

3.5. Tiempo de ejecución (en milisegundos) para la implementación en *GPU* 77

A.1. Similarity scores for *AMEE SAM* and *AMEE SID* with different number of passes. Comparison with mineral spectral and *end-members* provided by *USGS* XXVIII

A.2. Similarity scores for different methods using mineral spectral and *end-members* provided by *USGS* (*United States Geological Survey*) XXVIII

A.3. *CPU* features XXXIX

A.4. *GPU* features XLI

A.5. Execution time (in miliseconds) for *CPU* implementations on a Pentium IV Northwood (P4N) and a Pentium IV Prescott (P4P). XLII

A.6. Execution time (in miliseconds) for *GPU* implementations. . . XLII