Programa de Doctorado en Tecnologías de la Información y las Comunicaciones

Aceleración del Método de las Funciones Base Características por medio de Tarjetas Gráficas

Tesis Doctoral presentada por

D. JUAN IGNACIO PÉREZ SANZ

Alcalá de Henares, 2015

Universidad de Alcalá

DEPARTAMENTO DE AUTOMÁTICA
Edificio Politécnico
28871 Alcalá de Henares (Madrid)
Teléfono: 91 885 65 94
Fax: 91 885 69 23
secre@aut.uah.es

Dr. D. José Antonio de Frutos Redondo, Profesor Titular de la Universidad de Alcalá

Dr. D. Manuel Felipe Cátedra Pérez, Catedrático de la Universidad de Alcalá

INFORMAN: Que la Tesis Doctoral titulada **"Acelaración del método de las funciones base características por medio de tarjetas gráficas"**, presentada por D. Juan Ignacio Pérez Sanz y realizada bajo nuestra dirección, reúne méritos suficientes para optar al grado de Doctor, por lo que puede procederse a su depósito y lectura.

Alcalá de Henares, 24 de abril de 2015

Fdo: José Antonio de Frutos Redondo

Fdo: Manuel Felipe Cátedra Pérez

Universidad de Alcalá

DEPARTAMENTO DE AUTOMÁTICA
Edificio Politécnico
28871 Alcalá de Henares (Madrid)
Teléfono: 91 885 65 94
Fax: 91 885 69 23
secre@aut.uah.es

Dr. D. Agustín Martínez Hellín, Profesor Titular de la Universidad de Alcalá y Director del Departamento de Automática

INFORMA: Que la Tesis Doctoral titulada **"Acelaración del método de las funciones base características por medio de tarjetas gráficas"**, presentada por D. Juan Ignacio Pérez Sanz, y dirigida por los Doctores D. José Antonio de Frutos Redondo y D. Manuel Felipe Cátedra Pérez, cumple con todos los requisitos científicos y metodológicos para ser defendida ante un tribunal.

Alcalá de Henares, 24 de abril de 2015.

Fdo. Agustín Martínez Hellín
Director del Departamento de Automática

Programa de Doctorado en Tecnologías de la Información y las Comunicaciones

Aceleración del Método de las Funciones Base Características por medio de Tarjetas Gráficas

Tesis Doctoral presentada por
D. JUAN IGNACIO PÉREZ SANZ

Directores:
DR. D. JOSÉ ANTONIO DE FRUTOS REDONDO
DR. D. MANUEL FELIPE CÁTEDRA PÉREZ

Alcalá de Henares, 2015

Ésta es por ti, papá.

Agradecimientos

Ahora que se acerca (al menos ese es mi más ferviente deseo) el final de esta etapa, es agradable (a la par que conveniente) mirar atrás para intentar darme cuenta de cuánta gente me ha ayudado a avanzar por ella. Y para saldar, en la medida de lo posible, cuantas deudas de gratitud he ido contrayendo, debo añadir este pequeño recordatorio…

Me gustaría comenzar dando las gracias al Dr. José Antonio de Frutos, mi director, por mucho más que la dirección de esta tesis. He recorrido un largo camino a su lado, del que esta tesis supone únicamente un trecho, largo y laborioso de completar, pero uno entre otros. Gracias por todo: por su interés, por su dedicación y, sobre todo, por su paciencia.

El otro pilar de esta tesis lo representa mi otro director, el Dr. Manuel Felipe Cátedra. A él debo agradecerle especialmente la favorable disposición que ha mostrado siempre hacia este trabajo, y su implicación personal y desinteresada en él. Todo ello, por supuesto, sin menoscabo de la generosa aportación de sus conocimientos, en particular en el campo del electromagnetismo computacional, que suponen una parte tan importante del mismo.

Me gustaría también mostrar mi gratitud y reconocimiento al Dr. Eliseo García, una fuente inagotable de conocimientos, consejos, explicaciones y ayuda en general, a lo largo de todo este periodo de trabajo. Sería justo decir que, sin él, nada de ésto habría sido posible, y estoy seguro de que, con la conclusión de esta tesis, él finalmente descansará casi tanto como yo.

También he de recordar en este punto a las personas que han contribuido, con sus aportaciones, a mejorar este trabajo, revisando parte o la totalidad del manuscrito, y sugiriendo incontables mejoras del mismo: el Dr. Juan Francisco Jiménez Castellanos, el Dr. Raúl Durán Díaz y el Dr. José María Girón Sierra. Como se suele decir, cualquier defecto o debilidad que presente esta memoria es responsabilidad mía, y lo hace a pesar de sus buenos consejos.

En un ámbito más personal, no hay palabras para agradecer su apoyo incondicional y su paciencia al aceptar prescindir mí durante largas

temporadas, a mi mujer. Gracias, Merche, por alentarme, por confiar, por apoyarme, por reorientarme en ocasiones, y por recordarme, en los momentos difíciles, que el destino merecía la pena.

También quiero agradecer su apoyo y motivación al resto de mi familia: a mis padres, Carmen y Rafael, a quienes este logro seguramente haga más ilusión incluso que a mí mismo. Mi padre ha sido, sin duda ninguna, la persona que más ha influido para que decidiera embarcarme en esta aventura. Y a mis hermanos, Javi y Pablo. Y también a Ana, a Óscar, a Silvia, a Antonio, amigos que, con su interés personal, me han ayudado a continuar y finalmente terminar todo esto, aunque tal vez no hayan sido conscientes de ello.

Por último, pero no por ello menos importante, me gustaría recordar a mis compañeros de departamento. Es inevitable dejarse a algunos, ya que la lista es larga, y pido disculpas de antemano por evitar incluir un listado de todo el personal del mismo, pero me gustaría dar las gracias, amén de los que ya han aparecido más arriba, especialmente a Julia Clemente, a Javier de Pedro, a Álvaro Perales, a Rafa Rico, a José Gallego, a José Miguel Ruiz, a Virginia Escuder, a Rosa Estriégana, a Juani López, a Agustín Martínez, a Carmen Hernández y a Carmen de Luis. A todos, un abrazo, y nos seguiremos encontrando.

Resumen

En esta tesis se aborda el proceso de transformación de un método numérico, el Método de las Funciones Base Características, para que pueda ser ejecutado en una tarjeta gráfica que acepte la tecnología CUDA. Las tarjetas gráficas tienen una forma de ejecutar los programas radicalmente diferente a los computadores tradicionales, lo que exige grandes modificaciones en los algoritmos al ser trasladados de éstos últimos a ellas. Sin embargo, también pueden ofrecer una importante ganancia en el rendimiento de determinadas aplicaciones, lo que justifica el esfuerzo en la transformación.

El Método de las Funciones Base Características es un método numérico para el análisis electromagnético de cuerpos. En la implementación que se utiliza en esta tesis, se emplea para reducir la matriz de acoplos resultante de la aplicación de otro método de análisis, el Método de los Momentos, por lo que el primer paso es la adaptación a la tarjeta gráfica de este último. El algoritmo del Método de los Momentos, por sus características, se presta muy bien a la ejecución en una tarjeta gráfica. Sin embargo, es preciso estudiar cuidadosamente los detalles de la transformación para obtener la máxima mejora.

Una vez obtenida la matriz de acoplos, se le aplica el Método de las Funciones Base Características para reducirla, proceso que también debe ser adaptado a la tarjeta gráfica. Dado que el análisis electromagnético de un cuerpo medianamente grande suele exigir gran cantidad de recursos de computación, fundamentalmente de memoria, el análisis se suele hacer por regiones en lugar de aplicarse al cuerpo completo. Por lo tanto, se debe estudiar la influencia de esta partición en la ejecución del código.

Por último, se estudia cómo se puede aprovechar la presencia en el computador de varias tarjetas gráficas. Se analizan factores como la facilidad de la programación, o la gran influencia que puede tener en el rendimiento de la aplicación la política de reparto de trabajo.

Tabla de Contenido

Tabla de Figuras

Tablas

1 Introducción.

En la ingeniería actual los métodos numéricos ocupan un lugar fundamental. Prácticamente todos los problemas de ingeniería requieren de uno o varios métodos numéricos para poder ser resueltos. En ocasiones, estos métodos recurren a simplificaciones o aproximaciones para relajar las exigencias en cuanto a recursos de computación, sean éstos medidos en términos de memoria, séanlo en términos de potencia de cálculo. Al aplicar aproximaciones, los métodos pueden dejar de ser generales, de tal forma que sólo son útiles cuando concurren distintas condiciones en los objetos de estudio. Generalmente se cumple que, cuanto más general es un método, más exigente es con respecto a los recursos de computación, y a la inversa, métodos muy eficientes a menudo sólo son aplicables a objetos de estudio con características particulares.

Uno de los motores de avance en la aplicación de los métodos numéricos más generales es la constante mejora en la ingeniería de computación. La aparición de nuevos dispositivos con más capacidad de almacenamiento y mayor potencia de cálculo es un factor que hace que métodos que hasta ese momento no eran viables por sus requisitos pasen a poder ser aplicados. También ocurre que, en el momento en el que un nuevo dispositivo más potente se hace disponible, inmediatamente ve sus recursos saturados, ya que se le asignan todas esas tareas que hasta ese momento no se podían realizar en otros dispositivos previamente disponibles.

En esta espiral de crecientes capacidad de computación y demanda de la misma se sitúa la presente tesis. Se trata de adaptar un método numérico de ingeniería, en este caso de ingeniería electromagnética, a un (relativamente) nuevo dispositivo computacional, la GPU (‘*Graphics Processing Unit’,* Unidad de Procesamiento para Gráficos). La GPU es una adaptación de una tarjeta gráfica tradicional a la computación de altas prestaciones, y, en este trabajo, representa el dispositivo de reciente aparición que permite mejorar el rendimiento de métodos numéricos exigentes en cuanto a potencia de computación. La propuesta es investigar la capacidad de la nueva arquitectura para tratar un método de ingeniería electromagnética, el Método de las Funciones Base Características (CBFM, ‘*Characteristic Basis Function Method*’).

1.1 Antecedentes.

En el campo del análisis electromagnético de geometrías tridimensionales arbitrarias existe una gran variedad de métodos que se aplican en la actualidad, cada uno con características distintas en cuanto a la precisión de los resultados que proporcionan, el tipo de objeto para el que esa precisión es aceptable, y el tiempo de cálculo que requieren para producir esos resultados. Una posible clasificación inicial de estos métodos daría como resultado dos grandes grupos: los de *alta frecuencia* y los de *baja frecuencia*. El primero de ellos se emplea cuando el tamaño del objeto es muy superior a la longitud de onda de la radiación electromagnética que se considera. En estos casos, la corriente en cada punto del objeto bajo estudio depende únicamente de la corriente presente en una región del objeto cercana a él, es decir, tiene carácter local. El grupo de métodos de baja

frecuencia se utiliza cuando ocurre todo lo contrario, es decir, cuando la corriente en cada punto del objeto bajo estudio no tiene carácter local, sino que depende de la corriente presente en todo el resto de la geometría del objeto.

Algunos de los métodos de **alta frecuencia** más importantes son:

- El **Método de Óptica Geométrica** [1], en el que se supone que la energía se propaga en forma de rayos, y cumple el principio de Fermat. La reflexión y la refracción están descritos por la ley de Snell.
- El **Método de Óptica Física** [2], que se aproxima más a un tratamiento electromagnético de la radiación que el Método de Óptica Geométrica, aunque no llega a ser un tratamiento rigurosamente electromagnético. Por medio de rayos, se estima el campo en el objeto, y se utiliza ese campo para obtener el campo dispersado por el mismo.
- El **Método de la Teoría Geométrica de la Difracción** [3], que es una extensión del Método de Óptica Geométrica en el que se añaden al estudio los rayos producto de la difracción.
- El **Método de la Teoría Física de la Difracción** [4], que, a su vez, es una extensión del Método de Óptica Física que consiste en tener en cuenta para el análisis una serie de corrientes adicionales que dan cuenta de las discontinuidades del cuerpo.

Los métodos de alta frecuencia aplican algún tipo de aproximación, aprovechando que los efectos de la radiación son locales. Las aproximaciones son tanto más precisas cuanto mayor es el tamaño del objeto con respecto a la radiación que se considera. Sin

embargo, en ocasiones el problema exige una determinada precisión, por lo que es necesario prescindir de las aproximaciones y aplicar métodos más rigurosos. Entonces es preciso utilizar los **métodos de baja frecuencia**, en los que se aplican de forma directa las ecuaciones de Maxwell. En función de la formulación de las ecuaciones de Maxwell que se utilice, este grupo se subdivide entre **métodos integrales** y **métodos diferenciales**:

- Entre los **integrales** se encuentra el **Método de los Momentos** (MoM, '*Method of Moments*') [5]. El Método de los Momentos es un nombre genérico para métodos en los que una ecuación funcional se transforma en una ecuación matricial. Aplicado a electromagnetismo, el método convierte las ecuaciones de Maxwell en su versión integral en una ecuación matricial. Una alternativa a este método es el **Método del Gradiente Conjugado con la Transformada Rápida de Fourier** [6], en el que se usa el Método del Gradiente conjugado para resolver iterativamente las ecuaciones integrales de Maxwell, y la Transformada Rápida de Fourier para ejecutar de forma más rápida cada iteración.
- Los métodos **diferenciales** utilizan una malla de puntos que contiene al objeto que se analiza. Aplicando unas condiciones de contorno determinadas, los métodos buscan obtener el valor de los campos **E** y **H** en cada una de las celdas resultantes a la aplicación de la malla. Ejemplos de métodos diferenciales son el **Método de los Elementos Finitos** [7] y el **Método de Diferencias Finitas en el Dominio del Tiempo** [8].

Para el trabajo que se va a desarrollar en esta tesis se ha seleccionado el Método de los Momentos por dos razones principalmente: por un lado es uno de los métodos más generales con respecto al tipo de objeto sobre el que se practica el análisis. Por otro, dado que es un método riguroso, proporciona resultados con una gran precisión.

Estas dos características positivas tienen asociada una parte negativa: el método es muy exigente en términos de computación. Es cierto que el método transforma un complejo sistema de ecuaciones integro-diferenciales en un sistema de ecuaciones lineales. Sin embargo, a pesar de esta simplificación, ese conjunto de ecuaciones se hace muy grande de forma relativamente rápida, lo que pronto hace difícil o imposible tratar el problema por medios computacionales.

Existen varias soluciones a esta dificultad. La primera de ellas es el **Método Rápido de los Multipolos (*'Fast Multipole Method'*, FMM)** [9]. El método consiste en desarrollar la función de Green, que está en el núcleo de los cálculos del MoM, por medio de una expansión en multipolos, lo que permite agrupar fuentes cercanas y tratarlas como una única fuente. De esta manera, se puede reducir la complejidad del producto matiz-vector, que es la parte más exigente en cálculo de los métodos de resolución iterativos para los sistemas de ecuaciones lineales resultado de la aplicación del MoM. El FMM a menudo se aplica de forma jerárquica, de modo que la agrupación de fuentes se realiza en una serie de niveles, y a cada uno de ellos se le aplica el FMM. Es lo que se denomina el **FMM Multinivel ('Multilevel Fast Multipole Method', MLFMM)** [10].

Otra forma de poder seguir utilizando el MoM en estos casos es la que se ha elegido para esta tesis, y consiste en aplicar una técnica que reduzca el número de ecuaciones a resolver. Esto se consigue con el **Método de las Funciones Base Características (CBFM, '*Characteristic Basis Function Method*')** [11]. Este método reduce el tamaño del sistema de ecuaciones lineales que proporciona el Método de los Momentos, y la forma de hacerlo es expresar dichas ecuaciones en una base algebraica diferente, en la que sólo un subconjunto de las dimensiones tiene un peso perceptible en el sistema. En el caso del sistema de ecuaciones original, las corrientes inducidas (los valores que se pretende obtener) se expresan en función de una discretización de la geometría, es decir, tienen una base geométrica. En el Método de las Funciones Base Características, las corrientes se expresan en función de una serie de macro-funciones de alto nivel. Sólo unas pocas de entre todas estas macro-funciones proporcionan información relevante para el problema, con lo que expresando las corrientes inducidas en la geometría en función de esas pocas, el tamaño del sistema queda muy reducido. De esta manera, es posible resolver el sistema de forma directa.

1.2 Motivaciones.

1.2.1 Análisis electromagnético.

El análisis electromagnético se puede aplicar con distintos fines: análisis de circuitos, estudio de diagramas de antenas, análisis de guías de onda, cálculo de la sección radar, etc. Cada uno de ellos tiene a su disposición uno o varios de los métodos expuestos en el apartado anterior (y algunos más, ya que se trata de un breve

resumen). La presente tesis se enmarca dentro del problema de la obtención de la sección radar (RCS, *'Radar Cross-Section'*) de un conductor eléctrico perfecto (PEC, *'Perfect Electrical Conductor'*).

La RCS es una medida de la detectabilidad de un objeto (el PEC) frente a un radar. La Figura 1-1 muestra la situación. Al ser alcanzado por la radiación electromagnética que emite el radar, el objeto (el avión) refleja parte de esa energía. Cuánta y en qué direcciones depende de factores como su composición, su geometría, el ángulo de incidencia de la radiación, etc. Una forma de obtener la radiación reflejada en una dirección determinada es aplicar el método de los momentos para calcular las corrientes que la radiación incidente genera en la superficie del objeto (si es un PEC), y, a partir de ellas, obtener la radiación reflejada. Aplicando el MoM, las corrientes se pueden conseguir resolviendo un sistema de ecuaciones lineales, que es la parte más costosa en tiempo de cálculo del procedimiento. Todo esto se describe en más detalle en el capítulo 3.

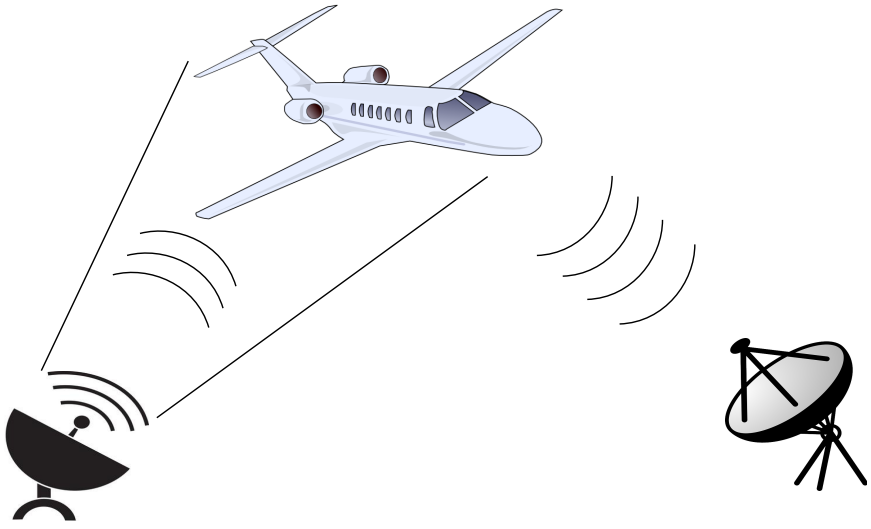


Figura 1-1. Radiación de un radar reflejada por un avión.

El Método de las Funciones Base Características (CBFM) permite reducir el sistema de ecuaciones, lo que hace más eficiente el proceso de obtención de la sección radar. A modo de ejemplo, la Tabla 1-1 muestra los tiempos, en segundos, para procesar un objeto y obtener su RCS cuando se usa únicamente el MoM y cuando se usa éste en combinación con el CBFM. El objeto en cuestión es un modelo de un avión, con 700.000 subdominios. El motivo de incluir el ejemplo es dar una idea del ahorro que puede suponer el CBFM al calcular la RCS, y qué peso relativo tienen las distintas fases del proceso entre sí.

Tabla 1-1. Tiempos de las fases de la RCS

Fase	**MoM**	**CBFM**
Preproceso	81	42
Obtención de la matriz	6747	16142
Resolución sistema	5212794	1594972
Cálculo RCS	900	38

Como se puede observar, la parte más importante es la resolución del sistema. El CBFM permite reducir este tiempo considerablemente (la reducción es del orden de 3.2, es decir, se tarda 3.2 veces menos con él que sin él). A pesar de ser un simple ejemplo, es representativo, y sirve para justificar la orientación de la tesis: acelerar la aplicación del CBFM.

1.2.2 Utilización de GPUs.

El CBFM permite abordar el estudio de una geometría y realizarlo en tiempos aceptables. Sin embargo, el proceso de reducción sigue siendo costoso en términos computacionales. Para acelerarlo, en esta tesis doctoral se analiza la posibilidad de realizarlo por medio de tarjetas gráficas o GPUs. Las tarjetas gráficas son unos dispositivos computacionales tradicionalmente concebidos para la generación y el proceso de gráficos por computador. Sin embargo, en los últimos tiempos han experimentado una reorientación en cuanto a su uso.

Por su misión original, las GPUs están diseñadas para procesar una gran cantidad de primitivas de gráficos. Esto implica, fundamentalmente, transformaciones geométricas, que, en esencia, son operaciones algebraicas. Siendo esto así, es posible utilizar su diseño para ejecutar ese mismo tipo de operaciones algebraicas cuando los objetos sobre los que trabajan no son gráficos, sino datos de un problema matemático diferente: las tareas que surgen de aplicar, a una geometría arbitraria, primeramente el Método de los Momentos, y a continuación, el Método de las Funciones Base Características.

De esta manera, la primera motivación de este trabajo es la reducción del tiempo de cómputo del método. Otras formas de paralelización, que utilizan otro tipo de dispositivos y tecnologías, como MPI ('*Message Passing Interface*', Interfaz para paso de mensajes) u OpenMP (*'Open Multi-Processing'*, Multiproceso "abierto") se han aplicado ya al mismo [12], pero los resultados publicados por diversos grupos de trabajo que utilizan las tarjetas gráficas para distintos problemas de cálculo científico [13], [14], [15] hacen prever una mejora de la eficiencia considerable.

Una segunda motivación es ampliar el rango de geometrías tratables. El Método de las Funciones Base Características se ha demostrado eficaz en la reducción del problema del estudio del comportamiento electromagnético de un objeto. Utilizando las GPUs, se pretende estudiar la posibilidad de aplicarlo a objetos aún mayores. Se trata de averiguar hasta dónde se puede llevar el método si se utiliza una GPU, es decir, de encontrar su nuevo límite con un dispositivo como este.

1.3 Contexto.

Con la reciente aparición de computadores de alto rendimiento con un coste económico razonablemente bajo se ha aplicado gran esfuerzo en crear versiones paralelas de los métodos numéricos expuestos en el apartado anterior utilizando distintos tipos de hardware [16], [17], [18]. Existen versiones paralelas para prácticamente todos los métodos numéricos mencionados en el apartado 1.1. En este apartado se va a hacer una pequeña exposición de los trabajos más interesantes relacionados con la paralelización del Método de los Momentos (MoM) y algunos métodos relacionados, fundamentalmente el Método Rápido de los Multipolos (FMM) y, obviamente, el Método de las Funciones Base Características (CBFM).

1.3.1 Paralelización tradicional.

Como es natural, desde que existen las máquinas paralelas, existen versiones paralelas de prácticamente todos los métodos que se utilizan en este campo. Por ejemplo, en el año 1997 Walker y Leung [19] realizaron una versión paralela del método de la ecuación

integral en el dominio del tiempo sobre un Cray T3D con 256 procesadores. La aceleración obtenida llega a 15, lo cual es un valor considerable, ya que para las mediciones que realizan no utilizan los 256 procesadores. El número máximo de procesadores utilizados es 16.

Sin embargo, hay resultados más recientes que utilizan técnicas más actuales. Por ejemplo, en [20] se realiza una aceleración del Método Rápido de los Multipolos Multinivel (MFMM) utilizando una máquina con 16 procesadores, paralelizando el código con MPI. La eficiencia (el porcentaje de utilización de los procesadores) es razonablemente buena y se acerca al 60% con 16 procesadores. Según los autores, los resultados pueden mejorarse optimizando la utilización de recursos y con algún ajuste fino de los parámetros del sistema. En una línea parecida se orienta [21]. También se realiza una paralelización sobre MPI del Método Rápido de los Multipolos. En este caso el rendimiento lo miden como aceleración de la versión paralela, e informan de un valor de hasta 8 utilizando 12 procesadores.

Los métodos se pueden hibridizar, y de ello es un ejemplo [22]. Se trata de un método que combina el Método de los Momentos con el Método Rápido de los Multipolos y el Método de Integración por el Camino de Mayor Pendiente (*'Steepest Descent Integration Path Method'*). La versión paralela se ejecuta en un cluster Beowulf de 32 procesadores sobre MPI, y se obtiene una aceleración de 7.2, en la línea de trabajos similares.

Centrándonos más en el MoM, en [23], Cwik et al. abordan la paralelización de este método en un computador Mark IIfp Hypercube [24]. De los 128 procesadores disponibles utilizan un

máximo de 64. Dado que es un multiprocesador de memoria privada, el reparto de los datos a procesadores es un punto fundamental. Los autores argumentan que la parte en la que se pierde la eficiencia es la factorización de la matriz de impedancias, ya que el pivotamiento obliga a aumentar las comunicaciones y la eficiencia baja. De hecho, según sus datos, una vez paralelizado el proceso, la factorización supone el factor dominante en el tiempo total de ejecución del método. La solución del sistema también es difícil de paralelizar, pero ya en el algoritmo secuencial tiene un peso menor sobre el total del programa. La ganancia obtenida llega a 2 en el caso de unas 1500 incógnitas.

Por su parte, Kaklamani et al. [25] hacen una comparación de diferentes tipos de computadores sobre los que ejecutar el MoM. Utilizan dos máquinas de memoria compartida, un Silicon Graphics Power Challenge y un Cray C90, así como dos máquinas de memoria distribuida privada, un Intel Paragon y un Cray T3D. No dan demasiados detalles sobre la ejecución del código (qué partes se benefician más de qué características, etc), pero el resultado general parece ser que, en cuanto a eficiencia el computador Silicon Graphics y el Cray T3D son los que mejor resultado dan. Los autores se decantan por éste último basándose en que, siendo modularmente extensible, se puede adaptar mejor a problemas más grandes.

1.3.2 Paralelización por medio de GPUs.

El apartado anterior se ha incluido para mostrar el interés por la paralelización de los métodos numéricos utilizados en electromagnetismo desde los principios de la computación paralela. Pero, dado que esta tesis se centra en el uso de GPUs, y más

concretamente, en el uso de CUDA, en esta sección se agrupan los resultados obtenidos en la bibliografía en este tipo de paralelización, centrándose especialmente en los algoritmos relacionados con los métodos numéricos para el análisis electromagnético.

CUDA ha supuesto una revolución en la programación de métodos numéricos. Generalmente se requiere reformar la aplicación de forma casi completa, pero en ocasiones es posible aprovechar resultados obtenidos en arquitecturas *multi-core* tradicionales. En [26] se hace precisamente esto: un método numérico, la factorización QR, que estaba adaptado a una arquitectura *multi-core*, es modificado para, además, utilizar GPUs. El método original realizaba una partición del trabajo, y lo que se propone es simplemente transformar algunas de esas particiones para que corran en GPUs, manteniendo las demás. De esta manera, es posible aprovechar una arquitectura mixta con múltiples CPUs y múltiples GPUs.

Sin embargo, para aprovechar al máximo los recursos de la GPU, generalmente es preciso reformar en gran medida la apliación. Un ejemplo de esto es [27]. Este artículo se separa de la corriente tradicional aconsejada por NVIDIA de generar tantos hilos de trabajo como requiera el trabajo en función de sus cálculos (generalmente del orden de millones de ellos), de forma que la planificación del trabajo la haga la propia capa física de la tarjeta. La propuesta es hacer únicamente un grupo de hilos que, a diferencia de lo propuesto por NVIDA, cuando terminan con una porción de trabajo, en lugar de terminar, toman una nueva porción. Son los hilos persistentes, y lo que se consigue es un reparto dinámico del trabajo, en el sentido de que los propios hilos van tomando más trabajo a medida que terminan la porción anterior. Lo que se ahorra es la creación de hilos

nuevos. Los resultados presentados son prometedores, aunque, a medida que se vayan desarrollando nuevas tarjetas, con nuevas capacidades, será preciso analizar si este planteamiento sigue siendo practicable, habida cuenta que no sigue la filosofía de trabajo propuesta por NVIDIA.

Utilización de varias GPUs.

Una de las líneas de trabajo más apasionantes, y que en esta tesis se explora de forma únicamente tentativa, es el aprovechamiento de sistemas con varias GPUs. La cuestión que se plantea cuando se utilizan varias GPUs es cómo realizar el control de las mismas. Si lo hace el programa principal de forma secuencial, la complejidad del mismo aumenta considerablemente, y si no se hace un diseño cuidadoso, se puede perder un tiempo precioso en esperas que puede compensar negativamente el aumento de recursos del sistema. Una alternativa a esta solución es utilizar alguno de los entornos de programación paralela existentes, ya sea por paso de mensajes, como MPI, ya sea con alguno de los entornos de programación multi-hilo. Existe en la bibliografía una gran cantidad de trabajos que documentan la utilización de MPI para acelerar, por medio de múltiples GPUs, la resolución de algunos problemas científicos. Algunos de ellos, además de trasladar un método numérico a la nueva arquitectura, realizan algún tipo de análisis que puede servir en otros casos, lo que los hace más interesantes. Por ejemplo, [28] presenta una serie de modelos matemáticos de los distintos componentes de un sistema multi-GPU con los que realizan predicciones con respecto al rendimiento que se puede obtener del mismo. El objetivo es dotarse de herramientas para realizar una asignación de tareas eficiente. Los

resultados obtenidos son muy precisos en los casos comprobados empíricamente. Sin embargo, los sistemas reales que evalúan constan de relativamente pocas GPUs (hasta 4), lo que limita la generalidad de sus conclusiones.

En esta línea de estudio teórico, también es interesante [29]. En este caso, el énfasis se pone sobre el sistema de memoria de la GPU. Se trabaja sobre un sistema dotado de un único nodo con hasta 7 GPUs, y se analiza el rendimiento de las transferencias en el bus PCI y su efecto en la eficiencia global de la aplicación, que realiza simulaciones de propagación de ultrasonidos. Se realizan distintas particiones de los datos, y se prueban distintas formas de distribuirlos entre las memorias del sistema. También se consideran distintos esquemas de reparto de trabajo entre las GPUs. El resultado es un conjunto de recomendaciones a la hora de repartir trabajo y datos en un sistema con varias GPUs, dependiendo de la cantidad de veces que se accede a los datos y de qué proceso los haga. El estudio se hace sobre un sistema específico y no demasiado extendido, por lo que estos resultados pueden no ser válidos para todo tipo de sistemas.

La cuestión de los accesos a memoria es una de las más importantes en el campo de las GPUs. Dada la extraordinaria potencia de cálculo de la que disponen, "devoran" los datos a una velocidad de vértigo. El desfase entre la velocidad de proceso y la de acceso a datos es un clásico en Arquitectura de Computadores, pero en este campo específico, juega un papel fundamental. Las técnicas que consiguen enmascarar los tiempos de acceso a datos tienen una gran importancia. En [30], por ejemplo, se presentan una serie de extensiones a XKaapi que permiten explotar toda la potencia de ejecución de un entorno multi-GPU. XKaapi [31] es una capa

software creada pensando en sistemas multiprocesador heterogéneos que permite, en tiempo de ejecución, realizar la planificación de las tareas de computación. El artículo extiende la funcionalidad básica para buscar el máximo solapamiento posible entre computación y acceso a datos. Se muestran los resultados sobre dos problemas clásicos de álgebra lineal: el producto de matrices y la factorización Cholesky. Con 8 GPUs obtienen una aceleración de hasta 6.74, lo cual es un dato muy positivo, especialmente teniendo en cuenta que trabajan con matrices que no caben completas en la memoria de la GPU, lo que obliga a un mayor número de accesos a memoria del sistema.

Otra forma de atacar el mismo problema, el de la latencia de los accesos a memoria, es el que plantea [32]. Con el Toolkit 4.1, NVIDIA ha proporcionado la posibilidad de realizar transferencias de GPU a GPU. Anteriormente estas transferencias obligaban a realizar dos pasos: uno primero de la primera GPU a memoria del sistema, seguido de un segundo de memoria del sistema a la segunda GPU. En el artículo se presenta un mecanismo de comunicación entre GPUs basado en MPI que hace uso de las transferencias GPU a GPU proporcionadas por el Toolkit 4.1 y que mejora la latencia entre un 70% y un 80%. La aplicación experimenta una mejora del 16% en el caso concreto expuesto.

Aplicación a métodos de análisis electromagnético.

Los trabajos anteriores abordan cuestiones generales de optimización del uso de GPUS. Algunas de estas técnicas se utilizan específicamente para los métodos numéricos relacionados con el análisis electromagnético. Por centrarnos en los más relacionados con

el contenido de esta tesis, se relacionan a continuación una serie de trabajos encaminados a paralelizar, por medio de GPUs, el FMM y el MoM.

El FMM es un algoritmo difícil de paralelizar, fundamentalmente porque parte del trabajo consiste en crear y mantener una estructura de datos de tipo árbol. En [33], Darve *et al.* Realizan un estudio de la paralelización de este algoritmo, utilizando como objetivos tanto arquitecturas CPU *multicore* como GPUs. Con un cuidadoso diseño del algoritmo, consiguen ganancias de hasta 7, lo cual da una idea da la dificultad de la tarea (un valor de 7 para una GPU es relativamente bajo). El objetivo en cualquier algoritmo implementado sobre una GPU es maximizar el número de operaciones aritméticas por cada palabra de memoria leída, dado que generalmente en una GPU la parte débil es el acceso a memoria. Esto lo consiguen con una serie de reordenaciones de los bucles de los algoritmos, con lo que consiguen agrupar los cálculos sobre los mismos datos.

Redundando en la dificultad que este método plantea a las GPUs, Chandramowlishwaran *et al.* [34] plantean un desarrollo algorítmico con el que consiguen que algunas arquitecturas *multicore*, fundamentalmente basadas en la familia x86 ofrezcan el mismo rendimiento que una GPU. Se centran en un único nodo de computación con dos CPUs de 4 *cores* cada una, y por medio de optimizaciones como ajuste fino o reorganización de las estructuras de datos (por ejemplo, arrays de elementos de tipo `struct` en C se transforman en colecciones de arrays de elementos individuales, cada uno de ellos para uno de los componentes de los `struct`) , consiguen mejorar considerablemente el rendimiento del programa. Además, hacen uso intensivo de las instrucciones de tipo SIMD presentes en

las CPUs que, aunque no tan potentes en cuanto al número de datos procesados en cada una como las análogas en las GPUs, consiguen una aceleración a tener en cuenta. Estas instrucciones tienen que insertarlas a mano, ya que el compilador no es capaz de detectar la posibilidad de su uso. Lo aplican a una versión general del FMM, ya que no es un método exclusivo para electromagnetismo, pero las conclusiones obtenidas son igualmente interesantes.

Por contrarrestar esta visión negativa del uso de GPUs para el algoritmo FMM, merece la pena incluir en este apartado el trabajo de Hu *et al.* [35]. Su objetivo es crear un algoritmo heterogéneo, destinado a ejecutarse en un sistema basado en nodos con una CPU y 2 GPUs. El número de nodos puede variar entre 2 y 32. Dada la dificultad de la paralelización de este método, buscan una asignación adecuada de trabajo tanto a la CPU como a las GPUs, de forma que cada uno de estos elementos haga aquello para lo que está más preparado. De esta manera, a la CPU le corresponde la gestión de los grupos de elementos (*boxes*), mientras que las GPUs se encargan de procesar la información relacionada con los elementos individuales. Su versión del FMM se puede aplicar a campos tan diferentes como la dinámica molecular, la astrofísica o la dinámica de fluidos. Medido en Mflops/dollar, su versión del FMM rinde más del doble que la máquina más eficiente hasta ese momento [36].

Por su parte, el MoM tiene un tratamiento algo más sencillo. La mayor parte de los trabajos publicados relativos a esta técnica se orientan a acelerar la creación de la matriz de impedancias, que es la parte principal del MoM. Un ejemplo es [37]. En este trabajo los autores consiguen una aceleración de 70 comparada con la ejecución en la CPU (es decir, la versión propuesta es 70 veces más rápida que

la versión de referencia). Los mismos autores en [18] hablan de una aceleración de hasta 140 para la misma tarea. Sin embargo, a continuación, el método requiere realizar una descomposición LU de la matriz, lo cual no se beneficia demasiado de la GPU, por lo que la aceleración global se queda en 45.

Los trabajos anteriores tratan la matriz de impedancias completa de una sola vez, estando así limitados por la cantidad de memoria de que disponga la tarjeta. Otra línea de trabajo dentro de este mismo campo se aplica en optimizar el algoritmo para que pueda superar este tipo de limitaciones impuestas por el hardware de la GPU. Por ejemplo, en [38] se propone un método mediante el cual la matriz de impedancias se divide en sub-matrices, y se procesa cada una de ellas de forma secuencial. La ganancia global es de 30, lo cual es muy interesante, ya que el propio hecho de dividir la matriz acarrea una pérdida consustancial en rendimiento, es decir, se está sacrificando rendimiento con el objetivo de poder procesar un objeto tan grande que de otro modo resultaría imposible de analizar. Bajo esta luz, la cifra de ganancia tiene una mayor relevancia. Un trabajo en la misma línea es [39]. En [40], Kiss *et al*, intentan evitar cálculos innecesarios obteniendo cada valor una única vez y reutilizándolo cuando sea necesario. Un gran ejemplo del ajuste fino en la adaptación del algoritmo a la máquina (la GPU) sobre la que corre es el trabajo de Topa *et al.* [41]. En él se analizan los recursos de la GPU para realizar un reparto de las tareas de cálculo entre sus unidades de computación para obtener el máximo de ganancia.

Todos estos trabajos guardan relación con el propuesto para esta tesis, ya que su objetivo también es realizar un análisis electromagnético de un objeto. Sin embargo, algunos detalles básicos

de la implementación de los algoritmos son diferentes, detalles tales como la representación de la geometría del objeto, el método de cálculo numérico utilizado para calcular las integrales necesarias, el criterio para la partición del objeto en bloques para su posterior procesado, etc. Otra diferencia son los métodos utilizados para el análisis. En nuestro caso se utilizará el CBFM que, hasta donde hemos podido comprobar, hasta la fecha, no ha sido adaptado a una GPU.

1.4 Objetivos.

El principal objetivo de esta tesis es implementar el Método de las Funciones Base Características en tarjetas GPU de NVIDIA por medio de la tecnología CUDA. Se pretende:

- Analizar la viabilidad de la implementación del algoritmo del CBFM cuando se ejecuta en una GPU.
- Analizar el efecto de diversas opciones de implementación en cuanto a tiempo de ejecución y a cantidad de memoria requerida.
- Obtener criterios generales y recomendaciones de implementación en GPUs que sean aplicables a métodos numéricos de cálculo científico.
- Analizar la viabilidad de las implementaciones del algoritmo cuando la carga de trabajo se reparte entre varias GPUs.

Para alcanzar estos objetivos será necesario cumplir una serie de objetivos particulares:

- Adaptar el algoritmo del Método de los Momentos al lenguaje de programación C y al entorno CUDA.

- Transformar el algoritmo del MoM para su ejecución en una GPU.
- Analizar las configuraciones más eficientes del MoM, tanto en tiempo de ejecución como en memoria requerida.
- Extraer criterios generales para el diseño de programas de cálculo científico con respecto a su ejecución en una GPU.
- Adaptar el algoritmo del Método de las Funciones Base Características a su ejecución en una GPU.
- Analizar las distintas formas de construcción de un programa para realizar la gestión de varias GPUs.
- Adaptar el algoritmo del CBFM para su ejecución en varias GPUs.

La principal dificultad del trabajo es la adaptación del método a un paradigma de computación completamente diferente, como es el de una GPU. En este sentido, el código ya existente tiene una utilidad limitada: sólo se puede mantener la parte que no se traslade a la GPU, es decir, la inicialización de las estructuras de datos de la geometría bajo análisis. Todo el algoritmo que aplica el Método de los Momentos, y trabaja sobre la matriz de acoplos para reducirla con el Método de las Funciones Base Características debe reescribirse completamente. En la práctica, es como crear una aplicación nueva prácticamente desde cero.

1.5 Estructura.

La presente memoria se ha organizado en ocho capítulos, incluyendo éste mismo. A continuación se describen los contenidos de cada uno.

En el Capítulo 2 se describe el funcionamiento de una GPU. Se presenta de forma somera su historia, y se entra en más detalle a la hora de exponer sus componentes, modo de funcionamiento y la forma en la que se utilizan, así como los requisitos que ésta impone al código que se pretenda ejecutar en ella.

El Capítulo 3 presenta los fundamentos matemáticos de la tesis. Se describen el Método de los Momentos y el Método de las Funciones Base Características, partiendo de las ecuaciones de Maxwell y explicando las técnicas de discretización que permiten transformarlas en ecuaciones lineales. Por esta razón se incluye un apartado que describe las distintas formas de representar matemáticamente una geometría.

En el Capítulo 4 se describe la adaptación a CUDA del Método de los Momentos. Además, se estudia cómo afecta el diseño de la aplicación al aprovechamiento de los recursos de la tarjeta gráfica. El Capítulo 5 hace algo análogo con el Método de las Funciones Base Características. En este caso no ha sido necesario un desarrollo tan largo, por lo que el estudio es más corto.

El Capítulo 6 se dedica a un esfuerzo adicional de paralelización en el cual se utilizan varias GPUs. Se describe el paradigma OpenMP, que ha sido necesario utilizar para controlar de forma sencilla y eficaz las distintas GPUs del sistema, y, a continuación, se describen los esfuerzos realizados en esta línea de trabajo.

Los resultados de todas las versiones creadas en los capítulos 4, 5 y 6 se muestran en el Capítulo 7. Se empieza comprobando la corrección de las versiones desarrolladas, y a continuación se pasa a analizar su comportamiento y las razones por las que se deben

introducir las modificaciones que dan lugar a las versiones posteriores.

Por último, el Capítulo 8 presenta las conclusiones que se pueden sacar del trabajo desarrollado, las aportaciones originales de esta tesis, y posibles líneas de trabajo para el futuro.

2 Computación con Procesadores Gráficos.

2.1 Introducción.

2.1.1 Evolución histórica.

Uso tradicional de las tarjetas gráficas.

Una de las razones más importantes para la rápida evolución en los últimos tiempos de las tarjetas gráficas en los computadores personales ha sido la fuerte demanda de gráficos de alta calidad en tiempo real, fundamentalmente en el campo de los videojuegos. Los gráficos por computador ya eran una disciplina establecida antes de la aparición del computador personal, pero estaba restringida a estaciones de trabajo de gran potencia y precios considerables. Esta restricción, a su vez, hacía muy difícil el abaratamiento de los costes, lo que cerraba el círculo vicioso.

Con la entrada en escena del computador personal, los gráficos por computador empezaron a beneficiarse de una mayor demanda, aunque la calidad que se podía conseguir en un computador personal en estos primeros momentos no era comparable con la conseguida en las potentes estaciones de trabajo dedicadas a gráficos. Sin embargo, dicha demanda se convirtió (lo sigue siendo en la actualidad) en el motor que impulsó el desarrollo de tarjetas gráficas mucho más baratas y mucho más potentes cada vez. La economía de escala empezó a hacer rentable la inversión en el diseño y desarrollo de prototipos experimentales con mejores prestaciones, ya que esa inversión se amortizaba con las cada vez mayores cifras de ventas. Surgieron compañías dedicadas a proporcionar tarjetas gráficas

específicamente para los computadores personales, y la tendencia se ha ido haciendo cada vez más sólida hasta el punto actual, en el que las diferencias entre computadores personales y estaciones de trabajo se han difuminado considerablemente.

Los dispositivos de proceso de los gráficos por computador se han ido haciendo cada vez más sofisticados a lo largo de esta evolución. Inicialmente, un procesador gráfico era un dispositivo dividido en unas pocas etapas (ver Figura 2-1), completamente cableado (es decir, con toda la funcionalidad incluida en la capa física), aunque con ciertas posibilidades de configuración. El desarrollo que han experimentado estos dispositivos ha ido en la línea de una mayor complejidad y sofisticación, hasta llegar al punto en el que empiezan a ser interesantes para el cómputo científico de altas prestaciones: el momento en el que se hacen programables.

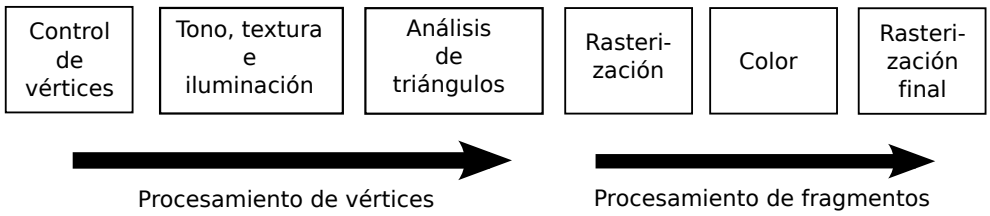


Figura 2-1. Etapas del procesamiento de gráficos.

Evolución de los procesadores gráficos programables.

A medida que los procesadores gráficos fueron acometiendo tareas más complejas, se fue haciendo necesario dotarlos de más flexibilidad. No existe una división en generaciones estricta, pero sí es posible ver la evolución que han ido experimentando hasta el momento actual. El primer paso en la programabilidad lo dio la tarjeta NVIDIA GeForce3, en el año 2001. El avance consistió en hacer accesible al

programador el motor de procesamiento de vértices. Internamente, este motor ya funcionaba por medio de algoritmos programados, por lo que el paso lógico era permitir al programador especificar dichos algoritmos con el repertorio de instrucciones interno del motor.

El siguiente paso corresponde a la tarjeta ATI Radeon 9700, que en el año 2002 permitía el acceso de una forma análoga al motor de procesamiento de fragmentos. Un nuevo avance fue la conclusión lógica de esta tendencia: en el año 2005 la consola XBox de Microsoft unificaba ambos motores, el de procesamiento de vértices y el de procesamiento de fragmentos, para presentar el primer procesador gráfico unificado. La idea se extendió, y en poco tiempo los demás fabricantes presentaban sus propias variantes de lo que se ha dado en llamar una GPU, del inglés *Graphics Processing Unit*. La GPU consta, en su visión más general, de una serie de procesadores de propósito más o menos general, valga la expresión, que se pueden dedicar tanto a procesar vértices como a procesar fragmentos, dependiendo de la necesidad del programador en cada momento. Lo más importante desde el punto de vista del trabajo presentado en esta tesis, es que esos procesadores son programables, y eso permite utilizarlos para tareas que no están limitadas a la generación de gráficos por computador.

Los inicios de la programación de GPUs.

A pesar de que con la aparición de las GPUs se abría el camino para la utilización de estos potentes coprocesadores para la computación científica, los inicios no fueron fáciles. Es cierto que las GPUs eran programables, pero también lo es que estaban diseñadas con el objetivo muy claro de proporcionar una potencia de cálculo

muy grande para el procesamiento de gráficos. Esta potencia se encauzaba por medio de fundamentalmente dos bibliotecas estándar de funciones: DirectX (o, más precisamente, su subconjunto para gráficos, Direct3D) y OpenGL. Estas bibliotecas contenían las funciones necesarias para poder generar los gráficos con facilidad, y eran la única forma práctica de acceder a los recursos de las GPUs. Como estas bibliotecas estaban pensadas exclusivamente para gráficos, los pioneros de la utilización de GPUs para cálculo científico tenían que convertir su problema en un problema gráfico, adaptando sus estructuras de datos y sus algoritmos a las que se usan en gráficos (texturas, colecciones de píxeles).

Sin embargo, no hizo falta mucho tiempo para que los fabricantes reconocieran el mercado potencial que tenían ante sí. Los primeros resultados publicados eran tremendamente prometedores, y en el mundo científico el interés por las tarjetas que los proporcionaban se fue haciendo patente cada vez más. Sólo hizo falta que fabricantes como NVIDIA o ATI proporcionaran un interfaz de programación más amigable al mundo científico para que éste se volcara sin reservas con él. En el caso de NVIDIA, este salto vino de la mano de CUDA (*Compute Unified Device Architecture*, Arquitectura de Dispositivo Unificada para Programación), que proporcionaba una plataforma de programación para aplicaciones de propósito general para sus tarjetas gráficas, y que ha sido la opción elegida para este trabajo. ATI, por su parte, lanzó inicialmente un producto parecido, llamado ATI Stream, pero posteriormente decidió apostar por OpenCL, que es un lenguaje de programación y un entorno de desarrollo más general, y que no está ligado a ningún dispositivo concreto (de hecho, ni siquiera es específico para GPUs y puede

generar código para CPUs con múltiples *cores*). Las razones para decantarse por CUDA fueron el estado más avanzado de desarrollo de su plataforma de programación y la mayor disponibilidad de bibliotecas de funciones de terceros, que son factores muy a tener en cuenta al embarcarse en un proyecto como este.

La actualidad.

Actualmente la programación de GPUs es una disciplina completamente madura. Al año se producen cientos de trabajos de investigación hechos posibles por las GPUs, con ganancias de rendimiento considerables. Muchos de ellos consisten simplemente en trasladar algoritmos ya utilizados previamente en otras plataformas a las nuevas GPUs, pero otros muchos son parte de esfuerzos por diseñar nuevas aplicaciones específicamente para ellas. Los fabricantes de las tarjetas actualizan periódicamente el conjunto de herramientas de diseño y desarrollo de aplicaciones para estas plataformas, lo cual facilita el trabajo de los programadores y permite, a su vez, aplicaciones más sofisticadas, ya que los recursos disponibles, tanto de programación como de características físicas de las tarjetas, siguen aumentando. Ha surgido, incluso, una serie de compañías que proporcionan desde librerías de funciones matemáticas, de gran interés en las aplicaciones de cálculo científico, hasta herramientas de desarrollo muy potentes, que permiten la creación, desarrollo y depuración de aplicaciones con mucha más comodidad que la que podían soñar los pioneros del cálculo científico en GPUs.

El proyecto Top500 [42], que se encarga de recabar información sobre los supercomputadores de los distintos centros de cálculo del

mundo para generar una lista de los 500 más potentes, recoge entre sus estadísticas el progresivo aumento de la utilización de GPUs entre ellos. Como ejemplo, en 2012 el número de supercomputadores que utilizaban GPUs ascendía a 62 de los 500 (desde 58 el año anterior). El informe se puede encontrar en [43]. Según el último informe hasta la fecha, de noviembre de 2013, 4 de los 10 más potentes utilizan GPUs.

2.1.2 La programación de GPUs para aplicaciones de propósito general.

Desde el punto de vista de la programación, la GPU funciona como un coprocesador matemático más, muy potente y sofisticado, pero sin diferencias conceptuales con los otros. De igual manera que antiguamente los microprocesadores de propósito general disponían en ocasiones de otros microprocesadores subordinados para ejecutar algunas funciones especiales (por ejemplo, el 8086 de Intel disponía del 8087 como coprocesador matemático para instrucciones de manejo de datos en coma flotante), ahora la CPU (del inglés, *Central Processing Unit*, generalmente referido al procesador principal del sistema) tiene a su disposición a la GPU para algunas tareas con unas características especiales. Estas características especiales son las que determinan si es conveniente trasladar la ejecución de esas tareas a la GPU o no, puesto que no todas se pueden beneficiar de ella. Sobre estas características se tratará con más detalle más adelante.

La forma de utilizar la GPU, pues, consiste en trasladarle partes de la aplicación. El resto de esta última, la parte que no se traslada a la GPU, se sigue ejecutando en la CPU, y entre ambas se establece una comunicación para transferir datos del problema desde la CPU a

la GPU y los resultados obtenidos en sentido inverso. La aplicación se conforma, de esta manera (ver Figura 2-2), como una secuencia de tareas repartidas entre la CPU y la GPU, con una relación de precedencia entre ellas marcada por los datos que es necesario transferir de tarea a tarea. Una tarea obtiene sus datos de la o las anteriores, procesa esos datos, y genera otros de salida, que debe tansferir a su vez a otra u otras tareas posteriores. Es labor del programador o del diseñador de la aplicación decidir cómo se divide la misma en tareas, y organizar la ejecución de cada una en la unidad de proceso (central o de gráficos) que corresponda, así como establecer qué datos y en qué momento deben ser transferidos a cuál de las unidades de proceso.

En la Figura 2-2 se muestra el proceso de ejecución de una aplicación repartida entre la CPU y la GPU. Hay tareas que se ejecutan en la CPU y otras que se ejecutan en la GPU. En la medida en la que la aplicación lo permita, es posible ejecutar de forma simultánea una tarea en la CPU y otra en la GPU, como en el punto 5 de la figura. En otras ocasiones no es posible simultanear, generalmente por relaciones de dependencia entre las tareas. Es el caso de los puntos 1 y 3. Habitualmente, antes de ejecutar una tarea en la GPU, es preciso transferir los datos de entrada (punto 2) y una vez ejecutada, devolver los resultados a la CPU (punto 4).

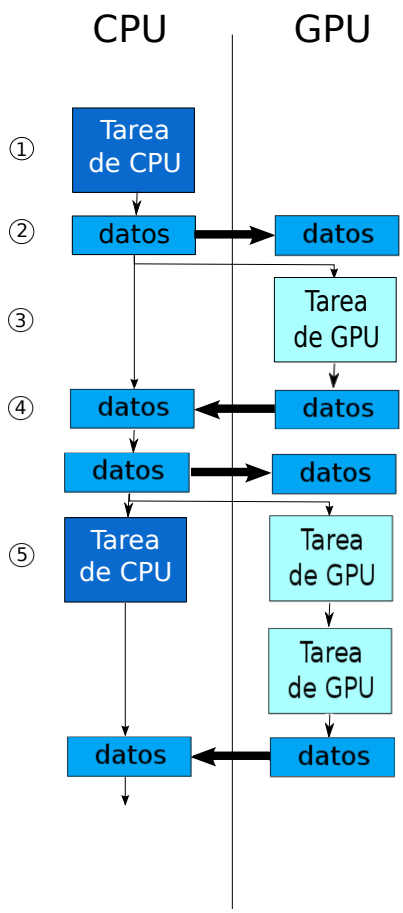


Figura 2-2. Ejecución de una aplicación en una GPU.

Las tareas que se trasladen a la GPU deben ser capaces de aprovecharse del modo de funcionamiento de la misma. Hay que tener en cuenta que una GPU está diseñada históricamente para procesar gráficos, y este procesamiento consiste en dos grandes grupos de tareas: procesamiento de vértices y procesamiento de texturas. En el primer caso, se trata de realizar operaciones geométricas sobre los vértices que definen las figuras en la imagen, como traslación, rotación, etc. El segundo grupo corresponde al análisis de las figuras para obtener los detalles de los puntos que las forman. Por ejemplo, es preciso averiguar a qué puntos se les debe aplicar una textura u otra, dependiendo de en qué superficie esté, o

qué puntos de qué superficies son visibles y cuáles no, etc. Todas estas operaciones deben realizarse para una enorme cantidad de elementos (píxeles, superficies, etc, según la operación) en cada imagen. Como la realización de una operación sobre un elemento es independiente de la realización de esa u otra operación sobre otro elemento, la carga de trabajo de las aplicaciones de gráficas es enormemente paralelizable, y es esta característica la que han explotado de forma sobresaliente las GPUs. Por su diseño, son capaces de operar simultáneamente sobre una gran cantidad de elementos de la imagen. Esa es la clave de su alto rendimiento. La conclusión de todo esto es clara: una aplicación de propósito general que consista en una serie de tareas que han de aplicarse a una gran cantidad de elementos de forma independiente tiene muchas posibilidades de poderse ejecutar con una mejora considerable del rendimiento en una GPU.

2.1.3 Estructura básica de una GPU de NVIDIA.

Para el trabajo que se presenta en esta memoria hemos seleccionado una GPU del fabricante NVIDIA. La principal razón para ello ha sido el estado más avanzado de las herramientas de diseño y desarrollo de aplicaciones que proporciona el fabricante, ya que esto es un punto muy a tener en cuenta al plantearse crear aplicaciones para una determinada plataforma. En esta sección se describe, desde un punto de vista general, la estructura básica de una GPU de NVIDIA. Esta estructura es reconocible en GPUs de otros fabricantes, pero, puesto que cada familia de productos tiene características particulares, es conveniente especificar que lo que sigue a continuación se refiere a la familia de GPUs de NVIDIA.

El Streaming Multiprocessor *de las tarjetas de NVIDIA.*

El elemento central de una GPU de NVIDIA (y de otras familias también) es el Streaming Multiprocessor (la traducción es difícil; se podría traducir como Multiprocesador de Flujos, pero, por la dificultad de la traducción, se utilizará el nombre en inglés o las siglas SM). La Figura 2-3 muestra la estructura de un SM.

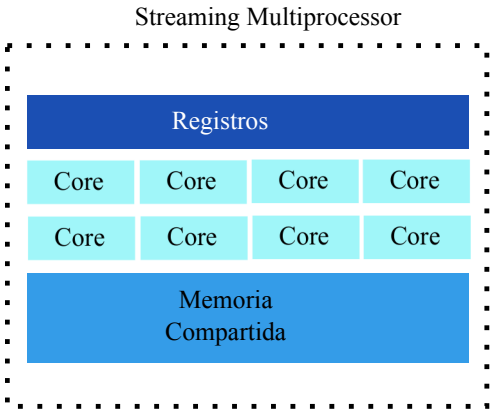


Figura 2-3. ***Streaming Multiprocessor.***

Un SM está formado por una serie de *Streaming Processors* o *Cores*. Un *Core* es un elemento básico de computación, y sería el equivalente de una Unidad Aritmético-Lógica en un microprocesador. Es capaz de ejecutar una instancia individual de una instrucción (aunque la instrucción ha de ser la misma para todos los *cores*), y tiene acceso a los registros y a la memoria compartida del SM. El número de *Cores* de un SM es una característica de la tarjeta gráfica, y varía entre generaciones de la misma familia. En las GPUs de la generación Tesla, la primera desarrollada por NVIDIA, el número de *cores* por SM es de ocho. Este número se mantuvo en la siguiente generación, Fermi, pero en la más reciente a la hora de escribir esta memoria, la generación Kepler, se ha ampliado hasta 192.

Como se indica más arriba, los *cores* de un SM ejecutan todos una instancia diferente de la misma instrucción. Es un concepto parecido a la categoría SIMD (*Single Instruction, Multiple Data*, Instrucción Única, Múltiples Datos) de la taxonomía de Flynn [44], aunque NVIDIA prefiere llamar a esta forma de ejecutar instrucciones SIMT (*Single Instruction, Multiple Thread*, Instrucción Única, Múltiples Hilos), ya que de esta manera se hace referencia a las características multi-hilo de las GPUs.

El concepto de instancia de una instrucción se desarrolla más ampliamente en secciones posteriores, pero está basado en la utilidad principal de las GPUs. Éstas están diseñadas para aplicaciones que deben realizar repetidas veces las mismas operaciones sobre un conjunto numeroso de datos. El ejemplo más sencillo de esto es una suma de dos vectores. La operación que se repite es la suma, y el conjunto de datos son todos los pares de componentes de los dos vectores, donde un elemento del par pertenece al primer vector y el otro al segundo. Cada suma de un par de componentes supone una instancia de la operación suma, y podría ser ejecutada en un *core* diferente.

Para gestionar el SM, existe una unidad de instrucciones, que se encarga de recibir el código que el SM va a ejecutar, de decodificar las instrucciones y gestionar los accesos a los registros y a la memoria compartida, y de indicar a los *cores* la instrucción que deben ejecutar. Los registros del SM suponen el almacenamiento más rápido de que consta la GPU, y se reparten entre los distintos *cores* de forma dinámica, en función de las necesidades del código que se ejecuta en el SM. Una vez asignados, no es posible que un *core* acceda a un registro de otro *core*, aunque la asignación se puede

cambiar cuando al SM le sea asignado otro código para ejecutar. Por su parte, la memoria compartida tiene como principal utilidad precisamente el almacenamiento de datos que los *cores* necesitan compartir.

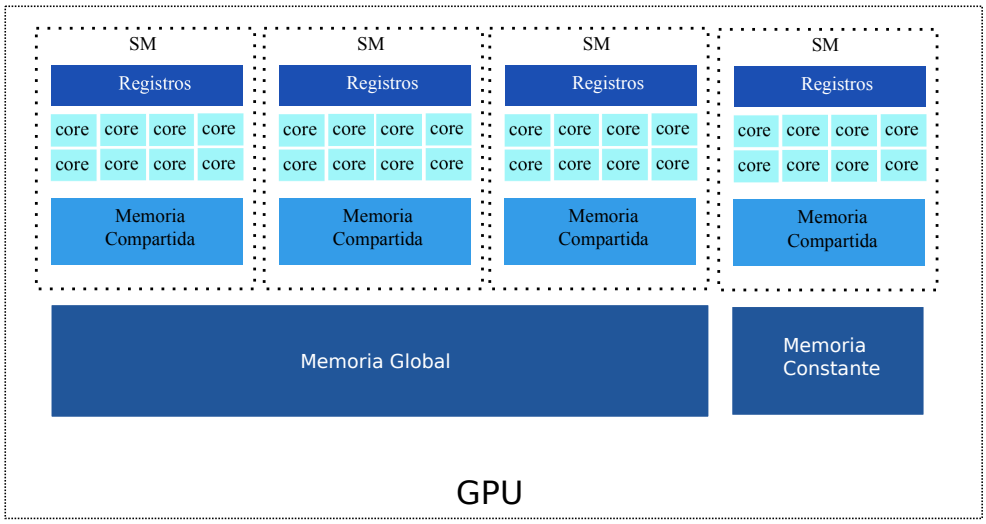


Figura 2-4. Diagrama de bloques de una GPU.

La Figura 2-4 es un diagrama de bloques de una GPU. Una GPU está compuesta por una serie de SMs, en mayor o menor número dependiendo de la gama de la tarjeta. Las tarjetas de gama baja pueden tener un único SM, mientras que las de gama más alta, como la Tesla C1060 que se ha utilizado en el trabajo que describe esta memoria, tiene 30. Además de los SMs, la GPU dispone de una memoria global, la más abundante, y una memoria para constantes. La primera es la vía de entrada de los datos a la GPU. Una vez ahí, se distribuyen al resto de memorias según sea necesario. La segunda es un almacén especial para datos que no se ven modificados con la ejecución de la aplicación.

La jerarquía de memoria.

En la sección anterior ya se ha hecho referencia a los niveles de la jerarquía de memoria de la GPU. En esta sección se van a presentar las características más importantes de cada uno de ellos, especialmente en lo que respecta al tipo y al tiempo de acceso de cada una.

Tabla 2-1. Tipos de acceso

Memoria	**Ámbito**
Registros	*Core*
Compartida	SM
Local	*Core*
Global	GPU
Constantes	GPU

La Tabla 2-1 resume el tipo de acceso que admite cada nivel de la jerarquía. Las posibles opciones son:

core: los datos sólo son accesibles por un *core* específico, aquel al que le ha sido asignada la posición en la que están almacenados.

SM: los datos son accesibles por parte de todos los *cores* del mismo SM.

GPU: los datos son accesibles por cualquier *core* de la GPU.

Generalmente, el compilador intenta ubicar en registros todos los datos locales de las funciones, ya que estos datos no necesitan ser

compartidos por los *cores*. Si una función tuviera más datos locales de los que caben en el conjunto de registros que se le asignan a un *core*, el compilador los colocaría en la memoria local. La memoria local es un nivel lógico de la jerarquía, y se utiliza para este tipo de datos, los datos locales que no caben en registros. Físicamente está ubicada en la memoria global, aunque, por su función, no es accesible más que por el *core* al que le corresponda. Los datos compartidos por varios *cores* pueden residir en memoria global o en memoria compartida. Ésta última sólo es accesible desde los *cores* que están situados en su mismo SM, por lo que el programador debe analizar cuidadosamente si una variable se puede ubicar en ella o no. En caso de que deba poder ser utilizada por cualquier *core*, independientemente de en qué SM esté, la variable debe residir en memoria global.

Aquellos datos que no se modifiquen durante la ejecución de la aplicación y que sean utilizados por todos los *cores* pueden residir en la memoria constante. Esta es una memoria de acceso global, pero con la característica de que está "cacheada" en cada SM, por lo que el acceso es mucho más rápido que el realizado a memoria global.

Con respecto al tiempo de acceso a los niveles, la regla básica es que cuanto más cerca del *core* se encuentre la memoria, más rápido es su acceso. Así, los registros suponen el almacenamiento más rápido, de ahí que el compilador prime su utilización frente a cualquier otro nivel. Desgraciadamente, los registros son un recurso escaso (un SM de la generación Tesla dispone de aproximadamente 8000 posiciones), por lo que, en la medida de lo posible, también es interesante utilizar el siguiente nivel por tiempo de acceso, la memoria compartida. Ésta es algo más lenta y algo más grande que el

banco de registros. Para los datos que no puedan residir ahí (por limitación de espacio, o porque deban ser compartidos por todos los SMs), se puede utilizar la memoria global, la más lenta de todas. Por último, la memoria de constantes tiene un tiempo inicial de acceso parecido al de la memoria global, pero el hecho de que sea "cacheada" por los SMs hace que los accesos siguientes sean más rápidos.

"Computing Capabilities"

Los ingenieros de NVIDIA son conscientes de que, con el ritmo de salida de sus productos al mercado, la presencia simultánea en cualquier momento de tarjetas de distintas generaciones es inevitable. Uno de sus objetivos de diseño es permitir que las aplicaciones se adapten a la tarjeta sobre la que se han de ejecutar con la mayor comodidad posible para el desarrollador. Para ello han creado el concepto de *Computing Capability* (Capacidad de Cómputo). Se trata de un número que indica el grado de avance de la tarjeta con respecto a las mejoras que NVIDIA va introduciendo. A cada número le corresponde un subconjunto de características, de forma que una tarjeta con un número menor tiene menos características que una con un número mayor.

Esta jerarquización de las tarjetas permite al programador incluir código en su aplicación que obtenga de la tarjeta cuál es su "Capacidad de Cómputo" y, en función del resultado, activar o no algunas características de la aplicación. El resultado es que las aplicaciones se adaptan fácilmente a la tarjeta en la que corren. El Apéndice A de la Guía de Programación de CUDA [45] lista las capacidades de cómputo de distintas tarjetas de NVIDIA. Las más

interesantes están resumidas en la Tabla 2-2, junto con la capacidad de cómputo a partir de la cual están disponibles. Algunas de ellas se describirán en posteriores secciones.

Tabla 2-2. Características en función de la capacidad de cómputo

Característica	**Disponible desde:**
Operaciones atómicas con enteros en memoria global	1.1
Intercambio atómico para coma flotante en memoria global	1.1
Operaciones atómicas con enteros en memoria compartida	1.2
Intercambio atómico para coma flotante en memoria compartida	1.2
Operaciones atómicas de 64 bits en memoria global	1.2
Votación en *warps*	1.2
Cálculos en coma flotante de doble precisión	1.3
Operaciones atómicas con enteros de 64 bits en memoria compartida	2.0
Sumas atómicas de coma flotante de 32 bits en memoria global y compartida	2.0
Paralelismo dinámico	3.5

2.1.4 Familia de tarjetas NVIDIA Tesla.

El nombre 'Tesla' provoca una cierta confusión al estudiar la gama de productos de NVIDIA, ya que el fabricante lo utiliza en dos contextos: por un lado, define una gama de tarjetas de gran potencia destinadas de forma exclusiva a computación de altas prestaciones, y por otro, se refiere a la arquitectura de una de las generaciones de sus tarjetas (tanto gráficas como de computación de altas prestaciones). En esta sección se describe la evolución de la familia de tarjetas Tesla, es decir, se utiliza el nombre 'Tesla' en el primero de los sentidos mencionados, aunque será preciso, en su momento, referirse a la 'arquitectura Tesla'.

Las primeras tarjetas de NVIDIA que admitían CUDA, y por lo tanto podían ser utilizadas con relativa facilidad para computación científica, fueron las de la serie G80. Se trataba de tarjetas gráficas para su uso tradicional, pero además, dado que estaban diseñadas a partir de elementos programables, susceptibles de ser utilizadas para cálculo científico. NVIDIA lanzó el entorno de desarrollo CUDA precisamente para facilitar esto último, y numerosos programadores se lanzaron a aprovechar las posibilidades que este hecho les presentaba.

NVIDIA reconoció el mercado que se le abría y diseñó una tarjeta específica para computación científica. Se trataba de la Tesla C870, basada en la misma arquitectura que sus contemporáneas gráficas.

El siguiente paso importante fue el desarrollo de una nueva arquitectura para sus tarjetas, la arquitectura Tesla. Esta arquitectura servía de base para numerosas tarjetas gráficas, y además, para un nuevo miembro de la familia Tesla, la Tesla C1060, sobre la que se ha desarrollado parte del trabajo presentado en esta

memoria. Para evitar la confusión explicada anteriormente, basta distinguir la arquitectura concreta de las tarjetas (gráficas y de cálculo científico) de esta generación de la familia concreta de tarjetas de cálculo científico. Ambas líneas (arquitectura y familia) se cruzan en este modelo concreto.

A continuación, NVIDIA desarrolló una nueva arquitectura, la arquitectura Fermi. De nuevo sirvió de base para una generación de tarjetas gráficas y de cálculo científico, en este último caso la Tesla Fermi, de la cual hay tres variantes, la C2050, la C2070 y la C2075. El modelo C2075 es otro de los que se han utilizado para la realización de este trabajo, y se hará referencia a él en el capítulo de resultados.

Actualmente, las tarjetas de NVIDIA utilizan la arquitectura Kepler, y forman la cuarta generación de tarjetas que admiten CUDA. Las variantes de cálculo científico son las Tesla Kepler, a día de hoy los modelos K10, K20 y K20X. NVIDIA ha anunciado dos arquitecturas nuevas que deben seguir a Kepler: Maxwell y Volta. En el momento de escribir esta memoria, aún no se conocen detalles de estas arquitecturas. La Tabla 2-3 lista la capacidad de cómputo de las tarjetas de la familia Tesla.

Tabla 2-3. Capacidad de cómputo de la familia Tesla

Tarjeta	**Capacidad**
C870	1.0
C1060	1.3
C 2050/2070/2075	2.0
K 10/20/20X	3.5

2.2 Modelo de Ejecución de una GPU.

El modelo de ejecución explica cómo se ejecuta el código en una GPU. Es radicalmente diferente al de una CPU, y es fundamental comprenderlo para poder aprovechar al máximo las capacidades de cómputo que ofrece la GPU. Esta sección se centra en la programación de una GPU por medio de las herramientas de desarrollo de NVIDIA, especialmente el *CUDA Toolkit*, que está basado en el lenguaje de programación C. Otras empresas proporcionan otras herramientas de desarrollo para otros lenguajes, por ejemplo, para FORTRAN. El modelo de ejecución es el mismo en cualquier caso, pero es preciso tener esto en cuenta al analizar los ejemplos que se incluyen.

2.2.1 El *kernel*.

La unidad elemental de asignación de trabajo a la GPU es el *kernel*. Un *kernel* es sintácticamente muy parecido a una función de lenguaje C, con algunas salvedades que tienen como objetivo añadir información para su ejecución en la GPU. Del código del *kernel* se crean múltiples instancias en la GPU, de una forma muy parecida a como se crean hilos en la programación multi-hilo. Cada una de estas instancias o hilos debe trabajar sobre un subconjunto de los datos de la aplicación. Es importante tener en cuenta que el código es el mismo para todos los hilos, es decir, no se crean distintas versiones de él. El número de hilos que se crean lo debe especificar el programador de la forma que se indica más adelante.

El Listado 2-1 muestra cómo se define un *kernel*. Tiene el formato de una definición clásica de una función, con las únicas salvedades de no requerir un tipo para el valor devuelto y tener un calificador:

`__global__`. Este calificador es la forma que tiene CUDA de indicarle al compilador que el código de esta función deberá ejecutarse en la GPU, por lo que, entre otras cosas, debe generarse código máquina para la misma, y no para la CPU.

```
__global__ void media (float *x, float  promedio)
{

      ...

      return;
}
```

Listado 2-1. Definición de un *kernel*.

El kernel se invoca desde el programa principal con una sintaxis parecida a la de una función normal, con algún detalle más. El

Listado 2-2 muestra cómo. Se puede observar cómo es esencialmente igual a una invocación a una función, con la única diferencia de que se añaden un par de parámetros a la invocación, en concreto:

```
<<<Bloques,Hilos>>>
```

Esto es una de las modificaciones al C estándar que añade CUDA, y sirve para indicarle a la GPU el número de hilos que crear. El significado de estos parámetros se describe en la siguiente sección.

```
float x[512], promedio[512];

...

int main (int *argc, char **argv)
{
      dim3 Bloques(1,1);
      dim3 Hilos(512,1,1);

      ...

      /* Lanzar kernel "media" con "Bloques"
         bloques en el grid, e "Hilos" hilos por
         cada bloque */

      media<<<Bloques,Hilos>>>(x,promedio);

      ...

      return 0;
}
```

Listado 2-2. Invocación de un *kernel*.

2.2.2 Organización de un *kernel* en ejecución.

Una GPU de alta gama de la arquitectura Tesla, con 8 *cores* por cada SM y un total de 30 SMs, tiene un total de 240 *cores*. Con estos *cores*, puede mantener en ejecución a otros tantos hilos de forma simultánea. Sin embargo, para evitar que las pérdidas de tiempo que se dan en la ejecución normal de una aplicación (esperas por largos accesos a memoria, por dependencias de datos o de control entre instrucciones, por fallos de caché, etc) afecten al rendimiento, está diseñada para admitir muchos más que esos 240 hilos. De esta forma, cuando alguno de ellos está detenido esperando que termine alguna de estas operaciones, puede ser retirado de la ejecución y ser

sustituido por otro que no tenga que esperar. Así es posible (en teoría) aprovechar el tiempo al máximo.

Con el objeto de poder gestionar grandes cantidades de hilos sin tener que pagar el correspondiente coste en las estructuras de control que suele incorporar una CPU, y que se llevan gran cantidad del silicio del microprocesador en estos casos, las GPUs cuentan con una forma simplificada de organizar los hilos. Éstos se agrupan formando una organización jerárquica de dos niveles. En el nivel más bajo de los dos se encuentran los *bloques*. Un bloque es un grupo de hilos, y puede tener hasta tres dimensiones. El bloque es la unidad de asignación a un SM. El número máximo de hilos que puede haber en un bloque es un valor fijado por la arquitectura de la GPU. En el caso de la Tesla C1060 este número es 512.

El nivel superior de la jerarquía lo forma el '*grid*' (la traducción sería algo así como *rejilla* o *cuadrante*). Los bloques se disponen en un *grid* o rejilla de, como máximo, dos dimensiones. Un *grid* puede contener un máximo de 2^{16}x2^{16} bloques, es decir, algo más de cuatro mil millones de bloques. La Figura 2-5 muestra un grid con sus bloques, y un bloque con sus hilos.

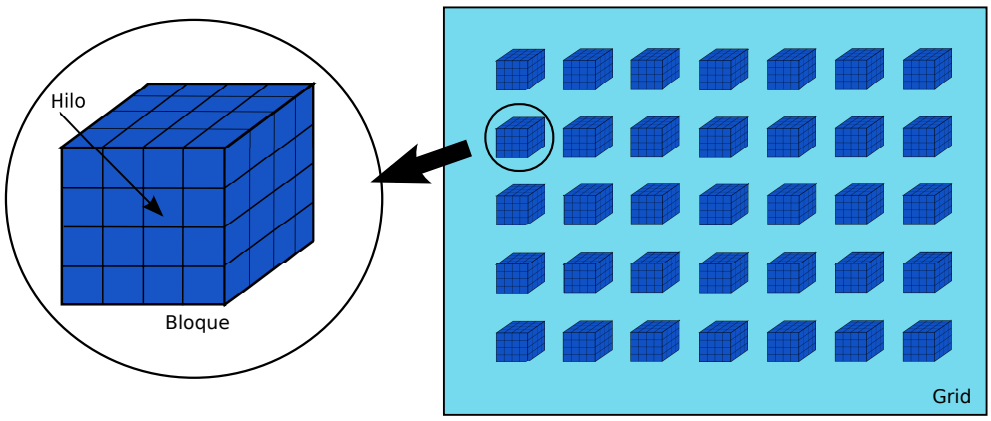


Figura 2-5. Organización de un grid y un bloque.

Juntando todo lo anterior, el número máximo de hilos que puede manejar una GPU es del orden de dos billones. En realidad, este número es el máximo de hilos de que puede constar un *kernel*. El número máximo de hilos *en ejecución* en un instante dado se ve afectado por otras limitaciones, la principal de las cuales es el número máximo de hilos que es capaz de manejar un SM con los recursos de que dispone. Este punto se amplía un poco más en la próxima sección.

Para permitir a cada hilo seleccionar la porción de los datos con la que debe trabajar, CUDA incorpora una serie de variables predefinidas. Estas variables están disponibles al programador sin necesidad de declararlas, y hacen referencia a la posición de cada hilo en la jerarquía globlal de hilos. Puesto que tienen un valor diferente e identificativo en cada hilo, se pueden usar para repartir el trabajo entre ellos. Entre estas variables predefinidas, las más comúnmente utilizadas son `threadIdx`, `blockIdx`, `blockDim` y `gridDim`. La variable `threadIdx` responde a un tipo abstracto de datos, en forma de `struct` de tres enteros. Sus campos son `x`, `y` y `z`. Estos campos indican la posición del hilo en el bloque. La variable `blockIdx` es análoga, y proporciona la posición del bloque al que pertenece el hilo en el *grid*. En este caso, ya que un *grid* sólo tiene dos dimensiones, la variable sólo tiene dos campos, `x` e `y`. Las variables `blockDim` y `gridDim` definen las dimensiones de los bloques y el *grid*. Son también un `struct`, de tres y dos campos respectivamente, correspondiendo a las dimensiones de un bloque y un *grid*.

2.2.3 Asignación de recursos en la GPU.

Una vez que se lanza un *kernel*, se analiza la configuración del mismo para saber las dimensiones del *grid* y de los bloques. Con los datos obtenidos se empieza a hacer la asignación hilos a *cores*. Dado que uno de los objetivos a la hora de diseñar CUDA era que las aplicaciones se adapten con facilidad a las distintas tarjetas en las que van a poder ejecutarse, esta asignación debe ser fluida y rápida. Además, como es lógico, debe tener en cuenta las limitaciones de la tarjeta en la que corre.

El proceso de asignación se hace en dos etapas. Inicialmente, se asignan bloques a SMs. El grid puede tener, como se ha explicado anteriormente, hasta unos cuatro mil millones de bloques. Un SM puede albergar un máximo de 8 bloques. Este número puede verse reducido por otros factores, además, pero si se consigue el máximo, una tarjeta como la Tesla C1060, con sus 30 SMs puede manejar, en un instante dado, hasta 240 bloques.

La asignación de bloques se adapta dinámicamente al número de SMs de que disponga la tarjeta (ver Figura 2-6), de forma que el mismo código se puede ejecutar sobre una gran variedad de tarjetas sin necesidad de recompilar o reconfigurar. Los SMs reciben bloques mientras tengan recursos libres (y la aplicación bloques sin asignar). En realidad, el que a un SM se le agote algún recurso impide la asignación de más bloques a ese SM, aunque otros recursos no se le hayan agotado. Cuando ningún SM pueda recibir ningún bloque más, se interrumpe la asignación hasta que alguno pueda volver a admitir bloques. Esto último ocurre cuando ese SM termine completamente la ejecución de alguno de los bloques que tenía asignados.

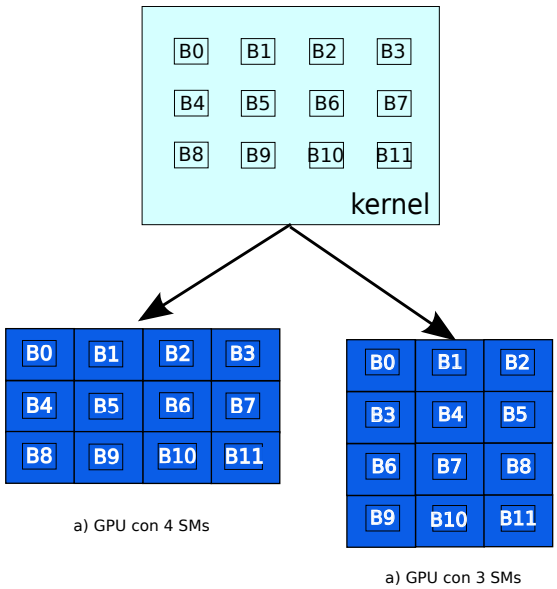


Figura 2-6. Asignación de bloques a SMs en distintas GPUs.

Los recursos con los que cuentan los SMs, además de los 8 *cores* de ejecución, son registros, memoria compartida, y capacidad de control de bloques, hilos y *warps*[1]. En breve se explicará lo que es un *warp*, pero antes es preciso especificar la cantidad de cada recurso mencionado. Esta cantidad varía con la arquitectura, y tarjetas más recientes pueden asignar más recursos a cada SM. En el caso de la Tesla C1060, cada SM tiene hueco, como ya se ha mencionado, para 8 bloques. Además, puede controlar 24 *warps* y 768 hilos. Con respecto al espacio de almacenamiento, dispone de 8192 registros (de 4 bytes) y 16 KB de memoria compartida. El almacenamiento se reparte entre todos los hilos asignados al SM.

[1] *Warp* es un término de la industria textil, y se refiere a la urdimbre del tejido. NVIDIA juega con la noción de hilo para referirse a un grupo de ellos por medio de este término. En esta memoria hemos mantenido el nombre original inglés.

De entre todos los recursos, el primero que se agote impide la asignación de nuevos bloques hasta que queden más libres. Por ejemplo, el SM puede aceptar 8 bloques, pero si recibe uno de 512 hilos, y otro de 256, ha completado el número de hilos que puede gestionar, por lo que no aceptará más bloques, aunque en teoría podría aceptar otros 6. Análogamente, si recibe un bloque de 512 hilos, cada uno de ellos con necesidad de 15 posiciones de almacenamiento en registros (lo que no es demasiado), el total de posiciones para el bloque sería 512*16=8192. Eso agotaría completamente el espacio de registros, lo que no daría para otro bloque más. Por lo tanto, aunque habría suficientes huecos para bloques (7) y para hilos (256), no podría aceptar más.

2.2.4 Ejecución de los hilos.

Agrupación de los hilos de un bloque.

En las secciones anteriores se ha explicado cómo se asignan hilos a los SMs. En esta se mostrará cómo se ejecutan una vez en ellos, es decir, en qué momento entran en ejecución, cómo se selecciona qué hilos se ejecutan, cómo se asignan los hilos a los *cores* del SM, etc.

La primera cuestión es decidir qué bloque de entre los asignados al SM se ejecuta antes y por cuánto tiempo, y la solución es la más simple: el primero en llegar y mientras no tenga sus hilos detenidos esperando a algún dato.

Los hilos de un bloque se agrupan en *warps*. Un *warp* es una cantidad de 32 hilos, por lo que, como el número de hilos del bloque puede no ser múltiplo de 32, el último se rellena con hilos "vacíos".

Los hilos se van agrupando siguiendo primero la "dimensión" x del bloque, luego la y, y por último la z. La Figura 2-7 muestra el proceso para un bloque de dos dimensiones con 16 hilos en x (el ancho) y 16 en y (el alto).

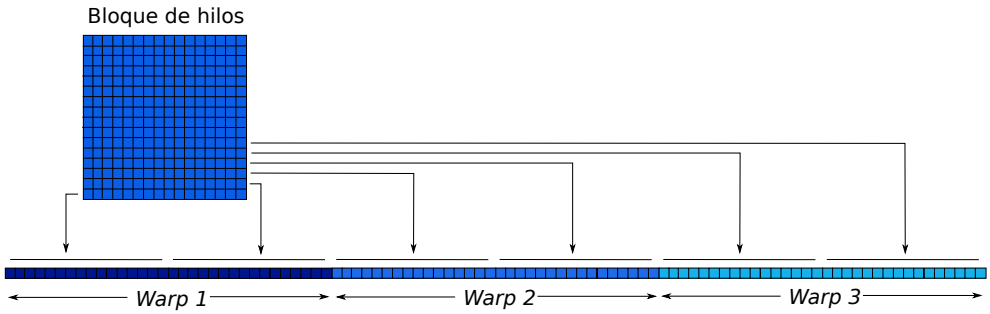


Figura 2-7. Formación de *warps* a partir de los hilos de un bloque.

El SM selecciona un *warp* para ejecutar en función de criterios como la antigüedad o el estado (si no está esperando el resultado de una carga, por ejemplo). Entonces ejecuta la primera instrucción pendiente del *warp* (la primera de todas las que forman el *warp* en caso de que éste no se haya ejecutado hasta ese momento), y la ejecuta para todos los hilos que lo forman. Puesto que son 32 hilos, y el SM tiene 8 *cores*, tarda 4 ciclos en ejecutar la instrucción para todos los hilos que forman el *warp*. La ejecución del *warp* continúa de esta manera con la siguiente instrucción, hasta que una de ellas se bloquea (por una espera, por ejemplo) o el *warp* termina su ejecución. En ese momento se selecciona otro, y se sigue la ejecución.

El proceso de ejecución de instrucciones, con más detalle, es como sigue: el SM dispone de 8 *cores* y dos SFUs (*Special Function Unit*, Unidad de Funciones Especiales). El *core* estándar es en realidad una unidad de multiplicación y suma (*Multiply-Add Unit*, MAU) escalar

en coma flotante y precisión simple (32 bits). Las instrucciones más complejas (trigonométricas, raíces, funciones transcendentes, etc) se ejecutan en las SFUs. Cada una de éstas tiene 4 multiplicadores en coma flotante. Tanto las SFUs como los *cores* están segmentados, y pueden admitir una operación cada ciclo.

El envío de instrucciones a los *cores* y a las SFUs se va alternando cada dos ciclos. Es decir, primero se selecciona una instrucción (ciclo 1) y se envía a los *cores* (ciclo 2). A continuación se selecciona una instrucción para las SFUs (ciclo 3) y se envía (ciclo 4). Los envíos se van solapando porque los *cores* tardan 4 ciclos en ejecutar la instrucción asignada para los 32 hilos del *warp*. Con este mecanismo se consigue esconder el tiempo de planificación, o, como dice NVIDIA, se consigue la planificación de instrucciones a coste cero ciclos.

Los hilos pueden ejecutar instrucciones de salto condicional. Cuando esto ocurre, el SM analiza qué hilos del *warp* lo toman y qué hilos no. Este fenómeno se denomina *divergencia*. Lo que hace el SM es ejecutar una de las ramas, con los hilos del *warp* que la siguen y bloqueando los que no. Éstos últimos deben contabilizarse como ciclos perdidos, ya que no se aprovechan sus *cores* para hacer trabajo útil. A continuación ejecuta la otra rama, bloqueando los hilos que antes se ejecutaron y activando los que no. En este caso, los hilos que cuentan como ciclos perdidos son los primeros. La ejecución normal se reanuda en el punto de convergencia de las dos ramas. Esta forma de ejecución se plantea a nivel de *warp*. Es decir, puede ocurrir que un salto condicional sea tomado de forma diferente por los hilos de un *warp* (como en el ejemplo anterior), mientras que en otro *warp* todos los hilos se comporten de la misma manera ante él (todos lo

toman o todos no lo toman). En este segundo *warp* no se daría penalización, ya que no es necesario ejecutar nada más que una rama del salto, y todos los hilos del *warp* estarían activos.

El efecto de un salto condicional es, generalmente, una gran pérdida de rendimiento ya que, en promedio, la mitad de los *cores* se quedan sin hacer trabajo útil mientras se ejecutan las ramas del salto. Por ello, el compilador tiende a optimizar el código sustituyendo los saltos condicionales por código lineal mediante la *predicación* [46] si es posible. Si las ramas son muy largas, es conveniente que el programador se plantee cómo eliminar el salto condicional, tal vez sustituyéndolo por otra versión del código en la que el salto no sea necesario, aún a costa de hacer un poco más largo el programa. Si una aplicación consta de una serie de *kernels* en los que es imposible evitar el uso continuo de saltos condicionales, puede ser indicativo de que, bien el diseño de la aplicación no está bien hecho, bien no es una aplicación que se vaya a beneficiar en gran medida de la ejecución en una GPU.

Sincronización de los hilos de un bloque.

En ocasiones es preciso introducir la sincronización entre los hilos de un *kernel*. Estas situaciones se dan cuando es preciso que todos los hilos hayan terminado una parte del *kernel* antes de que se pueda continar con su ejecución, por ejemplo porque cada hilo deba calcular un valor que vaya a ser usado por los demás. Dada la forma en la que se ejecutan los hilos en CUDA, no es posible garantizar el orden en la ejecución de los hilos, ni que todos avancen a la par. En CUDA la sincronización se puede dar entre todos los hilos de un mismo bloque,

pero no es posible sincronizar hilos de diferentes bloques. La función que proporciona esta operación es:

```
void __syncthreads();
```

Esta función lo único que hace es esperar a que todos los demás hilos la hayan ejecutado. Tiene utilidad, como se ha mencionado, para evitar mal funcionamiento del código cuando se dan dependencias de lectura después de escritura entre hilos del mismo bloque. Es muy importante tener cuidado de asegurarse de que todos los hilos la ejecutan, ya que en caso contrario el *kernel* se quedará bloqueado (los hilos que la ejecutan se quedan esperando a los hilos que no la ejecutan y no terminan nunca). Por ejemplo, si se utiliza en una rama de un salto condicional, hay que asegurarse de que también se ejecutará en la otra.

Las tarjetas con capacidad de computación 2 o superior admiten variantes de esta función. Todas ellas incorporan la funcionalidad básica de syncthreads. La Tabla 2-4 describe el funcionamiento de las demás funciones de la familia.

Tabla 2-4. Variaciones sobre la función syncthreads.

Función	**Descripción**
`int __syncthreads_count(int predicado);`	evalúa el valor de la variable `predicado` en cada hilo y devuelve el número de ellos en el que es distinto de cero.
`int __syncthreads_and(int predicado);`	evalúa el valor de la variable `predicado` en cada hilo y devuelve cero si ésta toma el valor cero en todos.
`int __syncthreads_or(int predicado);`	evalúa el valor de la variable `predicado` en cada hilo y devuelve cero si ésta toma el valor cero en alguno de ellos.

Además de las funciones de sincronización, en CUDA existen las llamadas *operaciones atómicas*. Se trata de instrucciones que ejecutan una secuencia de lectura-modificación-escritura sobre una posición de memoria sin interrupción, por parte de otros hilos, durante todo el proceso. Es decir, no permiten que, una vez empezada la lectura del dato por el hilo que la ejecuta, otro hilo pueda leer o modificar el dato hasta que el primero ha terminado la escritura final. Estas operaciones están relacionadas con el uso de semáforos, aunque no se restringen a él. Son útiles, por ejemplo, para acumular resultados de todos los hilos en un único dato, garantizando que el valor resultante de la acumulación (en la terminología de la

programación, una *reducción*) se va actualizando correctamente por todos los hilos. La fase de modificación de la operación atómica puede tomar diversas formas: suma, resta, intercambio, mínimo, máximo, etc. El apéndice B de la Guía de Programación [45] describe en detalle todas ellas, y además indica a partir de qué capacidad de computación están disponibles.

Streams.

Los *streams* (la traducción del inglés sería algo así como *flujos*) se podrían definir de una forma simplificada como colas de operaciones en la GPU. Las operaciones de estas colas pueden ser lanzamientos de un *kernel*, operaciones de transferencia de datos, eventos, etc. No se deben confundir con los hilos de un *kernel*, ni con sus instrucciones. Un *stream* proporciona un medio sencillo de controlar la secuencia de operaciones que se deben ejecutar en la GPU.

Las operaciones de un stream habitualmente se hacen de forma síncrona. Esto quiere decir que, mientras la primera no termina, la segunda no empieza. Esta forma de funcionar no es la única posible, y también se puede especificar, en el caso de algunas operaciones (por ejemplo, transferencias de datos), que no es necesario que la segunda operación espere a que se termine la primera, o, técnicamente, que la primera operación funciona en modo asíncrono. Por ejemplo, si se deben extraer de la GPU los resultados de un *kernel* y, a continuación, se debe ejecutar un segundo *kernel*, y siempre y cuando éste último no necesite el espacio de memoria liberado con los datos del primero, podrá ejecutarse sin esperar a que se copien esos datos completamente. En esta situación, la transferencia de datos desde la GPU hacia la CPU puede realizarse en paralelo con la ejecución del

segundo *kernel*. No todas las GPUs permiten este funcionamiento. Las más antiguas no son capaces de gestionar las dos operaciones de forma simultánea. Pero las que pueden, reducen el tiempo global de ejecución de la aplicación.

En el ejemplo del párrafo anterior se muestra el paralelismo que se puede dar entre operaciones de un mismo *stream*. Las aplicaciones sencillas de GPU generalmente están escritas de esta manera y usan un único *stream*, pero es posible usar todos los que se necesiten. De hecho, en ocasiones el uso de varios *streams* permite mejorar el rendimiento ya que, incluso en el caso de que todas las operaciones de cada *stream* funcionen en modo síncrono, lo que sí es posible es ejecutar en paralelo operaciones de diferentes *streams*, siempre que las operaciones no utilicen los mismos recursos de la GPU (por ejemplo, si una es una transferencia de datos y la otra la ejecución de un *kernel*) y la tarjeta permite simultanearlas.

2.2.5 Uso de la jerarquía de memoria.

Uno de los factores que mayor influencia tienen en el rendimiento de una aplicación CUDA es el uso que haga de la memoria de la tarjeta. Como se ha explicado anteriormente, una GPU dispone de distintos tipos de almacenamiento, cada uno con sus características de extensión y tiempo de acceso (ver Figura 2-4). El uso inteligente de las ventajas de cada uno de ellos en la medida en la que pueda beneficiar a la aplicación es lo que puede hacer que una aplicación obtenga un gran rendimiento. Por el contrario, si no se tiene en cuenta el funcionamiento del sistema de memoria de la tarjeta, es muy fácil que el rendimiento obtenido no sea demasiado interesante. Los tipos de memoria de una GPU ya han sido descritos

anteriormente. En esta sección se explican con algo más de detalle y concretando para una tarjeta como la Tesla C1060

El nivel de acceso más rápido es el de los *registros*. Un SM dispone de 8192 bytes de registros. Si se tiene en cuenta que tanto enteros como números en coma flotante de precisión simple ocupan 4 bytes, el número total de datos que puede almacenar un SM en registros es de 2048. Hay que tener en cuenta que esa cantidad debe repartirse entre todos los hilos de todos bloques asignados a ese SM. Puesto que el número máximo absoluto de hilos que puede gestionar un SM es de 768, en promedio podrían tener entre 2 y 3 datos en registros.

El tipo de variable que el compilador suele asignar a los registros son las variables locales escalares del *kernel*. Si, debido al diseño de éste, no hubiera suficiente espacio en el bloque de registros para contener estas variables, el compilador utilizará para las que no caben una zona de memoria global denominada memoria local. Ésta no es un tipo distinto de memoria de la GPU, sino una partición lógica de la memoria global, con una forma de acceso diferente al habitual: cada hilo tiene acceso exclusivo a sus variables almacenadas en ella (es decir, no se pueden compartir). El problema es que la memoria global es la más lenta de todas las que tiene la GPU. Por lo tanto, hay que tener este hecho en cuenta cuando se deciden las variables locales que va a utilizar un *kernel*. La presión no es tanta como parece en un principio, ya que los registros se pueden reutilizar (es decir, cuando una variable local deja de usarse en el *kernel*, el registro al que se asignó puede ser utilizado para contener otra), pero es preciso diseñar este aspecto del *kernel* con cuidado.

Otro tipo de memoria rápida, algo más extensa, y que tiene la ventaja de que se puede compartir entre hilos, es la memoria compartida. Además de su tiempo de acceso (algo mayor que el de los registros) y su mayor capacidad, una diferencia importante con los registros es que el programador puede etiquetar variables para que se almacenen en ella. Si un array de datos debe ser compartido por varios hilos, basta etiquetar el array como compartido para que se almacene en esta. La compartición es posible para todos los hilos de un mismo bloque, pero no para hilos de distinto bloque. Como muestra la Figura 2-4, cada SM tiene su porción de memoria compartida, que no es accesible por otros SMs. Si es preciso que una variable sea compartida por hilos de distintos bloques, debe ser almacenada en memoria global o constante, dependiendo de si puede o no verse modificada. La cantidad de memoria compartida presente en un SM varía con los modelos de tarjeta. En la Tesla C1060 es de 16 KB. Esta memoria debe repartirse entre todos los bloques que se ejecutan en el SM.

La memoria más extensa de la GPU es la memoria global. Es la memoria a la que se transfieren los datos desde la CPU, y se puede acceder desde cualquier hilo en ejecución. Su tiempo de acceso es muy alto, y la consecuencia es que, cuando un hilo lee o escribe un dato de o en ella, se vea suspendido hasta que termina el acceso. Es muy habitual que, cuando un *kernel* trabaja con grandes estructuras de datos que, por su tamaño, deben almacenarse en memoria global, si el acceso a esos datos es intenso (se realiza repetidas veces), se copie la parte con la que trabaja en cada momento a memoria compartida para reducir el tiempo de las lecturas y escrituras de esos datos. La cantidad de memoria global en una GPU también varía con

el modelo, y es uno de los factores que determina su precio. En la Tesla C1060 se trata de 4 GB, una cantidad grande en su momento, ya que está destinada a cálculo científico, donde es preciso manejar gran cantidad de datos.

El último tipo que se va a exponer en esta sección es la memoria constante. Se trata de una memoria de ámbito global pensada fundamentalmente para datos que son leídos por todos los hilos de un *kernel*, como constantes numéricas o tablas de referencia, etc. Sólo pueden ser modificados por la CPU, y, aunque en principio su tiempo de acceso es parecido al de la memoria global, tiene la ventaja de que es cacheable en el SM, por lo que, una vez se ha utilizado un dato, posteriores accesos a él son mucho más rápidos. La tarjeta Tesla C1060 dispone de 64 KB de memoria constante. La Tabla 2-5 resume toda esta sección. En el manual de referencia de CUDA [45] se puede encontrar información más detallada de los tipos de memoria de una GPU.

Tabla 2-5. Tipos de memoria.

Tipo	**Ámbito de acceso**	**Velocidad de acceso**
Registros	Hilo	Rápido
Compartida	Bloque	Medio
Local	Hilo	Lento
Global	*Kernel*	Lento
Constante	*Kernel*	Lento (cacheado)

2.3 Interfaz de programación.

2.3.1 Capas de software.

El entorno CUDA ofrece distintas alternativas a la hora de acceder a los recursos de la GPU. Estas alternativas toman la forma de capas de software, que constituyen una jerarquía de niveles, cada uno con más nivel de concreción y detalle a medida que se acercan a la capa física que constituye la GPU. La Figura 2-8 muestra gráficamente estas capas.

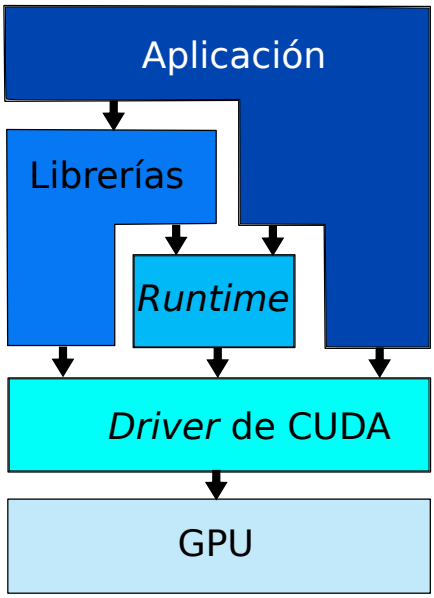


Figura 2-8. Capas de software de CUDA.

La capa más inferior (descontando la propia GPU) es el *driver* o controlador de dispositivo. Es la que accede directamente a la GPU, y ofrece una serie de funciones (una API o *Application Programming Interface*) al programador que tienen como característica que proporcionan el máximo nivel de detalle en el control de la GPU. El precio que se paga por este control más preciso es una mayor

cantidad de trabajo para llevar a cabo cada tarea. Puesto que tiene más opciones en cada operación, es necesario ser más específico con cada una de ellas. Sin embargo, no proporciona una mejora sensible del rendimiento. Éste depende casi exclusivamente de cómo estén programados los *kernels* (y la aplicación, en general).

La mayoría de programadores pueden encontrar mucho más interesante el llamado *runtime*, que es una biblioteca de funciones que proporcionan la funcionalidad necesaria para acceder a la GPU, pero sin pagar el alto precio de la complejidad del *driver*. Salvo que se necesite especificar detalles muy precisos de cómo deben funcionar los *kernels*, lo más aconsejable es recurrir al *runtime*. Por ejemplo, para lanzar un *kernel* en la GPU, es preciso crear un *contexto*, que es una estructura lógica que contiene toda la información necesaria para gestionar su ejecución en la GPU, como las zonas de memoria asignadas y otra información de este tipo. El *runtime* gestiona el contexto de un *kernel* de forma automática y transparente al programador, de forma que éste se puede centrar en diseñarlo de la forma más eficaz posible, sin tener que preocuparse de esos detalles. El programador que desee manejar varios contextos para varias aplicaciones, debe recurrir al *driver*. Pero el resto puede optar por la versión más simplificada del *runtime*.

La principal funcionalidad del *runtime* a la que va a recurrir un programador tiene que ver con la gestión de la memoria pero, además de esa, existen una gran variedad de funciones para otras tareas que van desde gestionar la GPU hasta obtener información de errores, pasando por la gestión de *streams* o de texturas. La Tabla 2-6 muestra un subconjunto de las funciones del *runtime*, las más usadas. En el Manual de Referencia de CUDA [47] se describen todas las que

hay de forma más completa. Las funciones se ejecutan desde la parte de la aplicación que corre en la CPU, e interactúan con la GPU desde ahí.

Tabla 2-6. Algunas funciones del *runtime*.

Tipo	**Nombre**	**Descripción**
Sistema	`cudaGetErrorString`	proporciona información sobre un código de error
	`cudaGetLastError`	proporciona el código del último error
	`cudaDeviceGetAttribute`	proporciona información sobre el dispositivo
	`cudaDeviceReset`	reinicializa el dispositivo
	`cudaChooseDevice`	busca un dispositivo que responde a las características que se indican
	`cudaGetDeviceCount`	proporciona el número de dispositivos válidos presentes en el sistema
	`cudaSetDevice`	selecciona el dispositivo sobre el que se va a trabajar
	`cudaGetDevice`	indica qué dispositivo se está utilizando
Gestión de *streams*	`cudaStreamCreate`	crea un *stream*
	`cudaStreamDestroy`	elimina un *stream*
	`cudaStreamQuery`	proporciona información sobre si una operación del *stream* ha terminado
	`cudaStreamSynchronize`	espera a que terminen las tareas pendientes de un *stream*

Tipo	Nombre	Descripción
Gestión de memoria	`cudaMalloc`	asigna memoria en la GPU
	`cudaMemcpy`	copia datos a/desde la memoria de la GPU
	`cudaFree`	libera memoria asignada en la GPU
	`cudaMemcpyAsync`	realiza una copia asíncrona de datos a/desde la GPU
	`cudaHostAlloc`	asigna memoria en la CPU
	`cudaFreeHost`	libera memoria asignada en la CPU
	`cudaMemset`	inicializa una zona de memoria de la GPU
	`cudaMemcpyToSymbol`	copia datos a la memoria de constantes de la GPU
	`cudaMallocPitch`	asigna memoria en la GPU con características especiales de alineamiento
	`cudaMalloc3D`	asigna memoria en la GPU (función específica para arrays de 3 dimensiones)
	`cudaMemcpy2D`	copia memoria (función específica para arrays de 2 dimensiones)
	`cudaMemcpy3D`	copia memoria (función específica para arrays de 3 dimensiones)

Por último, en algunos casos, los algoritmos que se quieren ejecutar en la GPU están contenidos en bibliotecas de código proporcionadas por algunos fabricantes. En estos casos, el programador de la aplicación puede abstraerse completamente de la

gestión de la GPU, y simplemente invocar las funciones de estas bibliotecas, que hacen el trabajo por él. Desgraciadamente no es una situación demasiado frecuente, y lo más habitual es que, en el mejor de los casos, el programador pueda utilizar estas funciones para aligerar su trabajo, pero deba completar su funcionalidad para terminar de adaptarlo a su aplicación.

2.4 Herramientas de programación.

2.4.1 Generación de código.

La principal herramienta para generación de código para tarjetas de NVIDIA en CUDA es el CUDA *Development Toolkit*, un paquete de herramientas de desarrollo que incluye un compilador, bibliotecas de funciones matemáticas y algunas herramientas para optimizar las aplicaciones. Está disponible en versiones para Windows, Linux y Mac OS X. En el momento de arrancar este trabajo, NVIDIA proporcionaba una herramienta de depuración gráfica muy potente (a la que se hace referencia más adelante) para la plataforma Windows, y eso decantó la decisión del entorno por esta plataforma.

El compilador proporcionado es capaz de generar código para la GPU, pero el código para la CPU debe ser generado por un compilador adecuado que no se proporciona. En el caso de Linux y Mac OS X, se recurre al compilador *gcc*, que es una herramienta de libre distribución y muy extendida. En el caso de Windows, el *Toolkit* requiere la presencia del entorno de compilación de Microsoft Visual Studio. Puesto que Microsoft dispone de una versión gratuita del mismo que es válida para el *Toolkit*, esto no plantea ningún problema.

El proceso de compilación de una aplicación para la GPU es algo más complejo que el de una aplicación normal. Hay que tener en cuenta que hay que generar código para dos (o más) dispositivos diferentes. El compilador de NVIDIA analiza la aplicación y decide qué parte se ejecuta en la CPU y qué parte en la GPU. El código de la GPU lo compila él directamente, y traslada el código para la CPU al compilador externo (*gcc* o Visual Studio, según la plataforma). Para permitir a la aplicación adaptarse lo más posible al dispositivo sobre el que corre, el compilador puede generar, en lugar de código binario, un código intermedio llamado PTX (*Parallel Thread eXecution*, ejecución paralela de hilos). Este código puede ser recompilado por el *driver* en el momento en que la aplicación se lanza a ejecutar, para poder adaptarlo a la tarjeta presente en ese momento. Este proceso se llama *Just-In-Time Compilation* (compilación de última hora). El compilador es una herramienta muy completa, y su funcionalidad se puede consultar en su Manual de Referencia [48].

2.4.2 Herramientas de depuración y análisis de rendimiento.

NVIDIA proporciona dos herramientas de depuración principales. La primera de ellas es una versión del tradicional depurador `gdb` de Linux, adaptada para poder manejar hilos. Se trata de una herramienta de interfaz de comandos, y disponible únicamente para entornos Linux y Mac OS X.

La segunda es un entorno gráfico de depuración llamado NSight. En el momento de iniciar este trabajo, estaba disponible únicamente para la plataforma Windows, ya que se añadía al entorno de desarrollo Visual Studio (al igual que hacía el compilador), pero

actualmente existe una versión para Linux y Mac OS X basada en el entorno de desarrollo Eclipse.

NSight permite depurar las aplicaciones sobre la propia GPU. Se trata de una herramienta muy potente, ya que con ella es posible depurar la ejecución de un *warp* específico dentro del *kernel*, paralelizando si es necesario la ejecución del resto de *warps* independientemente de en qué SM se estén ejecutando. Esto es muy útil, ya que, en ocasiones los errores del programa no se manifiestan en todos los hilos (por ejemplo, al manejar arrays de varias dimensiones, no es infrecuente cometer errores con los índices, lo que puede provocar que algunos hilos accedan a zonas de memoria que se salen del array), y con esta opción es posible detectar fácilmente los errores en los hilos donde ocurren.

Además, NSight permite realizar análisis del rendimiento de una aplicación, mostrando el tiempo que se emplea en cada *kernel* y en cada llamada a una función del *runtime*. De esta manera, encontrar los posibles cuellos de botella, o elegir objetivos para optimizar es mucho más fácil.

Otra herramienta interesante en su día, aunque actualmente su funcionalidad está prácticamente cubierta por NSigh, era la utilidad `computeprof`. Realiza la ejecución de una aplicación, y contabiliza los tiempos empleados en cada parte de la misma, así como el número de invocaciones a distintas funciones que la forman.

También existen herramientas creadas por terceros. Por ejemplo, Allinea dispone de DDT, un depurador para aplicaciones paralelas. Es compatible con MPI y con OpenMP, y puede operar en *clusters* con múltiples CPUs y GPUs. Está disponible para plataformas Linux. Otro ejemplo es TotalView, de RogueWave. Igualmente

admite programas para MPI y OpenMP además de CUDA, y las aplicaciones pueden estar escritas en C, FORTRAN y algún lenguaje más. TotalView puede correr sobre Linux, Mac OS X y algunas versiones propietarias de Unix.

2.4.3 Bibliotecas de funciones.

Las bibliotecas de funciones permiten, en ocasiones, no tener que meterse en el mundo de CUDA para utilizar la GPU. En otras, esto no es posible evitarlo, pero aún así facilitan la labor de creación de las aplicaciones. NVIDIA proporciona una serie de bibliotecas con las funcionalidades más necesarias, pero otras empresas han empezado a ofrecer otras bibliotecas para casos más particulares. La Tabla 2-7 lista algunas de las bibliotecas más usadas. Para este trabajo ha sido necesario utilizar la biblioteca cuBLAS, que es una adaptación a CUDA del paquete de rutinas BLAS [49] (Basic Linear Algebra Subroutines), que permite realizar operaciones de álgebra lineal básicas (productos de matrices por matrices, matrices por vectores), y la biblioteca CULA Dense [50] (que es una adaptación de las rutinas de LAPACK [51] (Linear Algebra Package, funciones de álgebra lineal algo más avanzadas, como diagonalización de matrices, solución de sistemas lineales, etc). Más adelante se explica qué funciones se han utilizado y la funcionalidad que tienen.

Tabla 2-7. Bibliotecas de funciones para CUDA

Biblioteca	**Fabricante**	**Descripción**
cuFFT	NVIDIA	Transformadas de Fourier rápidas
cuBLAS	NVIDIA	Rutinas de BLAS (Basic Linear Algebra Subroutines)
cuSPARSE	NVIDIA	Rutinas de álgebra lineal para matrices dispersas
cuRAND	NVIDIA	Generación de números aleatorios
NPP	NVIDIA	Primitivas de funciones para proceso de imagen y video
Thrust	NVIDIA	Plantillas para algoritmos paralelos
CUDA Math Library	NVIDIA	Conjunto de funciones matemáticas y estadísticas
CULA Dense	EM Photonics	Rutinas de LAPACK (Linear Algebra Package) para matrices densas
CULA Sparse	EM Photonics	Rutinas de LAPACK (Linear Algebra Package) para matrices dispersas
MAGMA	Magma project	Rutinas de LAPACK (Linear Algebra Package)
ArrayFire	Accelereyes	Manejo de arrays

2.5 OpenCL.

La alternativa a CUDA más prometedora en la actualidad es OpenCL. Se trata de una plataforma de desarrollo diseñada con el objetivo de crear aplicaciones que se ejecuten en entornos heterogéneos compuestos por elementos de proceso diversos, como CPUs, GPUs, procesadores digitales de señal, etc. Se centra fundamentalmente en conseguir el aprovechamiento de todos los elementos del sistema para la ejecución del programa de la manera más eficaz. El programa puede identificar los diferentes elementos de proceso de la máquina (CPUs, GPUs, etc) y definir qué partes del mismo se ejecutan en cuáles de ellos.

La cara de OpenCL es que permite que las aplicaciones no estén ligadas a un fabricante, ya que el código desarrollado en el lenguaje OpenCL puede ser compilado para y ejecutado en cualquier tipo de plataforma. Así, es posible, por ejemplo, desarrollar una aplicación para una tarjeta de NVIDIA y un tiempo después cambiar a otra más potente de ATI sin demasiadas dificultades. Si se extiende el uso de OpenCL, la consecuencia probable será una mayor competencia en cuanto a prestaciones y precio entre los fabricantes de GPUs en particular y de sistemas de computación en general.

La cruz es que el rendimiento que puede ofrecer una plataforma como CUDA, pensada exclusivamente para una familia de dispositivos que además son fabricados por los mismos que diseñan CUDA, es mayor. Si la aplicación en cuestión necesita de la máxima potencia posible, entonces CUDA es la elección más adecuada. El objetivo de esta sección es presentar similitudes y diferencias entre CUDA y OpenCL, y explicar las razones por las que, para el trabajo descrito en esta memoria, se eligió CUDA.

2.5.1 Características de OpenCL.

La característica principal de OpenCL es que busca ser adaptable a cualquier sistema de computación. Cuando una aplicación se ejecuta, inicialmente realiza una identificación del sistema en el que corre, en la que se adquiere información acerca de las plataformas y los dispositivos de esas plataformas presentes en el equipo. También se definen los contextos, que no son más que agrupaciones de dispositivos que se usarán para la resolución de un *kernel*.

Una plataforma está constituida por uno o varios dispositivos, cada uno de ellos con sus capacidades características, tales como unidades de ejecución, memoria, etc. A su vez, las plataformas se caracterizan por su tipo de dispositivo (CPU, GPU, etc), el fabricante, la versión de OpenCL que puede ejecutar, etc. OpenCL dispone de funciones que permiten averiguar el número y las características de las plataformas presentes en el sistema, así como las características de los dispositivos de cada plataforma. Con estos datos se forman los contextos, mediante los cuales el programador define qué dispositivos agrupa para cada operación. En OpenCL, un *kernel* se ejecuta en un contexto, y para cada contexto se crea una cola a la que van llegando los *kernels* que deben ejecutarse en ese grupo de dispositivos.

Tanta complejidad no es necesaria en CUDA, que es un entorno más homogéneo. En CUDA, la aplicación puede obtener información sobre el sistema sobre el que corre, para averiguar el número de GPUs y sus respectivas características. Y con la compilación de última hora puede adaptarse a las GPUs presentes en el sistema. OpenCL, por su parte, tiene más requisitos en este respecto, ya que sus objetivos también son más ambiciosos. Al contrario que CUDA,

pensado exclusivamente para tarjetas GPU de NVIDIA, OpenCL puede trabajar con más tipos de dispositivos de cálculo con características más diversas.

Modelo de ejecución.

En OpenCL se consigue la escalabilidad de una forma parecida a la que usa CUDA. Para gestionar una gran cantidad de hilos sin dedicar demasiado silicio a estructuras de control, estos se agrupan en dos jerarquías. Las rutinas que se ejecutan en los contextos (también llamadas *kernels*) se organizan en un primer nivel en una rejilla llamada *NDRange*, que puede tener hasta 3 dimensiones. Los elementos de un *NDRange* se distribuyen en *WorkGroups* (grupos de trabajo), que a su vez se organizan en otra rejilla de hasta 3 dimensiones, cuyos elementos se denominan *WorkItems* (elementos de trabajo). Los elementos de trabajo del mismo grupo pueden sincronizarse y compartir datos.

Jerarquía de memoria.

OpenCL ofrece al programador una jerarquía virtual de memoria. En tiempo de ejecución se establecen las correspondencias necesarias entre los niveles virtuales y los físicos. Con esto se pretende que el programador no tenga que estar al corriente de todos los detalles de la memoria de los dispositivos. Los niveles de la jerarquía son muy parecidos a los que existen en CUDA: existe una memoria global que comparten todos los elementos de trabajo, y que es muy extensa y lenta. Con características parecidas existe la memoria de constantes, que se utiliza para constantes numéricas y datos que todos los elementos de trabajo leen a la vez (por ejemplo, si todos leen a la vez

el mismo elemento de un array). También existe una memoria local que pueden compartir los elementos de trabajo del mismo grupo. Y por último, la memoria privada es exclusiva de cada elemento de trabajo. La Tabla 2-8 muestra las correspondencias entre CUDA y OpenCL en lo que respecta a los niveles de la jerarquía de memoria. Como se puede ver, son los mismos. La única salvedad es que, si en un futuro los dispositivos incorporasen algún tipo adicional de memoria, las aplicaciones de OpenCL no necesitarían modificarse, ya que la compilación de última hora (*just-in-time compilation*) podría adaptar las correspondencias entre los niveles de la jerarquía virtual y la física para tenerlos en cuenta. CUDA, por su parte, debería incorporar esos niveles a su entorno de programación y a su modelo de ejecución.

Tabla 2-8. Tipos de memoria en CUDA y OpenCl

CUDA	**OpenCL**
Global	Global
Compartida	Local
Local	Privada
Constante	Constante

Programación.

La forma de construir los programas es parecida a la que se da en CUDA si se utiliza el *driver* en lugar de la biblioteca de tiempo de ejecución (el *runtime*). Es preciso crear un contexto, y dentro de él, una cola de comandos para cada dispositivo. El contexto sirve para coordinar la interacción entre host y dispositivos. Las colas de

comandos, al estar dedicadas cada una a uno de los dispositivos, requieren sondear el sistema para encontrar los que existen, analizar sus características, etc. Este proceso es farragoso y pesado, y en CUDA se puede evitar si se usa el *runtime*.

Herramientas de desarrollo.

AMD proporciona un entorno de desarrollo para OpenCL llamado CODEXL. Contiene un editor, un compilador y un potente depurador que permite analizar la ejecución de los *kernels* en la GPU en tiempo real. Además, también realiza análisis de rendimiento (*profiling*) en ejecución y en estático. Está disponible para plataformas Windows y Linux. En el caso de Windows, se integra en el entorno de Visual Studio.

OpenCL Studio es otro entorno de desarrollo, en este caso proporcionado por la empresa Geist Software Labs, Inc. En este caso, no se proporciona demasiada información sobre el producto, por lo que no ha sido posible estudiarlo más en detalle.

Bibliotecas de terceros.

El "ecosistema" de OpenCL realmente no ha terminado de despegar aún, y en lo que respecta a bibliotecas de rutinas de terceros, CUDA tiene considerable ventaja. Para OpenCL existe clAmdBlas, una adaptación de BLAS para OpenCL, proporcionada por AMD, y clAmdFft, una biblioteca de transformadas de Fourier rápidas. Además, está en proceso una versión de MAGMA para OpenCL, llamada clMAGMA. MAGMA (*Matrix Algebra on GPU and Multicore Architectures*, álgebra matricial en GPUs y arquitecturas *multicore*), que ya se ha mencionado anteriormente en

la sección de herramientas para CUDA, es una adaptación de LAPACK para sistemas con múltiples CPUs y GPUS. En la versión para OpenCL no están incluidas todas las rutinas de LAPACK, pero sí algunas de las más comunes, como, por ejemplo, las factorizaciones LU, QR y Cholesky, reducciones, transformaciones ortogonales y soluciones a problemas de autovalores y valores singulares.

A pesar de la desventaja en este campo que presenta OpenCL, se puede esperar que el entorno vaya evolucionando favorablemente, habida cuenta de la importancia de las empresas que están detrás de él (Apple, Intel, AMD, ARM, Qualcomm, TexasInstruments, etc).

Razones para la elección de CUDA frente a OpenCL.

Como se ha mencionado en secciones anteriores, la principal razón para elegir CUDA frente a OpenCL fue el estado más avanzado de desarrollo del conjunto de herramientas disponibles en el momento de tomar la decisión. NVIDIA ha venido realizando un esfuerzo considerable por dotar a los desarrolladores del mejor software posible, lo que ha conferido a CUDA una notable ventaja que las empresas que están detrás de OpenCL aún están intentando compensar. En concreto, la existencia de un depurador avanzado como es NSight facilita considerablemente la tarea de corregir los inevitables errores que se producen en el desarrollo de las aplicaciones. Esta tarea se vuelve terriblemente ardua sin una utilidad de este tipo, y es una buena noticia que haya algo parecido ahora para OpenCL, pero este no era el caso al inicio del trabajo descrito en esta memoria.

Otra de las razones que han desequilibrado la balanza en favor de CUDA ha sido la existencia de bibliotecas de funciones matemáticas.

Ya que esta tesis consiste precisamente en adaptar una serie de métodos numéricos para el análisis electromagnético a una nueva arquitectura computacional, el hecho de poder utilizar estas funciones ha permitido avanzar más en el procedimiento. De nuevo, en el momento de la toma de decisión, el entorno CUDA estaba más desarrollado, lo que ha sido un factor más en favor de esta plataforma.

CUDA presenta algún aspecto negativo. El principal es la exclusividad. Las aplicaciones desarrolladas en CUDA sólo sirven con tarjetas de NVIDIA, mientras que OpenCL hace posible su implantación en un conjunto mayor de ellas. Los fabricantes de GPUs son fundamentalmente NVIDIA y AMD, y es difícil determinar cuál de ellos fabrica la tarjeta más potente en cada momento. Utilizar CUDA limita las posibilidades a tarjetas de NVIDIA, mientras que OpenCL permite trabajar con ambos fabricantes. Ambos mejoran continuamente sus productos, y la mejor tarjeta generalmente es la más reciente. En este sentido, tener la posibilidad de cambiar de fabricante sobre la marcha es un gran punto a favor de OpenCL

En cuanto a la comparación de rendimientos entre CUDA y OpenCL, existen estudios al respecto que parecen darle cierta ventaja a CUDA, como [52] y [53]. Pero hay que tener en cuenta que las pruebas están realizadas sobre GPUs de NVIDIA (la única posibilidad, ya que CUDA no se puede utilizar con tarjetas AMD), a las que CUDA está naturalmente mejor adaptado. NVIDIA tiene mucho interés en CUDA, y está desarrollando un gran esfuerzo en esta línea, por lo que la comparación con la versión de OpenCL para

tarjetas NVIDIA, también desarrollado por NVIDIA, no se realiza en igualdad de condiciones..

A la vista de todos estos elementos de juicio, la decisión final tomada fue optar por CUDA. Las ventajas con respecto a las herramientas de desarrollo y las bibliotecas de código ya existentes fueron el factor principal en esta decisión. Es cierto que, en la práctica, se asume un compromiso de exclusividad con las tarjetas de NVIDIA, pero resulta difícil, como se ha explicado, decidir qué familia proporciona mejor rendimiento bruto, por lo que esta exclusividad no es un factor decisivo, en cualquier caso.

3 El Método de las Funciones Base Características.

En este capítulo se expone el método numérico que se pretende implementar en la GPU, el Método de las Funciones Base Características (CBFM). Se trata de un método cuyo objetivo principal es reducir el número de incógnitas que surgen al aplicar otro método de análisis numérico preexistente, el llamado Método de los Momentos (MoM), para hacer factible aplicarlo a superficies mucho mayores que, de otro modo, resultarían en un problema no abarcable computacionalmente en la actualidad.

El capítulo comienza con una breve descripción del Método de los Momentos, y se incluye una discusión sobre la representación geométrica de la superficie bajo estudio. A continuación, se pasa a describir el Método de las Funciones Base Características. En el caso de una superficie de tamaño arbitrario, y para hacer el problema tratable computacionalmente, es preciso dividirla en porciones o bloques, de forma que al computador le sea posible procesarla por partes. Por ello, la última sección aborda el procedimiento de la división de la geometría que se estudia, y cómo ésta afecta al Método de las Funciones Base Características.

3.1 El Método de los Momentos.

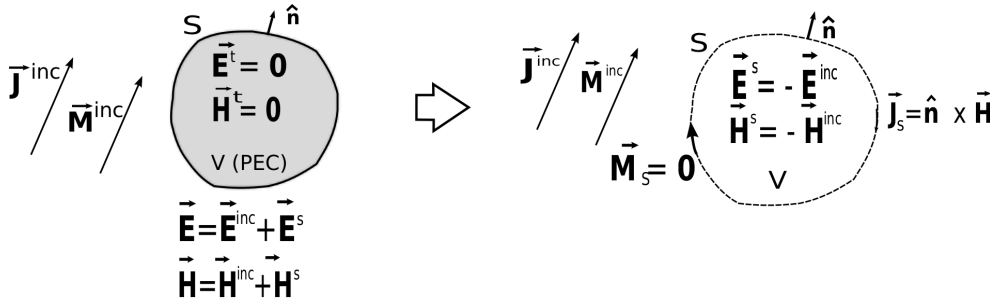
El Método de los Momentos (MoM, *Method of Moments*) [54] es una de las técnicas más utilizadas a la hora de analizar problemas de radiación y dispersión electromagnéticas. Los métodos que tratan estos problemas se pueden clasificar de acuerdo a diferentes criterios, algunos de ellos expuestos en el apartado 1.1. Si se atiende a la

longitud de onda de la radiación, el MoM es un método de baja frecuencia, ya que es aplicable cuando la longitud de onda de la radiación es comparable al tamaño del objeto. En cuanto a la forma de aplicar las ecuaciones de Maxwell, el MoM entra dentro de la categoría de los métodos integrales. Por último, con respecto a las técnicas electromagnéticas empleadas, el MoM es un método basado en la estimación de la corriente, puesto que, para su formulación numérica, se muestrea la corriente sobre la superficie del objeto utilizando una serie de funciones base.

3.1.1 Ecuaciones Integrales de Campo.

Una de las utilidades del MoM, y el que se le da en este trabajo, es el cálculo de la sección radar (RCS, *Radar Cross-Section*) de un objeto. Sobre el objeto incide una onda plana, y se trata de obtener la radiación creada en un punto del espacio como consecuencia de la incidencia de dicha onda sobre el objeto. Dos son las principales suposiciones del problema: por un lado, con respecto al objeto, que se trata de un conductor eléctrico perfecto (PEC, *Perfect Electrical Conductor*), y por otro, para poder aplicar el MoM, que la longitud de onda de la radiación incidente es comparable a la dimensión del objeto.

El análisis matemático del problema parte de las ecuaciones de Maxwell. Al aplicarlas, se hace uso del principio de equivalencia, según el cual, en el análisis, a efectos de campo eléctrico, se puede sustituir el objeto por la suma de espacio libre en el lugar que ocupa más una serie de corrientes equivalentes que tienen el mismo efecto en el entorno que el objeto al que sustituyen. La Figura 3-1 muestra esto mismo de forma gráfica.



a) medio original b) espacio libre

Figura 3-1. Principio de Equivalencia: el medio original se sustituye por espacio libre añadiendo corrientes equivalentes

En la parte a) de la Figura 3-1 se representa la situación original: unas fuentes de radiación externas, $\mathbf{J}^{inc}$ y $\mathbf{M}^{inc}$, y un objeto tipo PEC. En su interior, los campos son nulos, y en el exterior, son la suma de dos contribuciones: por un lado, el campo incidente que generan las fuentes externas (con el superíndice "*inc*"), y por otro el campo dispersado por el objeto (superíndice "*s*", del inglés '*scattered*'). La componente tangencial del campo eléctrico en la superficie del objeto es nula, como lo es la densidad de corriente magnética. La densidad de corriente eléctrica sobre la superficie del objeto está determinada por la componente tangencial del campo magnético: $\mathbf{J}_S=\mathbf{n}$ x $\mathbf{H}$.

En la parte b) de la misma figura, al aplicar el principio de equivalencia, el objeto se sustituye por unas corrientes que crean el mismo efecto en el exterior. Estas corrientes permiten calcular el campo que dispersa el objeto en el exterior del mismo. A partir de este punto, es posible obtener las ecuaciones integrales generales para los campos **E** y **H**. Estas ecuaciones reciben el nombre de ecuación

integral de campo eléctrico (EFIE, *Electric Field Integral Equation*) y ecuación integral de campo magnético (MFIE, *Magnetic Field Integral Equation*):

$$\vec{E}_{inc}(\vec{r}) = \frac{j\omega\mu}{4\pi}\iint_S \vec{J}_S(\vec{r}')G(\vec{r},\vec{r}')dS' + \frac{j}{4\pi\omega\varepsilon}\nabla\iint_S \nabla'\cdot\vec{J}_S(\vec{r}')G(\vec{r},\vec{r}')dS' \quad (3.1a)$$

$$\vec{J}_S = \hat{n}\times\vec{H}_{inc}(\vec{r}) + \hat{n}\times\left[\frac{1}{4\pi}\iint_S \nabla G(\vec{r},\vec{r}')\times\vec{J}_S(\vec{r}')ds'\right] \quad (3.1b)$$

La Ecuación (3.1a) es la EFIE. Relaciona el campo que incide en el objeto conductor perfecto ($\mathbf{E}_{\text{inc}}$) con la corriente ($\mathbf{J}_{\text{S}}$) que se crea en su superficie (S). En la ecuación, $\mathbf{r}$ es el punto de observación, y $\mathbf{r}'$ el punto fuente de campo. El operador nabla primado y sin primar se define respecto a las coordenadas de fuente y de observación, respectivamente. G($\mathbf{r}$,$\mathbf{r}$') es la función de Green en espacio libre, y se define:

$$G(\vec{r},\vec{r}') = \frac{e^{-jK|\vec{r}-\vec{r}'|}}{|\vec{r}-\vec{r}'|} \quad (3.2)$$

Por su parte, la ecuación (3.1b) es la MFIE, y relaciona el campo magnético incidente ($\mathbf{H}_{\text{inc}}$) con la corriente ya mencionada $\mathbf{J}_{\text{s}}$. La EFIE y la MFIE se catalogan como ecuaciones integrales de Fredholm de primera y segunda especie, respectivamente.

Resolver estas ecuaciones proporciona las corrientes inducidas en el objeto, y, a partir de ellas, se puede obtener el campo en cualquier punto del espacio exterior al mismo como suma de dos contribuciones: la fuente original y el campo dispersado por el objeto:

$$\vec{E}(\vec{r}) = \vec{E}_{inc}(\vec{r}) + \vec{E}_{s}(\vec{r}), \tag{3.3a}$$

$$\vec{H}(\vec{r}) = \vec{H}_{inc}(\vec{r}) + \vec{H}_{s}(\vec{r}). \tag{3.3b}$$

El trabajo desarrollado en esta tesis se centra en la EFIE, por lo que el punto de partida son las ecuaciones (3.1a) y (3.3a).

3.1.2 El Método de los Momentos.

El MoM está descrito en detalle en el libro de R.F. Harrington correspondiente a la referencia [54], y en esta sección se presenta un breve resumen. Se trata de un método que se utiliza para facilitar la resolución de las ecuaciones integrales que se obtuvieron en la sección anterior, las ecuaciones (3.3a) y (3.3b), en un conjunto de ecuaciones algebraicas. El procedimiento de trabajo se resume a continuación. El punto de partida es una expresión de la forma:

$$L(F) = Y \tag{3.4}$$

donde L es un operador lineal que transforma una función incógnita F en otra Y, que es conocida. En nuestro caso, L sería el operador integro-diferencial aplicado a la densidad de corriente $\mathbf{J}_S$, de la ecuación (3.1a), que haría el papel de F. Por su parte, Y sería el campo incidente $\mathbf{E}_{inc}$. El MoM convierte la expresión de (3.4) en otra de la forma:

$$\sum_{n=1}^{N} I_n L(B_n) = Y \tag{3.5}$$

En (3.5) se ha realizado una expansión de F con respecto a una serie de funciones base conocidas, B_n. I_n son los coeficientes de esa expansión:

$$F = \sum_{n=1}^{N} I_n B_n \tag{3.6}$$

Para que la representación de F fuera exacta, sería necesario que N=∞, pero un N finito permite alcanzar una solución numérica, a costa de perder precisión. Si se elige adecuadamente N, esta pérdida puede ser pequeña.

El siguiente paso del MoM consiste en utilizar un conjunto de N funciones W_i, denominadas funciones prueba, y realizar el producto interno de estas funciones con cada uno de los lados de la ecuación (3.5):

$$\sum^{N} I_n \langle W_m, L(B_n) \rangle = \langle W_m, Y \rangle, \qquad m = 1,..,N \tag{3.7}$$

El producto interno se define:

$$\langle f, g \rangle = \int_D f(x) g^*(x) dx \tag{3.8}$$

con g*(x) el complejo conjugado de g(x), de modo que los dos miembros de la igualdad (3.7) serían:

$$\left\langle W_j(\vec{r}), L(B_i(\vec{r})) \right\rangle = \int_D W_j(\vec{r}) \cdot L^*(B_i(\vec{r})) d\vec{r}, \tag{3.9a}$$

y

$$\left\langle W_j(\vec{r}),Y(\vec{r})\right\rangle = \int_D W_j(\vec{r})\cdot Y^*(\vec{r})d\vec{r}. \tag{3.9b}$$

La ecuación (3.7) representa una expresión matricial de la forma:

$$[Z][J]=[V] \tag{3.10}$$

si se define [Z] como:

$$[Z]=\begin{bmatrix} \langle W_1,L(B_1)\rangle & \langle W_1,L(B_2)\rangle & \cdots & \langle W_1,L(B_N)\rangle \\ \langle W_2,L(B_1)\rangle & \langle W_2,L(B_2)\rangle & \cdots & \langle W_2,L(B_N)\rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle W_N,L(B_1)\rangle & \langle W_N,L(B_2)\rangle & \cdots & \langle W_N,L(B_N)\rangle \end{bmatrix} \tag{3.11}$$

[J] como:

$$[J]=\begin{bmatrix} I_1 \\ I_2 \\ \vdots \\ I_N \end{bmatrix} \tag{3.12}$$

y [V] como:

$$[V]=\begin{bmatrix} \langle W_1,Y\rangle \\ \langle W_2,Y\rangle \\ \vdots \\ \langle W_N,Y\rangle \end{bmatrix} \tag{3.13}$$

Una vez planteada la ecuación (3.10), lo único que queda es obtener el vector de coeficientes [J], que, si [Z] es no singular, supone calcular:

$$[J] = [Z]^{-1}[V] \tag{3.14}$$

El método empleado para resolver (3.14) puede variar, y más adelante se exponen varios de los más utilizados.

3.1.3 Discretización. Funciones base y funciones prueba.

El paso de la ecuación (3.4) a la ecuación (3.5) supone una discretización del problema. Las funciones base permiten muestrear las incógnitas, mientras que las funciones de prueba aseguran que se cumplen las condiciones de contorno en puntos determinados del objeto bajo estudio. Los puntos en los que se evalúan estas funciones vienen determinados por la representación geométrica que se haga del objeto. En este apartado se van a tratar las cuestiones relacionadas con la discretización del problema. En primer lugar, se expone la representación geométrica del objeto. A continuación, se seleccionan las funciones base. Por último, se describen las funciones de prueba. Al tratar ambos tipos de función, se analiza cómo quedan transformadas las ecuaciones que describen el problema.

Representación de la geometría.

Dado que la representación de la geometría no es uno de los objetivos de esta tesis, este apartado es únicamente una exposición resumida de cómo se representa la geometría en ella. El grupo de investigación dentro del que se ha desarrollado esta tesis ha estudiado extensamente el problema (consultar [55] y [56]), por lo que, para este trabajo, adoptamos el método que utiliza el grupo.

Para representar matemáticamente un objeto, el punto de partida es una serie de puntos de su superficie elegidos estratégicamente, de

forma que definan las características espaciales del mismo que interesen para el análisis. A continuación se seleccionan una serie de superficies o parches (o bien curvas, según la dimensión del objeto) que recubren el objeto utilizando estos puntos. Existen dos grandes familias de métodos de representación, en función de la forma de recubrir el objeto: la interpolación y la aproximación. En la primera, lo que se busca es que las superficies o las curvas matemáticas que se utilizan para representar el objeto pasen por los puntos que lo definen. De esta forma, la representación es precisa, pero no es posible garantizar una transición suave entre unas superficies o curvas y otras. En la segunda, las superficies o las curvas no pasan por todos los puntos que definen el objeto, aunque se acercan a todos en conjunto, y lo que se busca es que se produzcan transiciones suaves y continuas entre los puntos. En este trabajo se opta por esta segunda opción.

Si el objeto es un volumen, para representarlo por aproximación existen varios métodos. Entre ellos, por ejemplo, el modelo de alambres (*wireframes*) o el modelo de fronteras (*boundary representation*). En el modelo de alambres, el objeto se define por medio de curvas y no por superficies. Para cada una de ellas es suficiente con especificar los puntos que representan su vértice inicial y su vértice final. De esta manera se reduce la cantidad de información necesaria para describir el objeto. La consecuencia negativa viene derivada de este mismo hecho: un mismo modelo puede representar varios objetos. Es decir, los modelos de alambres son *ambiguos*. En el modelo de fronteras el objeto se representa por una serie de parches que recubren el objeto completamente. La cantidad de información necesaria para representar el objeto es

mayor, ya que se necesita la descripción de cada uno de los parches, y además, de la forma en que se unen unos parches con otros, es decir, la información de topología. En este trabajo se opta por el modelo de fronteras.

A la hora de describir los parches que recubren el objeto, se pueden utilizar superficies planas y superficies paramétricas. La primera opción tiene la clara ventaja de que para describir el parche únicamente es necesario identificar sus vértices. Sin embargo, existen objetos con una superficie difícil de representar por medio de estos parches, y que requieren una gran cantidad de ellos para obtener una aproximación aceptable. Si se utilizan superficies paramétricas, es posible describir el objeto con la misma precisión, pero con un número de parches mucho menor. Teniendo en cuenta que las superficies paramétricas también describen parches planos, ya que éstos son un caso particular de aquellas, en nuestro caso hemos optado por ellas.

Curvas de Bézier.

Las superficies paramétricas que se utilizan en esta tesis para describir los objetos son los llamados parches NURBS (*Non-Uniform Rational B-Splines*, parches B-Spline racionales no uniformes). Una descripción más detallada de todo lo expuesto en este apartado se puede encontrar en la tesis doctoral del profesor Eliseo García [57]. El origen de estas construcciones matemáticas está en las curvas de Bézier [58], [59], creadas para su uso en sistemas de diseño geométrico asistido por computador (*Computer Aided Geometric Design*). Una curva de Bézier, según el formalismo de Forrest, se expresa en función de las bases de Bernstein de la siguiente forma:

$$\vec{C}(t) = \sum_{i=0}^{n} \vec{b}_i B_i^n(t), \qquad 0 \le t \le 1 \qquad (3.15)$$

donde n es el grado de la curva, y $\mathbf{b}_i$ son los llamados puntos de control. $B_i^n(t)$ son las bases de Bernstein [60], definidas por:

$$B_i^n(t) = \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i} \qquad (3.16)$$

Los puntos de control forman el polígono de control de la curva, es decir, el conjunto de puntos a los que se debe aproximar. La relación entre los puntos $\mathbf{b}_i$ y la curva se puede ver en la Figura 3-2. La curva se aproxima a los puntos lo que se necesite, en función de la precisión deseada, pero no se garantiza que pase por ellos, salvo en el caso del inicial y el final. El dominio de la curva es el segmento del espacio paramétrico entre 0 y 1.

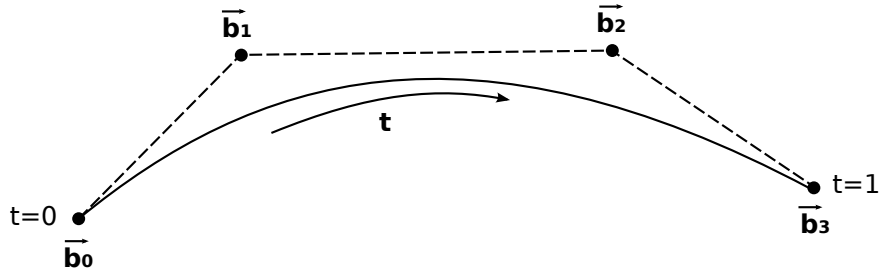


Figura 3-2. Curva de Bézier

Dada la importancia que tienen las cónicas, y la dificultad de su representación con las curvas de Bézier, se crea un nuevo tipo de éstas últimas a partir de la definición básica de la siguiente manera:

$$\vec{C}(t) = \frac{\vec{P}(t)}{W(t)} = \frac{\sum_{i=0}^{n} w_i \vec{b}_i B_i^n(t)}{\sum_{i=0}^{n} w_i B_i^n(t)} \qquad (3.17)$$

Este nuevo tipo de curva se denomina curva racional de Bézier [59]. En la definición, los valores w_i se llaman pesos. Están asociados a cada punto de control, y su función es "atraer" o "repeler" la curva hacia o desde el punto correspondiente. Si los pesos son todos 1, la curva racional de Bézier se convierte en una curva de Bézier a secas, por lo que éstas últimas son, realmente, un caso particular de aquellas.

Non-Uniform B-Splines (curvas B-Spline racionales no uniformes).

Las curvas racionales de Bézier tienen como limitación principal que modificar uno de los puntos de control afecta a toda la curva. Se dice que no tienen "control local". Por otro lado, si el objeto que se modela tiene una forma compleja, el grado de la curva se puede hacer demasiado grande para ser tratable.

Para paliar estas limitaciones se usan las curvas B-Spline [59], [61], [62]. Se trata de curvas de Bézier compuestas, por lo que las curvas de Bézier son un caso particular de las mismas. Su expresión es:

$$\vec{C}(t) = \sum_{i=1}^{n} \vec{d}_i N_i^k(t), \qquad 0 \le a \le t \le b \le 1,\ 2 \le k \le n+1 \qquad (3.18)$$

Los $\mathbf{d}_i$ definen el polígono de control de la curva. Los $N_i(t)$ son las bases B-Spline, y k es el grado de la curva. La definición de la curva

se completa con un conjunto de valores del dominio paramétrico de la misma, (el intervalo [a,b]), y que forman el llamado vector de nudos de la curva [57]. Lo habitual, y ese es también el caso de este trabajo, es que el intervalo paramétrico sea el intervalo [0,1]. Puesto que se trata de curvas compuestas, es preciso determinar en qué parte de la curva global influye cada una de las curvas componentes. Esta es la misión del vector de nudos, especificar dónde empieza y dónde termina, dentro del dominio paramétrico, cada una de las curvas componentes. El vector de nudos es una secuencia creciente de números reales de la forma:

$$\left\{t_i\right\}_{i=1}^{n+k} \tag{3.19}$$

A partir de la definición del vector de nudos, las bases N_i se definen:

$$N_i^1 = \begin{cases} 1, & si \quad t_i \le t \le t_{i+1} \\ 0 & en \ otro \ caso \end{cases}$$

$$N_i^k(t) = \frac{(t - t_i)N_i^{k-1}(t)}{t_{i+k-1} - t_i} + \frac{(t_{i+k} - t)N_{i+k}^{k-1}(t)}{t_{i+k} - t_{i+1}} \tag{3.20}$$

Es decir, cada nudo del vector define en qué porción del dominio paramétrico de la curva está activa la base correspondiente. En un punto dado del dominio paramétrico, sólo las bases activas determinan el aspecto global de la curva. Si los nudos de una curva se encuentran equiespaciados en el dominio paramétrico, las curvas se denominan uniformes. En caso contrario, se llaman no uniformes. Las curvas no uniformes permiten un diseño más elástico.

Análogamente a lo que ocurría entre las curvas de Bézier y las curvas racionales de Bézier, a partir de la definición de las curvas B-Spline se puede generalizar para definir las curvas B-Spline racionales [63]. La expresión de las mismas es:

$$\vec{C}(t) = \frac{\sum_{i=1}^{n} w_i \vec{d}_i N_i^k(t)}{\sum_{i=1}^{n} w_i N_i^k(t)}, \qquad 0 \le a \le t \le b \le 1,\ 2 \le k \le n+1 \qquad (3.21)$$

De nuevo aparecen aquí los pesos w_i, con la misma misión que en las curvas racionales de Bézier, y al igual que con las curvas B-Spline, existe un vector de nudos, con una función análoga. Las curvas B-Spline racionales no uniformes se conocen por su acrónimo: curvas NURBS (*Non-Uniform Rational B-Splines*).

Superficies.

Paralelamente a los modelos para curvas, se pueden desarrollar los modelos para superficies. El punto de partida para ellos son las superficies de Bézier [59], que se pueden entender como una extensión de las curvas de Bézier. A partir de ellas, se construyen, análogamente a como ocurría con las curvas, las superficies racionales de Bézier, que también se definen en función de los polinomios de Bernstein de la siguiente manera:

$$\vec{r}(u,v) = \frac{\sum_{i=0}^{m}\sum_{j=0}^{n} w_{ij}\vec{b}_{ij}B_i^m(u)B_j^n(v)}{\sum_{i=0}^{m}\sum_{j=0}^{n} w_{ij}B_i^m(u)B_j^n(v)} = \frac{\vec{P}(u,v)}{W(u,v)} \qquad (3.22)$$

En la definición se usan dos coordenadas paramétricas, u y v. Además, m y n determinan el grado de la superficie en cada una de ellas. La generalización a dos dimensiones del polígono de control es la malla de control, formada por una serie de puntos, $\mathbf{b}_{ij}$, cada uno con su peso, w_{ij}. En nuestro caso, las coordenadas paramétricas se encuentran entre los valores 0 y 1. De la definición se puede deducir una propiedad interesante: las curvas isoparamétricas, es decir, aquellas que se obtienen manteniendo constante bien el parámetro u, bien el parámetro v, son curvas racionales de Bézier.

Las superficies B-Spline racionales [59] son, igualmente, una extensión a dos dimensiones de las curvas B-Spline racionales. Su expresión es:

$$\vec{C}(t) = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} w_{ij}\vec{d}_{ij}N_i^k(u)N_j^l(v)}{\sum_{i=1}^{m}\sum_{j=1}^{n} w_{ij}N_i^k(u)N_j^l(v)}, \qquad 2 \le k \le m+1,\ 2 \le l \le n+1 \quad (3.23)$$

Los puntos $\mathbf{d}_{ij}$ forman la malla de control de la superficie, con sus pesos respectivos w_{ij}. En este caso, existe un vector de nudos para cada coordenada paramétrica, formada por los valores u_i y los valores v_i. Si los nudos no están equiespaciados, se tienen las superficies B-Spline racionales no uniformes, o superficies NURBS, que son las que se utilizan en este trabajo para representar los objetos. A continuación se mencionan algunas características importantes de las superficies NURBS:

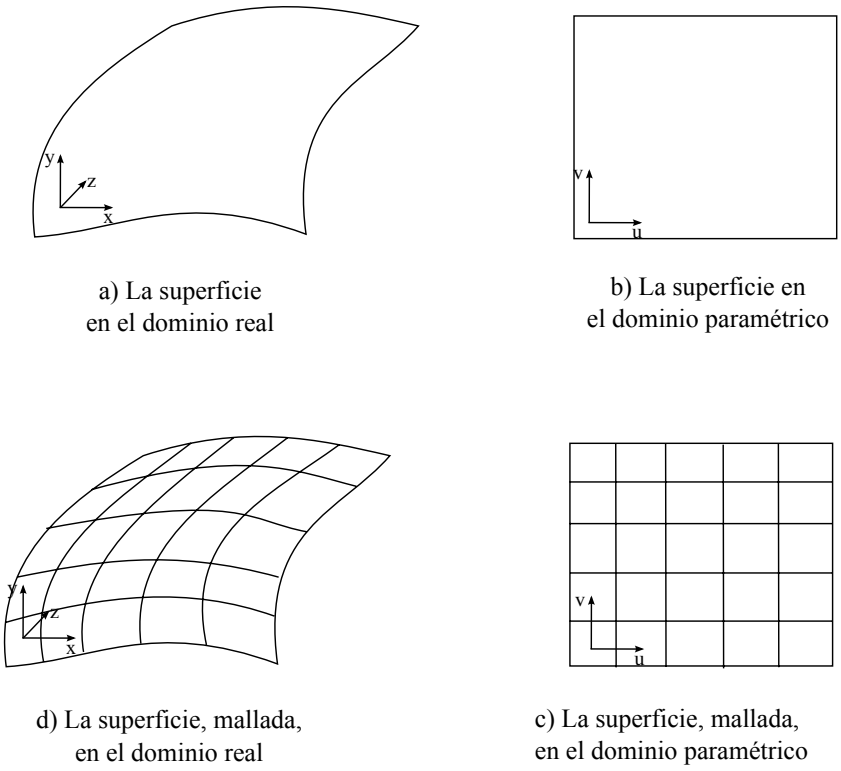
- Control local: es posible modificar uno de los puntos de control sin que esto afecte a toda la superficie. La región

afectada es únicamente aquella en la que el punto tiene influencia.

- Las superficies NURBS muestran continuidad C^{k-2} en la coordenada u y C^{l-2} en la coordenada v.
- Las superficies NURBS son invariantes ante transformaciones afines.

Discretización.

Aplicar el Método de los Momentos a un objeto requiere discretizarlo, de forma que sea posible definir las funciones base y prueba sobre su superficie. Partiendo de un objeto descrito por medio de parches NURBS, la discretización consiste en aplicarle un mallado, el cual se diseña sobre el espacio paramétrico de los parches por medio de curvas isoparamétricas. En muchas aplicaciones, el espaciado entre las curvas que forman la malla debe ser comparable a $\lambda/10$ para conseguir una precisión aceptable. La Figura 3-3 muestra el proceso para una superficie de ejemplo. Partiendo de la superficie en el dominio real (a), se procesa en el dominio paramétrico (b), donde se aplica el mallado (c). Éste se aplica con curvas de u constante o de v constante, y el resultado se refleja en el dominio real (d).



a) La superficie en el dominio real

b) La superficie en el dominio paramétrico

d) La superficie, mallada, en el dominio real

c) La superficie, mallada, en el dominio paramétrico

Figura 3-3. Proceso de mallado de una superficie.

La forma de aplicar el mallado busca crear subparches del mismo tamaño en el dominio paramétrico. Hay que tener en cuenta que al pasar al dominio real, el tamaño no tiene por qué ser igual para todos los subparches. Cada subparche está delimitado por cuatro fragmentos de curvas isoparamétricas denominados lados. Dos de ellos son constantes en u y los otros dos en v. De los dos constantes en u, uno de ellos corresponde a un valor mayor de u que el otro, y se nombran de acuerdo a este hecho: el primero es el lado u^+ y el otro el lado u^-. De forma análoga se nombran los lados constantes en v, v^+ y v^-.

Funciones base.

Sobre los lados del objeto que se obtienen al aplicarle el mallado se definen las funciones de prueba y base. Para las funciones base existen diversas opciones, pero en este trabajo hemos optado por una generalización de las funciones tejado (*rooftop*, en terminología inglesa) [64]. En el grupo de trabajo en el que se desarrolla esta tesis, la experiencia adquirida es que este tipo de función es muy adecuado para este problema [57]. Las funciones tejado originales están pensadas para parches planos, por lo que es preciso extenderlos para parches curvos. En la Figura 3-4 se muestra una función para un lado genérico i. La parte izquierda representa la geometría, y la parte derecha particulariza para el lado i. La función tejado se extiende por los dos subparches que comparten el lado i, el subparche A y el subparche B. En el ejemplo de la figura, el lado i es un lado de $\underline{v}$ constante. La corriente que pasa de un subparche al otro se descompone en una componente perpendicular y otra paralela al lado i. La que fluye de un subparche a otro es la componente perpendicular, y lo hace siguiendo curvas de u constante.

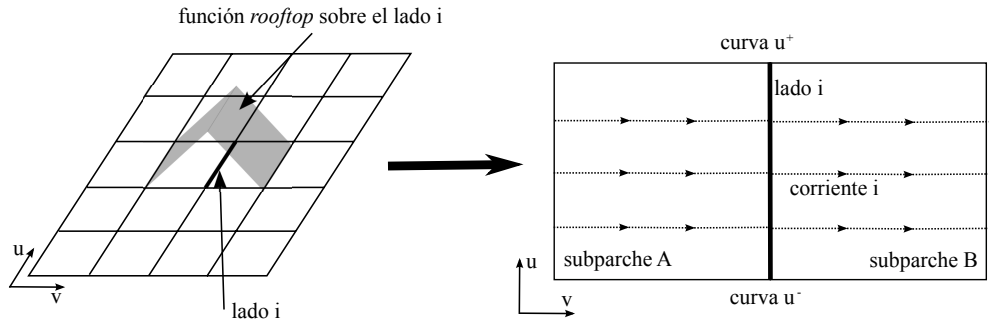


Figura 3-4. Función tejado sobre el lado i.

Como se describe en [65], si la densidad de carga ha de ser constante en ambos subparches, la divergencia de la corriente es constante. La carga asociada al subparche desde el que fluye la corriente es -1/wj, y la carga del subparche hacia el que fluye la corriente +1/wj. En la Figura 3-4, el lado i es un lado v=constante. Si el lado i es el lado v^+ del subparche A, la corriente asociada con la función tejado en este subparche es:

$$\vec{J}(u,v) = J_i^A(u,v)\,\vec{e}_v(u,v) \tag{3.24}$$

En (3.24), $\mathbf{e}_v$(u,v) es el vector paramétrico tangente a la curva u=constante en el lado i:

$$\vec{e}_v(u,v) = \frac{\partial \vec{r}(u,v)}{\partial v} \tag{3.25}$$

Análogamente sería:

$$\vec{e}_u(u,v) = \frac{\partial \vec{r}(u,v)}{\partial u} \tag{3.26}$$

J_i^A(u,v), por su parte, como se explica en [66], se puede expresar como:

$$J_i^A(u,v) = \frac{1}{S_A\sqrt{g(u,v)}}\int_0^v \sqrt{g(u,v)}dv, \tag{3.27}$$

donde

$$g^2(u,v) = [e_u(u,v)\cdot e_u(u,v)][e_v(u,v)\cdot e_v(u,v)] - [e_u(u,v)\cdot e_v(u,v)]^2 \tag{3.28}$$

Como se ha indicado más arriba, se ha supuesto que se fuerza a que la carga sea constante en el subparche A, y de esta manera:

$$\nabla\cdot\vec{J}(u,v)=\frac{1}{S_A} \tag{3.29}$$

con S_A el área del subparche A. Aplicando un razonamiento similar se pueden obtener las expresiones para todo tipo de lado en la geometría:

lado v^-:
$$J_i^A(u,v)=\frac{1}{S_A\sqrt{g(u,v)}}\int_0^{1-v}\sqrt{g(u,v)}dv, \tag{3.30}$$

lado u^+:
$$J_i^A(u,v)=\frac{1}{S_A\sqrt{g(u,v)}}\int_0^{u}\sqrt{g(u,v)}du \tag{3.31}$$

lado u^-:
$$J_i^A(u,v)=\frac{1}{S_A\sqrt{g(u,v)}}\int_0^{1-u}\sqrt{g(u,v)}du \tag{3.32}$$

Al sustituir J_i^A en (3.24), debe tomarse el vector $\mathbf{e_u}$ o $\mathbf{e_v}$, según corresponda.

Funciones de prueba.

Las funciones de prueba se usan para poder introducir las condiciones de contorno en el problema. De nuevo aprovechando la experiencia del grupo de trabajo en el que se enmarca esta tesis, se eligen las funciones cuchilla (o *razorblade*, en inglés) [64], ya que se ha comprobado que dan buenos resultados en conjunción con las funciones tejado. Y, al igual que ocurría con estas últimas, es preciso modificar la formulación original para adaptarlas a parches curvos

[66]. En el caso de la EFIE, que es el que trata este trabajo, se disponen en paralelo con la corriente, como muestra la Figura 3-5. La parte izquierda de la figura muestra cómo se dispone la cuchilla con respecto al lado en cuestión. La parte derecha muestra una vista de la cuchilla desde su perpendicular. La parte a) de la figura muestra la cuchilla rigurosa. Con vistas a facilitar los cálculos es posible usar una aproximación por tramos rectos, como muestra la parte b) de la figura.

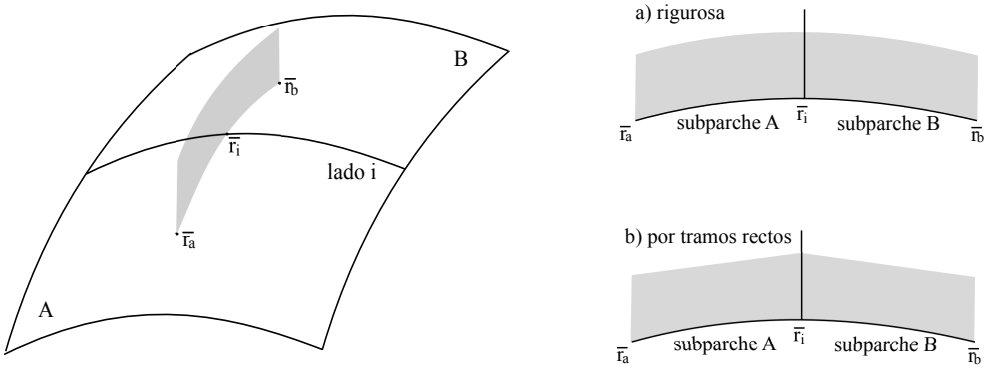


Figura 3-5. Función cuchilla.

La expresión de la parte de la función que se localiza en el subparche A para los lados v^+, por ejemplo, es:

$$W_i^A = \begin{cases} 1, & si \quad u = 0.5 \quad y \quad 0.5 \le v \le 1 \\ 0 & en \ otro \ caso \end{cases} \tag{3.33}$$

Y para los lados v^-:

$$W_i^A = \begin{cases} 1, & si \quad u = 0.5 \quad y \quad 0 \le v \le 0.5 \\ 0 & en \ otro \ caso \end{cases} \tag{3.34}$$

Las expresiones para el subparche B se obtienen análogamente, e igual ocurre para los lados u^+ y u^-.

3.1.4 Matriz de Impedancias.

Las funciones base y de prueba permiten el cálculo de la matriz de impedancias Z, en la que el elemento z_{ij} corresponde al acoplo entre la corriente en el lado j y el elemento del lado i. El lado j se denomina subdominio activo y el lado i subdominio víctima. La Figura 3-6 muestra ambos. Para el subdominio activo se muestra su función tejado, y para el subdominio víctima su función cuchilla. La Figura 3-7 muestra con más detalle la función cuchilla en el subdominio víctima.

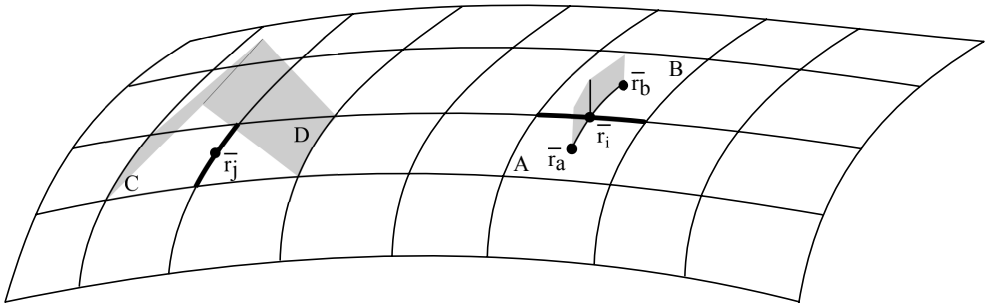


Figura 3-6. Subdominios activo y víctima.

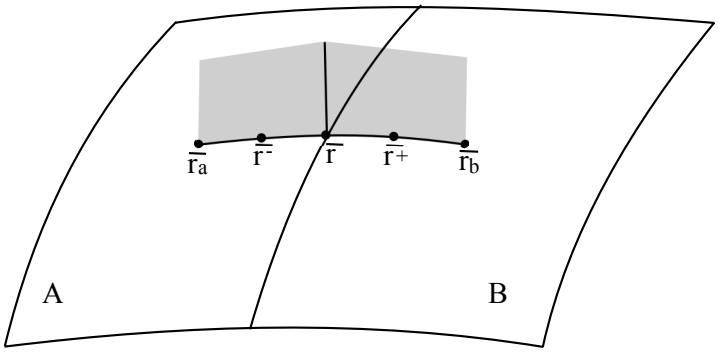


Figura 3-7. Subdominio víctima.

La expresión del acoplo entre ambos subdominios es la particularización de la ecuación (3.1a) para ambos subdominios. El

acoplo se puede descomponer en dos términos, uno inductivo y otro capacitivo:

$$z(i,j) = z_{ind}(i,j) + z_{cap}(i,j) \tag{3.35}$$

La expresión para el término inductivo es:

$$Z_{ind}(i,j) = \int_{r_a}^{r_b} \left[\frac{j\omega\mu}{4\pi} \int_{S_j^C} G(\vec{r},\vec{r}') J_j^C(\vec{r}') ds' + \frac{j\omega\mu}{4\pi} \int_{S_j^D} G(\vec{r},\vec{r}') J_j^D(\vec{r}') ds' \right] d\vec{l}, \tag{3.36}$$

El cálculo del término inductivo puede llegar a ser muy costoso computacionalmente si se realiza tal cual. Por ello, y dado que las integrales de superficie tienen una variación suave con respecto al punto de observación, se puede aproximar de la siguiente forma:

$$Z_{ind}(i,j) = \frac{j\omega\mu}{4\pi} \left[(\vec{r} - \vec{r}_a) \cdot \int_{S_j^C} G(\vec{r}^-,\vec{r}') J_j^C(\vec{r}') ds' + (\vec{r}_b - \vec{r}) \cdot \int_{S_j^D} G(\vec{r}^+,\vec{r}') J_j^D(\vec{r}') ds' \right] \tag{3.37}$$

En esta expresión, $\mathbf{r}$ es el punto medio del lado i, como indica la Figura 3-7. Los puntos $\mathbf{r}^+$ y $\mathbf{r}^-$ son los puntos centrales de los tramos rectos con los que se aproxima la función cuchilla. Los puntos $\mathbf{r_a}$ y $\mathbf{r_b}$ son los centros de los subparches que comparten el lado i.

Por su parte, el término capacitivo es función únicamente de la carga del subdominio j. La expresión que lo define es:

$$z_{cap}(i,j) = \int_{r_a}^{r_b} \left[-\frac{1}{j4\pi\omega\varepsilon} \nabla \int_{S_j^C} \frac{G(\vec{r},\vec{r}')}{S_j^C} ds' + \frac{1}{j4\pi\omega\varepsilon} \nabla \int_{S_j^D} \frac{G(\vec{r},\vec{r}')}{S_j^D} ds' \right] d\vec{l} \tag{3.38}$$

Esta expresión se puede interpretar como el gradiente de un potencial. De esta forma, al promediar con la función cuchilla del lado i, se puede transformar a:

$$z_{cap}(i,j) = P^C(\vec{r}_b) - P^C(\vec{r}_a) - \left[P^D(\vec{r}_b) - P^D(\vec{r}_a)\right] \tag{3.39}$$

donde:

$$P^C(\vec{r}_b) = \frac{-1}{j4\pi\omega\varepsilon_0}\int_{S_j^C}\frac{G(\vec{r}_b,\vec{r}')}{S_j^C}ds' \tag{3.40}$$

y el resto de términos se obtiene de forma análoga.

El cálculo de la matriz de impedancia es una de las partes más costosas en tiempo y recursos, fundamentalmente memoria, del proceso. Hay que tener en cuenta que existe un elemento de la matriz por cada par ordenado de lados posible en la superficie de estudio. Es decir, si la geometría se divide en N lados, la matriz tendrá N^2 elementos. Esta es la razón por la que se han ido desarrollando métodos como el de las Funciones Base Características, que reducen el número de elementos de la matriz, lo que reduce los requisitos de memoria del problema. Para reducir el tiempo de proceso, se usa el propio CBFM, conjuntamente con la paralelización del algoritmo, lo que constituye la parte fundamental de este trabajo.

3.2 El Método de las Funciones Base Características.

El Método de las Funciones Base Características (CBFM, *Characteristic Basis Function Method* en terminología inglesa) [11] busca una forma eficaz de resolver el sistema matricial que plantea el Método de los Momentos (3.10). Se basa en el uso de funciones base

características, un conjunto de funciones base de alto nivel, es decir, que se definen sobre grandes regiones de la geometría, generalmente denominadas *bloques*. De esta forma, y dado que no se ven afectadas por la discretización, transforman el sistema de ecuaciones reduciendo el número de incógnitas. El resultado es que el sistema se puede resolver por un método directo.

El método es independiente de las funciones base y de prueba utilizadas ya que las CBFs se pueden expresar en función de cualquier tipo de funciones de bajo nivel. En nuestro caso, aplicaremos el CBFM sobre el Método de los Momentos y las CBFs se obtendrán en función de las funciones tejado y cuchilla descritas en secciones anteriores. Una vez obtenida su expresión, se podrá realizar la conversión del sistema y obtener la matriz reducida.

El CBFM está pensado para aplicarse sobre una división en bloques de la geometría bajo estudio. Para facilitar la exposición, en esta memoria se plantea en primer lugar el funcionamiento del método cuando la geometría es un único bloque para, a continuación, describir el proceso cuando la geometría se divide en varios bloques.

3.2.1 Obtención de las Funciones Base Características.

El planteamiento original del método [11] divide las CBFs que se van a obtener en dos grupos: las que se deben directamente al campo incidente y las que se deben al campo irradiado por las corrientes inducidas en otras partes de la geometría. Esto hace que la matriz reducida que se obtiene dependa del campo externo. En nuestro caso, lo que se pretende es obtener una matriz de impedancias reducida lo más general posible, de forma que no sea necesario recalcularla continuamente.

Para conseguir esto, nuestra técnica utiliza un conjunto de ondas planas (el "espectro de ondas planas", o *Plane Wave Spectrum*, PWS) que inciden sobre el objeto desde todas las direcciones del espacio, como se muestra en la Figura 3-8. Las direcciones de incidencia están uniformemente espaciadas, y el espaciado se puede ajustar según se necesite. En este trabajo, basándonos en las experiencias previas obtenidas en el grupo de trabajo, se ha utilizado un espaciado de 10 grados, tanto para el ángulo polar θ como para el azimut φ. Dado que θ va desde 0 a 180 grados, y φ va desde 0 hasta 360, el total de ondas en el PWS es de 648.

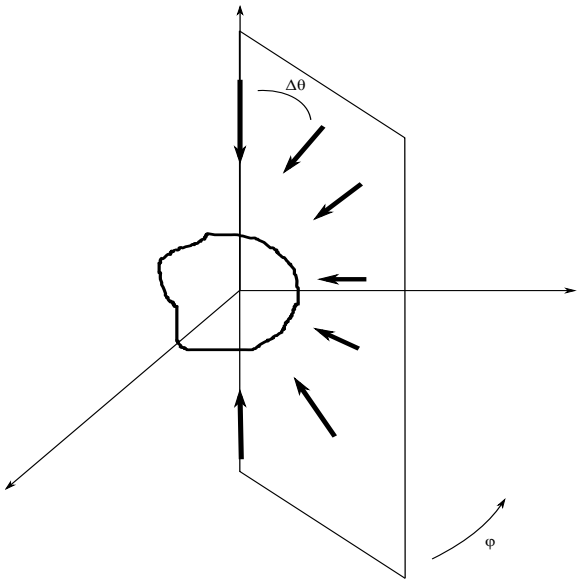


Figura 3-8. Generación del PWS.

Una vez generadas las excitaciones del PWS, se deben obtener las corrientes inducidas en el objeto por esas excitaciones. Es posible calcularlas por medio del Método de los Momentos, pero también si se utilizan métodos de Óptica Física. El Método de los Momentos es aconsejable si la geometría contiene estructuras irregulares o

pequeñas. Si el objeto tiene una geometría suave y bloques grandes, los métodos de Óptica Física obtienen los resultados con suficiente precisión y de forma mucho más rápida. Para dotarlo de mayor generalidad en cuanto a las geometrías que se pueden tratar, en este trabajo se utiliza el Método de los Momentos. Esto implica resolver el sistema de la ecuación (3.10), con las excitaciones como términos independientes. El resultado es la matriz de corrientes inducidas J, en la que cada corriente está expresada en forma de un vector columna de coeficientes. Esta matriz de corrientes se ortogonaliza. En su artículo de 2005 [67], Tiberi *et al.*, proponen como método la Descomposición en Valores Singulares (SVD, *Singular Value Decomposition*), que es el que usamos nosotros. La SVD descompone la matriz J:

$$J = USV^{*} \tag{3.41}$$

donde S es una matriz diagonal, que contiene los valores singulares, y U y V son matrices unitarias. V* representa la transpuesta conjugada de V.

El convenio establece que los elementos de S están ordenados de mayor a menor peso. Estos valores singulares pueden llegar a diferir entre sí varios órdenes de magnitud (del mayor al menor), por lo que es suficiente retener únicamente los mayores de entre todos ellos. En su artículo de 2008 [68], García *et al.* muestran el efecto en la precisión de prescindir de los valores singulares de menor peso. Sus observaciones son que manteniendo sólo aquellos valores singulares con una magnitud de como poco entre 500 y 1000 veces menor que el mayor de todos se consigue una precisión aceptable en la mayoría de

los casos. La aplicación de un umbral permite reducir enormemente el tamaño del sistema.

Las CBFs se pueden poner en función de las funciones base y de prueba que se usan en el MoM (representadas aquí por T_i para las funciones tejado o '*rooftop*' y R_i para las funciones cuchilla o '*razorblade*':

$$J_k(u,v) = \sum^{N} \alpha_n^k T_n(u,v) \tag{3.42}$$

$$W_k(u,v) = \sum^{N} \alpha_n^k R_n(u,v) \tag{3.43}$$

La ecuación (3.42) expresa la función base de alto nivel (CBF) k-ésima J_k en función de las funciones base de bajo nivel, es decir, de las funciones tejado T_i, en una geometría con N subdominios. La ecuación (3.43) pone en relación la función de prueba de alto nivel k-ésima W_k con las N funciones de prueba de bajo nivel R_n (las funciones cuchilla). Los coeficientes α_i^k, en ambos casos, son los resultantes de la ortogonalización. Si la aplicación del umbral permite retener M CBFs, la matriz de acoplos reducida ZR tiene la forma:

$$[ZR] = \begin{bmatrix} \langle L(J_1),W_1\rangle & \langle L(J_2),W_1\rangle & \cdots & \langle L(J_M),W_1\rangle \\ \langle L(J_1),W_2\rangle & \langle L(J_2),W_2\rangle & \cdots & \langle L(J_M),W_2\rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle L(J_1),W_M\rangle & \langle L(J_2),W_M\rangle & \cdots & \langle L(J_M),W_M\rangle \end{bmatrix} \tag{3.44}$$

Los elementos de la matriz se obtienen de la expresión:

$$\left\langle L(J_j),W_i\right\rangle = \sum_{\_}^{N}\sum_{\_}^{N} \alpha_l^j \alpha_k^{*i} \left\langle T_l(u,v),R_k(u',v')\right\rangle \tag{3.45}$$

Generalmente, lo que interesa en este trabajo es obtener la matriz reducida a partir de la matriz de acoplos, por lo que se puede sustituir el producto interno de (3.45) por el elemento correspondiente de dicha matriz:

$$\left\langle L(J_j),W_i\right\rangle = \sum_{\_}^{N}\sum_{\_}^{N} \alpha_l^j \alpha_k^{*i} z_{kl} \tag{3.46}$$

O, en forma matricial:

$$ZR = U^* ZU \tag{3.47}$$

Donde U es la matriz que se obtiene al aplicar la SVD en la ecuación (3.41).

3.2.2 División en bloques.

El método descrito en el artículo de García *et al.* [68], que sirve de base a este trabajo, presupone una división del objeto de estudio en bloques. Sin embargo, si el objeto no es demasiado grande (en términos de número de subdominios), se le puede aplicar el CBFM en un único bloque, como se ha expuesto en la sección anterior. Esto supone tener que calcular la matriz completa de acoplos con funciones base de bajo nivel (a la que aquí nos referiremos como matriz de impedancias original), que es la parte más costosa del MoM en términos de tiempo de cálculo, pero, aun así, se puede obtener un beneficio, ya que generalmente la matriz resultante de la aplicación del CBFM, la matriz de acoplos reducida, es mucho más pequeña, lo

que permite calcular la sección radar para una radiación dada en mucho menos tiempo. El beneficio radica en que, una vez obtenida esta matriz reducida, es la que se utiliza para calcular la sección radar todas las veces que sea necesario, en lugar de la matriz completa original (es decir, se reutiliza).

Sin embargo, la utilidad del CBFM se muestra realmente con objetos grandes en los que es imprescindible practicar una división en bloques, ya que, en estos casos, al procesar la geometría por bloques, no es necesario obtener la matriz de impedancias original completa.

La aplicación del método por bloques parte de un preproceso del objeto que proporciona una división geométrica del objeto en bloques. A cada uno de ellos individualmente se les va a aplicar una versión adaptada del método expuesto en la sección anterior para un objeto entero. Las modificaciones tienen que ver con el hecho de que cada uno de los bloques no es un objeto por sí solo, aislado, sino que todos juntos conforman uno de entidad superior. La principal consecuencia de esto es que, si las corrientes inducidas en cada bloque por el campo incidente se obtienen por el MoM, como es el caso en este trabajo, es preciso ampliar ligeramente el bloque para garantizar la continuidad eléctrica entre bloques. A cada bloque original se le añaden los subdominios de otros bloques que son contiguos a los que forman la frontera del bloque en proceso, hasta una cierta distancia, como se muestra en la Figura 3-9. Una vez se han obtenido las corrientes inducidas, se ha garantizado la continuidad eléctrica, por lo que desde este punto se pueden desechar los datos correspondientes a los subdominios añadidos. A los subdominios añadidos se les suele denominar franja, y a la porción del objeto formada por un bloque original más los subdominios añadidos de esta manera (la franja) se

le denomina bloque extendido. En el caso de que las corrientes inducidas se obtuvieran por métodos de óptica física no sería necesario añadir la franja a los bloques. Pero el MoM permite procesar geometrías en las que los métodos de óptica física no funcionan tan bien, como es el caso de estructuras irregulares o pequeñas. Puesto que permite analizar superficies más generales, se eligió este método para este trabajo.

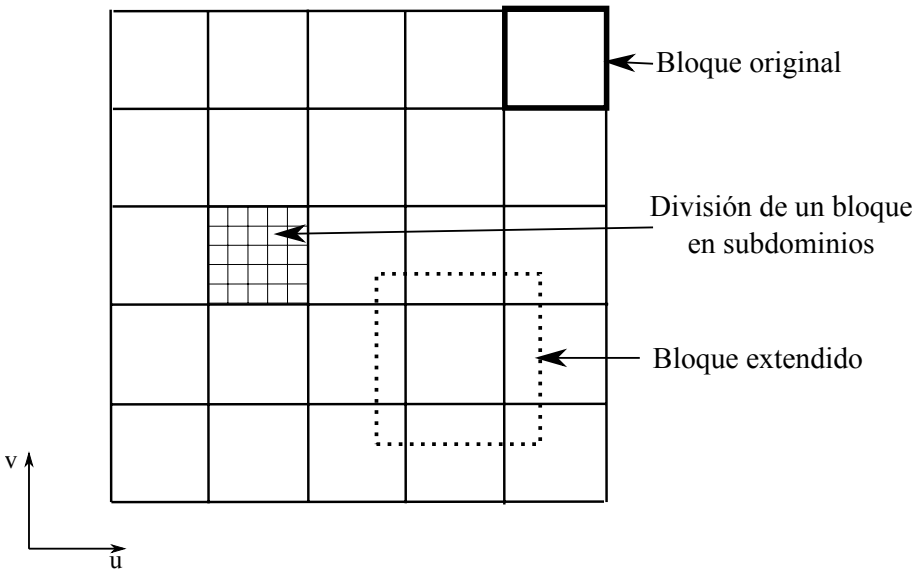


Figura 3-9. Bloque extendido.

Al procesar la geometría por bloques, las matrices pasan a estar compuestas por submatrices de la forma correspondiente, tanto la matriz de impedancias original como la matriz reducida. Por ejemplo, ésta última tiene la siguiente forma:

$$ZR = \begin{bmatrix} \left[ZR^{11}\right] & \left[ZR^{21}\right] & \cdots & \left[ZR^{k1}\right] \\ \left[ZR^{12}\right] & \left[ZR^{22}\right] & \cdots & \left[ZR^{k2}\right] \\ \vdots & \vdots & \ddots & \vdots \\ \left[ZR^{1k}\right] & \left[ZR^{2k}\right] & \cdots & \left[ZR^{kk}\right] \end{bmatrix} \tag{3.48}$$

suponiendo que la geometría se subdivide en k bloques. La submatriz $[ZR^{ij}]$ representa el acoplo del bloque i con el bloque j, donde i es el bloque víctima y j el bloque activo, siguiendo con el criterio que se utilizó al definir la matriz original Z. La descomposición de ésta última en submatrices es análoga. Los elementos de $[ZR^{ij}]$ responden a la expresión:

$$\left[ZR_{lk}^{ij}\right] = \left[\left\langle L(J_l^j), W_k^i \right\rangle\right] \tag{3.48}$$

La ecuación (3.48) muestra cómo se obtiene el elemento lk de la submatriz ij: sale de calcular el producto interno del operador L aplicado a la función base l-ésima del bloque j-ésimo (es decir, $L(J_l^j)$) con la función de prueba i-ésima del bloque k-ésimo (W_k^i). Los subíndices l y k indican el número de CBFs en los bloques j e i respectivamente. Las submatrices son cuadradas si i=j, ya que en este caso representan el acoplo de un bloque consigo mismo, pero no lo son en general si $j \neq i$, ya que cada bloque de la geometría tendrá un número diferente de CBFs.

4 Aceleración del MoM

4.1 Introducción.

El programa original en el que se basa este trabajo está escrito en el lenguaje de programación FORTRAN. En el momento de comenzarlo eso suponía un problema, ya que no existían compiladores de este lenguaje de programación que permitieran aprovechar todos los recursos que CUDA ofrece para adecuar al máximo el programa a las GPUs. Existía un compilador de FORTRAN, creado por Portland Group, que admitía una cierta orientación a CUDA, pero no era comparable en funcionalidad al compilador de C de NVIDIA. Esta compañía desarrollaba y ofrecía libremente un compilador que sí disponía de estos avances, pero era un compilador del lenguaje de programación C. Dado que se consideró que poder controlar al máximo la forma de organizar los datos en la tarjeta y la ejecución del código era fundamental para el máximo aprovechamiento de la misma, se optó por desarrollar en C la parte del programa que debía correr en la GPU. Por otro lado, la adaptación del programa a CUDA suponía un gran cambio en el algoritmo, dada la forma en la que funciona la GPU. Si se suman estos dos factores, el resultado es que se había de desarrollar un programa con una estructura muy diferente al original, y que además debía estar escrito en un lenguaje de programación con algunas diferencias notables, fundamentalmente en cuanto al almacenamiento de los datos.

Para facilitar la transición se tomó la decisión de crear una versión intermedia: una versión en C del programa original. Se trataría de un programa con la misma funcionalidad del original, y

que siguiera el mismo algoritmo, pero programado en C. Su objetivo debía ser servir de referencia intermedia, fundamentalmente para la detección y corrección de errores de programación. De esta manera, la versión en CUDA del programa podría compararse con la versión intermedia con algo más de facilidad, ya que, aunque el algoritmo era diferente, al menos el lenguaje de programación era el mismo.

La versión intermedia no está totalmente escrita en C, sin embargo. La traducción a C afecta únicamente a aquellas partes que posteriormente se ejecutarán en la GPU. La parte inicial del programa, que se encarga de realizar una serie de inicializaciones y cálculos previos, se decidió dejarla en FORTRAN, ya que no tiene un peso demasiado importante en el tiempo de ejecución del programa y su conversión no es interesante, a efectos de este trabajo. Es la parte principal del programa, especialmente en cuanto a peso en el tiempo total de ejecución del mismo, es decir, la que calcula la matriz de impedancias del Método de los Momentos (MoM), la que se convirtió a C en la versión intermedia y a CUDA posteriormente. La Figura 4-1 muestra qué grandes bloques se mantienen en FORTRAN y qué partes se traducen a C (y posteriormente a CUDA). Es preciso indicar que, en la figura, el tamaño relativo de los bloques no se corresponde con su tiempo real de ejecución.

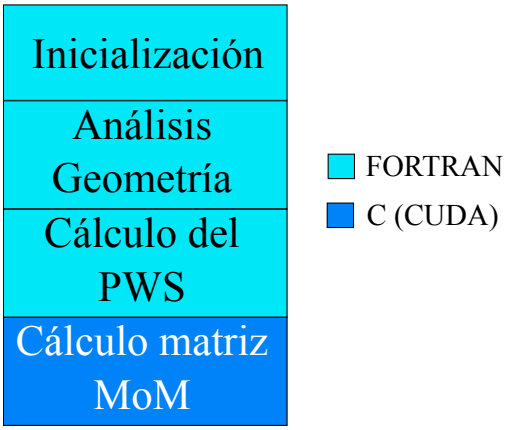


Figura 4-1. Bloques del programa

Pasar de esta versión en C que sigue el algoritmo original a una nueva versión en C con el algoritmo modificado para correr en CUDA fue el segundo paso de este trabajo. La primera versión para CUDA no consiguió una mejora en rendimiento reseñable. Fue necesario explorar más a fondo los recursos de la GPU y de CUDA para obtener rendimientos aceptables. A lo largo de este capítulo se describirán las características que se exploran y se explicarán las razones que hacen que el rendimiento sea mejor.

4.2 Versión intermedia en lenguaje C.

Como se explica en la sección anterior, la razón de ser de la versión intermedia en C es permitir realizar la transición completa de FORTRAN a CUDA con un punto intermedio que sirva para comprobar errores, en primer lugar en el paso de FORTRAN a C, y después, en el paso de ésta última (el algoritmo original, expresado en C) a una para CUDA.

Además de la conversión a C de las rutinas encargadas de calcular la matriz de impedancias, se ha aprovechado para introducir algunas modificaciones en la organización de los datos en memoria con vistas

a que ésta se adapte mejor a la jerarquía y modo de acceso de una GPU. Estas mejoras afectan fundamentalmente a las matrices que contienen la información de la geometría y consisten en reordenar los índices de acceso, de forma que elementos que van a ser leídos por el programa uno después de otro, estén situados en memoria en posiciones consecutivas. Esto tiene su interés en la versión secuencial, ya que estos datos podrán situarse en el mismo bloque de memoria caché, reduciendo el tráfico de ésta con niveles superiores de la jerarquía de memoria, y además también será conveniente cuando se cree la versión para la GPU.

El programa comienza procesando la geometría a la que hay que aplicarle el análisis electromagnético. Se divide en parches y subparches y se calculan las posiciones de los mismos (de su centro, concretamente), así como las de los segmentos que los separan (los subdominios), siguiendo el procedimiento indicado en el apartado de *Discretización* de la sección 3.1.3 (ver Figura 3-3). También se calculan las derivadas del vector de posición en los subparches, las posiciones en los mismos de los puntos de integración para las integrales lineales y de superficie y otros datos necesarios para el análisis. Para todo esto se ha respetado la versión original en FORTRAN. También se ha respetado para calcular el Espectro de Ondas Planas (*Planar Wave Spectrum*), aunque éste no se utilice en el MoM, sino en el CBFM.

Una vez analizada la geometría, se procede a aplicarle el método de los momentos. Esta parte ya debe ser traducida a C, y el procedimiento consiste en calcular el acoplo entre cada par de subdominios de la geometría. Para ello, se crean dos bucles que recorren, cada uno de ellos, todos los subdominios, como indica la

Figura 4-2, el más externo tratándolos como subdominio víctima y el más interno como subdominio activo. Para cada par de subdominios se calcula tanto el acoplo inductivo como el capacitivo. El inductivo consiste en una integral de línea a lo largo de la cuchilla. La integral se transforma, por el método de las cuadraturas de Gauss [69], en un sumatorio. Por ejemplo, las integrales entre los corchetes en la ecuación (3.36) se transforman de esta manera:

$$\int_{S_j^C} G(\vec{r},\vec{r}')J_j^C(\vec{r}')ds' \approx \sum_{i=1}^{N}\sum_{l=1}^{M} w_i w_l G(u_i,v_l)J_j^i \qquad (4.1)$$

Para obtener cada uno de los términos del sumatorio es preciso calcular una integral de superficie en los subparches que bordean el subdominio activo, y esto, a su vez y de forma análoga, se convierte en el sumatorio de un sumatorio. Cada término de éste último sumatorio requiere calcular el valor de la función integrada en un punto de una malla bidimensional que cubre ambos subparches del subdominio activo.

```
/*Inicialización y cálculos generales*/
Generar descripción de la geometría
Generar PWS
/*Generar matriz de impedancias*/
Para cada subdominio víctima
   Para cada subdominio activo
        /*Obtener elemento de Z*/
        Calcular acoplo inductivo víctima-activo
        Calcular acoplo capacitivo víctima-activo
        Acumular
```

Figura 4-2. Pseudo-código general del programa.

De esta manera, cada uno de los sumatorios necesarios para calcular el acoplo inductivo se transforma en un bucle de tipo `for`. Dentro de cada uno se encuentra otro bucle `for`, y así sucesivamente hasta completar los 3 niveles de anidamiento que supone el acoplo inductivo (los tres sumatorios). La Figura 4-3 representa esta construcción. En el bucle más interno se realiza el cálculo del valor de la función que se integra en ese punto concreto. La acción denominada '`Acumular`' representa la suma siguiendo el método de las cuadraturas de Gauss.

```
/*Acoplo inductivo para un par de subdominios*/

/*Integral de línea*/
Para cada punto en la cuchilla
   /*Integral de superficie*/
   Para cada punto P1 en la dimensión 1
         Para cada punto P2 en la dimensión 2
               Evaluar la función en (P1,P2)
               Acumular
         Acumular
   Acumular
```

Figura 4-3. Pseudo-código para el cálculo del acoplo inductivo de un par víctima-activo

La Figura 4-4 representa gráficamente todo el proceso para obtener el componente de acoplo inductivo de un elemento de la matriz. La primera fila de cajas (con la letra C) representa el bucle `for` que recorre la cuchilla (una caja por iteración). Para cada elemento de la cuchilla es preciso evaluar la función tejado, proceso representado por las dos filas (ya que es una integral de superficie) de

cajas con la letra T. Cada elemento final supone la evaluación (cajas con la letra E) de la función en un punto del subdominio activo.

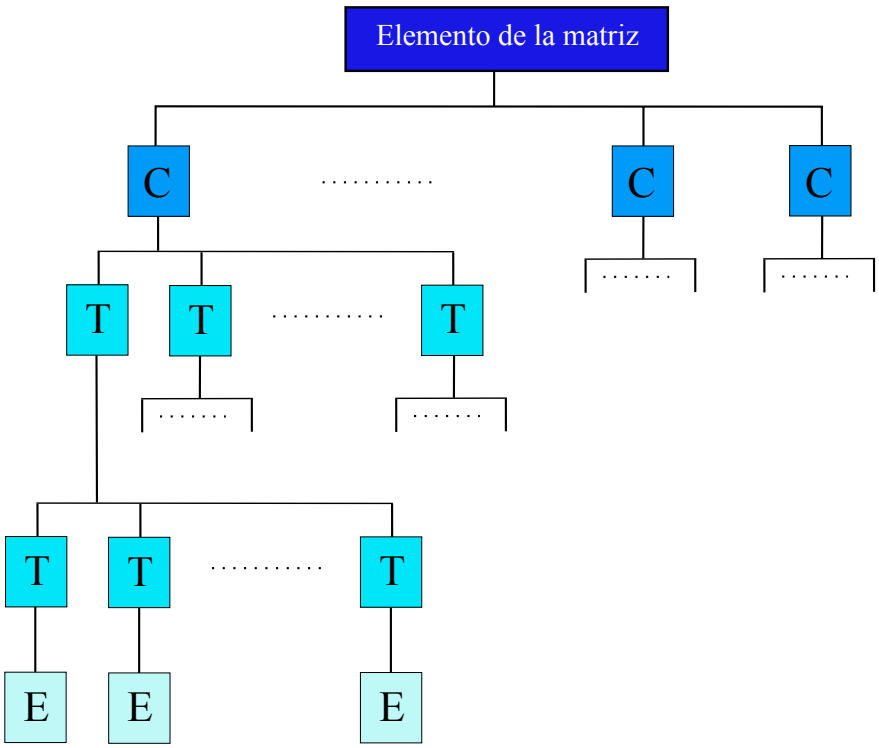


Figura 4-4. Esquema del cálculo del acoplo inductivo.

El acoplo capacitivo (ver Figura 4-5) tiene una estructura algo más sencilla, ya que sólo es preciso calcular la integral de superficie de cada uno de los subparches que delimitan el subdominio activo para obtener su efecto sobre el centro de cada uno de los dos subparches que delimitan el subdominio víctima. En total son cuatro integrales de superficie, y la secuencia de operaciones es parecida al caso inductivo, pero con un nivel menos de anidamiento, ya que el bucle correspondiente a la integral sobre la cuchilla no está presente.

```
/*Acoplo capacitivo para un par de subdominios*/
Para cada subparche víctima
   Para cada subparche activo
         Calcular número de puntos de la
         integral de superficie
         Calcular integral de superficie
         Acumular
```

Figura 4-5. Pseudo-código para el cálculo del acoplo capacitivo de un par víctima-activo

La versión intermedia es una conversión prácticamente línea a línea de la versión original en FORTRAN. Sólo en contadas ocasiones se ha introducido alguna modificación como desenrollado de bucles o hacer alguna función '*inline*' (sustituir su invocación por su código), optimizaciones que en cualquier caso puede realizar también el propio compilador sobre el programa original.

4.3 Versión inicial para CUDA.

La conversión de un código secuencial, como el programa original de este trabajo, a código para CUDA implica una cantidad ingente de modificaciones. La forma de funcionar de una GPU obliga a reordenar las tareas de computación y agruparlas de otra manera si se quieren aprovechar sus recursos. Y estas modificaciones sobre el código requieren otras sobre la distribución de los datos en la GPU.

En el caso de este programa, su estructura original ha debido ser rehecha completamente en las partes más importantes. Se trata de un programa con un alto grado de paralelismo, ya que el cálculo del

acoplo entre dos subdominios dados no influye en el mismo cálculo para otro par de subdominios distinto. Pero el paralelismo se encuentra en el interior de una serie de bucles anidados, lo que hace que su aprovechamiento no sea trivial.

La parte que es ejecutable en paralelo corresponde, en la Figura 4-4, a las cajas con la etiqueta E, es decir, a la evaluación de la función tejado. Al procesar un par de subdominios víctima-activo, en la versión original, los valores se calculan de forma secuencial y cada uno se termina de procesar completamente antes de calcular el siguiente. En la versión paralela se calculan todos de una vez para todos los puntos de la malla bidimensional por la que se aproxima la superficie de integración del subdominio activo. Una vez calculados los valores, de acuerdo con el método de las cuadraturas de Gauss, se realiza la suma ponderada de los valores, primero en una dimensión, luego en la otra (de la malla bidimensional, lo que supone la integral de superficie), y por último a lo largo de la cuchilla (la integral de línea). Estas operaciones (las sumas ponderadas) ya no son tan paralelas, y en la literatura de programación se conocen como 'reducciones' (ya que una colección de valores se "reduce" a uno, en nuestro caso, por medio de una suma).

El número total de sumas ponderadas para un par de subdominios dado depende del número de puntos de integración seleccionado en cada una de las integrales. Para las cuchillas se trabaja con 2 ó 4 puntos, lo que da 4 u 8 valores que hay que sumar. En el caso de las funciones tejado, se trabaja con 2, 4 ó 10 puntos de integración. Así, el número total de valores para un par de subdominios oscila entre un mínimo de 32 y un máximo de 1600. Esta disparidad será una de las causas de pérdida de rendimiento que se planteará más adelante.

En cualquier caso, incluso 1600 evaluaciones no son muchas para obtener una gran mejora de rendimiento. Si se calcularan solamente las 1600 en paralelo en la GPU, teniendo en cuenta que el proceso tiene una parte secuencial significativa (las operaciones de reducción), no se conseguiría una gran mejora. Para llegar a aprovechar el paralelismo de forma considerable es necesario procesar los acoplos de una gran cantidad de pares de subdominios a la vez, tantos como admita la GPU. De esta manera, el algoritmo realizará todas las evaluaciones de las funciones que se integran para una serie de pares subdominio víctima-subdominio activo en paralelo, y cuando acabe con ese grupo de subdominios seguirá con otros, hasta que no quede ninguno, antes de pasar a procesar las evaluaciones.

Esta forma de proceder obliga a dos cosas: por un lado, es preciso almacenar de forma temporal los resultados de las evaluaciones para luego poder procesarlas. Por otro, surge la necesidad de una cierta contabilidad, para saber en cada caso a qué par víctima-activo corresponde cada cálculo que se está realizando, y, dentro de un par determinado, cuál de los cálculos es. La primera necesidad se resuelve fácilmente reservando espacio en la memoria global de la GPU. En cuanto a la segunda, la información de contabilidad se prepara, previamente a la ejecución de código en la GPU, en la CPU. Para cada uno de los pares víctima-activo se calcula el número de puntos en la cuchilla y el número de puntos en las integrales de superficie. Además se calculan las posiciones de los subparches implicados. Se va rellenando una lista con esta información, que será utilizada más adelante por los sucesivos *kernels* que calculan la matriz de impedancias.

Una vez obtenida toda esta información y transferida a la GPU, se procede a obtener los elementos de la matriz de impedancias. Se procesan todos los posibles pares subdominio víctima-subdominio activo de una vez, es decir, se eliminan los bucles `for` del algoritmo secuencial. El primer *kernel* que se ejecuta realiza parte de los cálculos para el acoplo inductivo. En concreto, las integrales de superficie. Se crea un bloque de hilos en la GPU para cada punto de cada cuchilla. El bloque se encarga de calcular el valor de la integral de superficie que corresponde a ese punto de la cuchilla. Dentro de cada bloque, cada hilo se encarga de realizar la evaluación de la función en un punto de la malla bidimensional situada sobre cada uno de los dos subparches activos. Así, dado que las integrales son de 2, 4 ó 10 puntos, en total el número de hilos que puede haber en un bloque son 8 (2x2x2), 32 (2x4x4) ó 200 (2x10x10). La relación entre hilos, bloques de hilos, puntos en la malla bidimensional y puntos en la cuchilla se muestra en la Figura 4-6. El *grid* de bloques de hilos se encarga de calcular, en un único paso, todas las integrales de superficie de todos los pares subdominio víctima-subdominio activo.

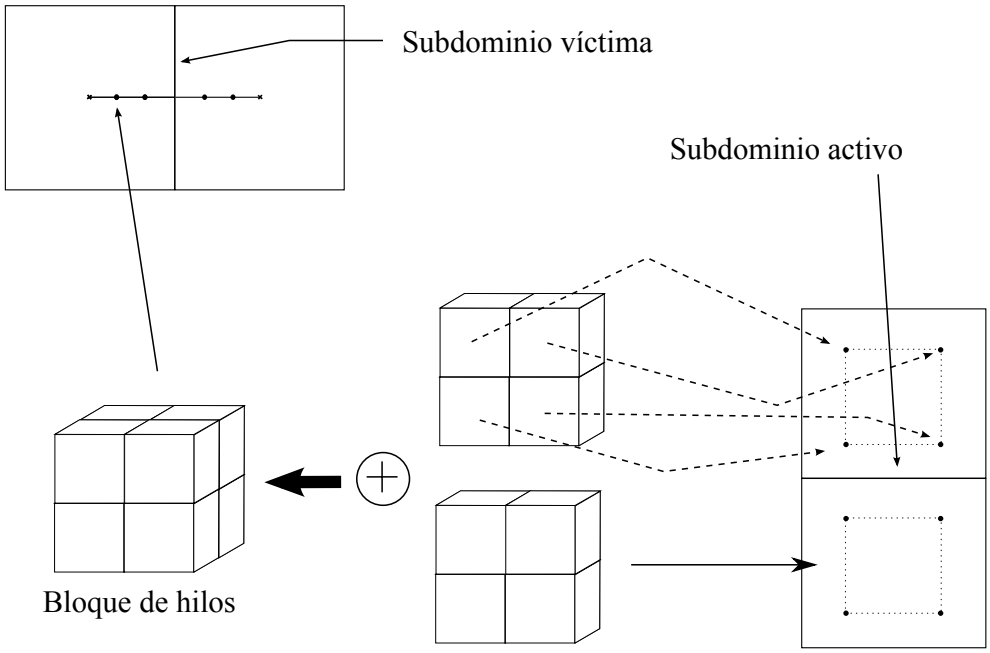


Figura 4-6. Estructura de los bloques de hilos (ejemplo para 2 puntos de integración).

En esta versión, de entrada todos los bloques de hilos se crean igual de grandes (es decir, tienen el tamaño máximo que permite procesar cualquier integral, 200), y cada uno luego se adapta dinámicamente al número de hilos que realmente necesita. Al comenzar la ejecución, los hilos de cada bloque analizan qué tipo de integral están realizando (es decir, cuántos puntos de integración se usan), y ejecutan una secuencia de código u otra (pero todos los hilos del mismo bloque la misma). Además, en cada secuencia de código, los hilos que no son necesarios simplemente no hacen nada. Esta decisión se realiza de forma dinámica en tiempo de ejecución: cada hilo, en función del número de puntos de integración que le corresponden a su bloque, debe decidir si debe realizar algún trabajo o no, y cuál. Para esto es necesario crear un índice lineal de hilo en función de su situación en el bloque de hilos. Los hilos que tengan un

índice superior a los que hagan falta para calcular la integral, simplemente terminan. De esta manera, con un único *kernel* se cubre toda la funcionalidad necesaria.

Cada hilo útil evalúa la función en el punto que le corresponde, y a continuación se realiza la operación de reducción. Para reducir el número de sentencias de control en el código del *kernel*, todos los hilos útiles colaboran en ella. Esto no es necesario, pero de esta manera se reducen las sentencias de control de tipo `if` y los bucles `for`, que suponen una penalización en el rendimiento. Así, en un primer paso, cada hilo calcula la suma ponderada del valor que ha calculado él y el del hilo "a su derecha", es decir, el que tiene el siguiente índice en el bloque de hilos. A continuación, cada hilo calcula la suma ponderada del valor que calculó anteriormente y el del hilo que está situado dos posiciones más allá, y así sucesivamente utilizando como desplazamiento potencias crecientes de 2 (para el caso de 10 puntos de integración es necesario adaptar este procedimiento, ya que no es válido directamente).

La operación de reducción se muestra gráficamente en la Figura 4-7. El caso mostrado parte de 8 valores que hay que sumar. En un primer paso, se suma cada valor con el de su derecha. Las flechas que entran a la figura por la derecha se corresponden por las que salen por la izquierda. El siguiente paso es sumar cada valor con el situado dos posiciones a la derecha. Por último, se suma cada valor (en la figura, por claridad, se muestra únicamente la suma que calcula el primer elemento, pero se harían todos) con el situado cuatro posiciones más allá. La operación de reducción realizada de esta manera tiene una duración en pasos igual al logaritmo en base dos del número de datos a sumar (para potencias de dos). Parte de la

ventaja, en el caso de las GPUs, es que se evitan saltos condicionales, ya que en forma secuencial sería necesario un bucle.

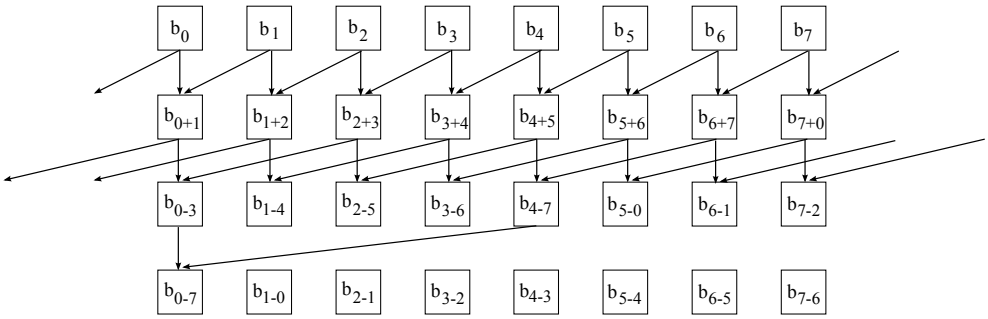


Figura 4-7. Operación de reducción.

El proceso es más largo o más corto dependiendo de cuántos hilos útiles tenga el bloque de hilos. Cuando acabe, todos los hilos tienen el valor final, pero sólo el primero de ellos lo deposita en memoria. Este valor corresponde al punto de la cuchilla asociado con el bloque de hilos que lo ha generado, y será utilizado después por otro *kernel*.

A continuación, se ejecuta el *kernel* que calcula el acoplo capacitivo. En este caso únicamente hay que calcular cuatro integrales de superficie, por lo que se crean sólo dos bloques de hilos. Cada uno procesa dos de ellas, las dos que se necesitan para el mismo subparche víctima. A diferencia con el acoplo inductivo, en el capacitivo el algoritmo original especifica que el número de puntos de integración de los dos subparches activos puede ser diferente en cada uno, lo que complica un poco el cálculo. Por ello, en este primer *kernel* del capacitivo, para ambos subparches activos se utiliza el mismo número de puntos de integración, el correspondiente al primero de los dos subparches activos, y un *kernel* posterior corregirá el valor del segundo para los casos en los que el segundo necesite un número de puntos de integración diferente.

El desarrollo del *kernel* para el capacitivo es muy parecido al del inductivo. Cada bloque de hilos opera con el máximo de hilos inicialmente, y en función del número de puntos de integración necesario ejecuta una secuencia de código u otra, y los hilos que no son necesarios terminan. El resultado, al igual que ocurre con el acoplo inductivo, se queda almacenado en la memoria global de la GPU para ser procesado después.

El valor del acoplo capacitivo debe corregirse en algunos casos en los que el número de puntos de integración no es el mismo para los dos subdominios activos. Un nuevo *kernel*, el tercero, se encarga de esto, modificando los valores almacenados en la memoria de la GPU de la forma adecuada. La estructura del *kernel* es muy parecida a la de los dos anteriores.

Una vez calculadas las integrales de superficie y los acoplos capacitivos, un cuarto *kernel* se encarga de calcular las integrales de línea y de sumar el acoplo capacitivo. Para cada cuchilla, calcula la suma ponderada de las integrales de superficie correspondientes y al resultado final, le suma el acoplo capacitivo. Al igual que en el caso del cálculo de las integrales de superficie, se crea un único tipo de bloque de hilos, y cada hilo ejecuta una secuencia de código, en función del número de puntos en la cuchilla que procesa. Es decir, cada uno de los hilos obtiene una integral de línea y le suma el acoplo capacitivo. De esta manera, su resultado es un elemento de la matriz de acoplos. La matriz de impedancias resultante queda almacenada en la memoria de la GPU para su posterior procesamiento.

La Figura 4-8 muestra la secuencia de *kernels* y los elementos sobre los que trabajan. Muestra las integrales de superficie y las de línea. Por simplificar la figura, se han omitido los *kernels* que

calculan el acoplo capacitivo y su corrección, que irían entre las integrales de superficie y las de línea. El primer *kernel*, el de integrales de superficie, consta de una serie de bloques, y cada bloque se dedica a calcular una integral de superficie. De esta manera, tiene asignado todo un subdominio activo y un punto en la cuchilla de un subdominio víctima. Los bloques tienen diferentes tonalidades para representar que cada uno utiliza un número de puntos en la integral independiente del que usen los demás.

Una vez ejecutado el *kernel* que calcula las integrales de superficie, y los que calculan el capacitivo (no mostrados en la Figura 4-8), se ejecuta el que calcula las integrales de línea. Éste está formado por bloques de tamaño máximo, y cada hilo se encarga de procesar toda una cuchilla, por lo que genera un valor final de la matriz de impedancias Z.

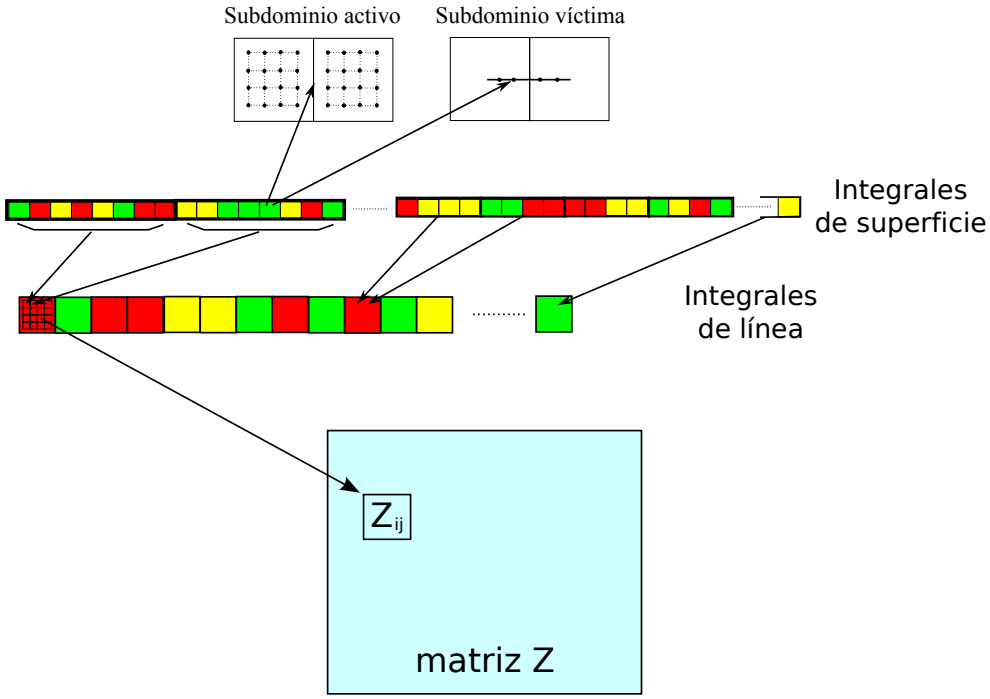


Figura 4-8. División del trabajo entre *kernels*.

La Figura 4-9 muestra la secuencia de invocación de los sucesivos *kernels*, esta vez incluyendo los que se encargan del acoplo capacitivo. El primero calcula las integrales de superficie del acoplo inductivo, el segundo las del capacitivo, el tercero corrige los casos en los que, en éste último, se debe usar diferente número de puntos de integración en cada subparche del subdominio activo, y el último de todos, a partir de los resultados de los anteriores, obtiene los elementos de la matriz de impedancias.

```
Kernel integrales superficie inductivo
Kernel integrales superficie capacitivo
Kernel corrección capacitivo
Kernel post-proceso
```

Figura 4-9. Secuencia de llamadas a los *kernels*.

4.4 Exploración de los recursos de CUDA.

En la sección anterior se describe nuestra primera adaptación del Método de los Momentos a una tarjeta gráfica. El proceso es largo y laborioso debido a que la GPU ejecuta el código de una forma diferente a como lo hace la CPU, por lo que el único objetivo era crear una versión del código que siguiera las líneas maestras de la programación en GPUs y que respetara los requisitos del método, de forma que proporcionara los mismos resultados que la versión que corre en la CPU.

Para esta sección, el objetivo es distinto. Una vez se cuenta con una versión correcta en cuanto a los resultados, se trata de evaluar

cómo influye en el rendimiento la forma de diseñar la aplicación. Por ejemplo, hay cálculos más indicados para la CPU y otros que es más beneficioso asignar a la GPU. También se puede estudiar cómo se asignan las tareas a *kernels*, y cómo se organizan éstos. Cuestiones como éstas pueden tener un efecto notable en el rendimiento final de la aplicación, y las conclusiones que se obtengan de su estudio se pueden extrapolar a otras aplicaciones diferentes, de forma que el trabajo realizado aquí se podrá aprovechar en otros diferentes.

El resto de la sección se organiza de la siguiente manera: en primer lugar se plantean algunas decisiones de diseño razonablemente sencillas y evidentes, por lo que ya están presentes en la versión inicial. A continuación, se pasa a tratar otras decisiones que, por ser menos evidentes, es preciso tratar con más detenimiento.

4.4.1 Utilización de la jerarquía de memoria.

Una de las primeras cuestiones a tratar es el uso de la jerarquía de memoria. La principal GPU de trabajo, la C2075, dispone de 6 GB de memoria global, además de 64 KB de memoria para constantes y 48 KB de memoria compartida para cada bloque de hilos. El reducido tamaño de la memoria de constantes desaconseja su uso para las matrices que caracterizan la geometría, que serían candidatas idóneas a residir aquí, por lo que, en principio, deben hacerlo en memoria global. Una pequeña mejora se puede conseguir almacenando al menos los punteros a esas matrices en memoria de constantes, ya que son únicos (los punteros, desde el punto de vista de la GPU, son constantes). El ahorro es mínimo, pero no cuesta nada, por lo que se opta por él. Igualmente se almacenan en memoria de constantes las tablas con los coeficientes de Gauss, y otras

constantes parecidas que se usan en los cálculos. Estos datos tienen una extensión considerablemente menor que las matrices de la geometría, y caben en la memoria de constantes perfectamente.

Dado que no pueden residir en memoria de constantes debido a su tamaño, y con el fin de reducir el tiempo de acceso a las mismas, se ha considerado la posibilidad de almacenar la parte de las matrices que utiliza cada bloque de hilos en la memoria compartida de ese bloque, dada la mayor velocidad de acceso de este tipo de memoria. Esta es una opción interesante en el caso de datos que se reutilizan varias veces. Dado que para almacenar datos en memoria compartida estos datos se han de llevar, en un primer paso, de la RAM del sistema a memoria global, y despues desde ahí a memoria compartida, según se fueran necesitando, el primer acceso sería a velocidad de memoria global, pero todos los siguientes se harían a velocidad de memoria compartida. Sin embargo, en el caso de la aplicación que se desarrolla en esta tesis, los datos de la geometría se utilizan una única vez, por lo que llevarlos hasta la memoria compartida no aporta ningún beneficio.

Sí se utiliza la memoria compartida para almacenar los resultados parciales de las integrales de superficie. Su uso en este caso sí es ventajoso: esta memoria es especialmente eficaz cuando almacena datos que se han generado en la propia GPU, como es el caso, y se tienen que reutilizar alguna vez más. Si se almacenan en memoria compartida, estos datos no necesitan llevarse a la memoria global, y con esto se mejora el tiempo de ejecución. A la memoria global sólo se llevan los resultados finales de las integrales.

Como último escalón de la jerarquía por tratar se encuentran los registros de los SMs. Sin embargo su uso lo gestiona mejor el compilador (analizando los tiempos de vida de las variables de los *kernels*), por lo que desde el diseño de la aplicación no se toma ninguna decisión al respecto.

4.4.2 Desenrollado de bucles.

Una de las técnicas que utilizan los compiladores para mejorar la ejecución en procesadores superescalares es el desenrollado de bucles [70]. La técnica consiste en una transformación de los bucles diseñados por el programador, de forma que en una iteración del bucle transformado se realice el trabajo de dos o más iteraciones del bucle original. Las ventajas residen, fundamentalmente, en el paralelismo a nivel de instrucción que se desvela y en la reducción del número de instrucciones de control del bucle ejecutadas.

```
for (i=0;i<MaxElementos;i++)
     vectorC[i]=vectorA[i]+vectorB[i];
```

Versión original

```
for (i=0;i<MaxElementos;)
{
     vectorC[i]=vectorA[i]+vectorB[i];
     vectorC[i+1]=vectorA[i+1]+vectorB[i+1];
     i+=2;
}
```

Versión desenrollada

Figura 4-10. Desenrollado de bucles.

La Figura 4-10 muestra un ejemplo de desenrollado. El bucle original recorre cada posición de los arrays calculando la suma. El bucle desenrollado los va procesando de dos en dos, por lo que tendrá la mitad de iteraciones.

La utilidad de esta técnica en el caso de una GPU es otra, dado que los *cores* no son superescalares, y las instrucciones de control del bucle se ejecutarían en paralelo (lo que no aumentaría el número de ciclos de ejecución). En una GPU el principal efecto beneficioso del desenrollado de bucles es la eliminación de saltos, que, como se ha explicado anteriormente, pueden llegar a tener un efecto negativo considerable sobre el rendimiento de la aplicación. Además, también pueden servir para ocultar latencias de memoria, ya que el compilador reordenará las instrucciones para espaciar los accesos.

Los bucles que contiene la aplicación no son demasiado grandes en términos de iteraciones. Las integrales de 10 puntos tienen bucles con 5 iteraciones, las de 4 con 2 y las de 2 no tienen bucles. De esta manera, el principal factor limitador del desenrollado en las CPUs, que es la presión sobre el banco de registros (cuanto mayor sea el factor de desenrollado, más registros se necesitan para contener las variables que corresponden a las distintas iteraciones condensadas en una sola del bucle transformado), no tiene un efecto demasiado grande en nuestro caso. Por consiguiente, se opta por hacer un desenrollado completo de los bucles.

4.4.3 Asignación de cálculos a la CPU o a la GPU.

En el diseño inicial de la aplicación para CUDA se ha trasladado a la GPU la parte de la aplicación que de forma más evidente se

beneficia de la misma: todo el cálculo de integrales, tal y como se indicó en la sección 4.3. La CPU se encarga de preparar los datos para los *kernels*, de reservar espacio en la memoria de la GPU para almacenamiento intermedio, de lanzar la ejecución de los mismos, y de sacar el resultado (la matriz Z) de la GPU. En esta sección se estudia la posibilidad de trasladar más cálculos a la GPU y su efecto sobre el rendimiento global de la aplicación.

Reducción de la cantidad de parámetros de los kernels

Los *kernels*, especialmente los que calculan las integrales de superficie, necesitan una serie de datos de la geometría. Por ejemplo, cada hilo se encarga de evaluar la función de Green en un punto del subdominio activo, y para ello necesita identificar ambos subdominios (víctima y activo) y obtener algunos datos referentes al cálculo que tiene que realizar. Estos datos se obtienen de las matrices que caracterizan la geometría. En nuestra versión inicial, la CPU lee estos datos (para todos los hilos), los almacena en un array, y los transfiere a la GPU (a memoria global). La opción que se baraja ahora es que cada hilo obtenga los suyos, accediendo directamente a las matrices que residen en la memoria global de la GPU. En la estructura de datos que prepara la CPU se dejarían únicamente la cantidad mínima de datos que permiten a cada hilo identificar a qué par de subdominios pertenece el cálculo que él realiza, y cuál es su tarea dentro de ese cálculo. De esta manera, el número de datos que contiene el array es lo más reducido posible. Optando por esta opción, el tamaño del array se reduce en un 400%.

La memoria global de la GPU es muy sensible a la alineación de los accesos. Si hilos consecutivos acceden a posiciones de memoria

consecutivas, los accesos se dice que están alineados. Si hilos consecutivos acceden a posiciones no consecutivas, no lo están. La GPU está diseñada para que los hilos puedan hacer accesos a memoria en paralelo (es decir, todos los accesos en el tiempo de 1 solo), siempre que esos accesos estén alineados. En caso contrario, se pueden dar varios grados de serialización, y los accesos se realizan en grupos: dentro de cada grupo los accesos son paralelos, pero los grupos entre sí no lo son. Los grupos de accesos pueden ir desde 2 hasta tantos grupos como hilos, en el caso peor.

Las matrices que contienen los datos de la geometría tienen varias dimensiones, lo que hace que, generalmente, los accesos a las mismas no sean alineados. Esto hace que no sea trivial la elección entre ambas opciones. Por un lado, la CPU prepara los datos de los hilos en serie, mientras que la GPU lo podría hacer en paralelo. Pero, por otro lado, los accesos a memoria de la GPU serán, en su mayoría, no alineados, lo que, en la práctica, podría serializar la ejecución de los hilos.

Hay más factores a tener en cuenta, sin embargo: si los datos se preparan en la CPU, enviarlos a la GPU lleva un tiempo, mientras que la GPU los obtiene de las matrices, que ya han sido transferidas previamente a la GPU (los *kernels* las necesitaban para obtener los demás datos que usan). Además, el array que utiliza la CPU para almacenar estos datos ocupa espacio en la memoria de la GPU, que se debe quitar de otro sitio, mientras que, si los datos los obtiene cada hilo cuando los va a usar, no necesita almacenarlos.

El capítulo de resultados muestra las medidas de tiempos realizadas. La conclusión que se puede sacar es que el paralelismo de

la GPU compensa los accesos no alineados. Además, la reducción del tiempo que se invierte en trasladar a la GPU los parámetros de los *kernels* es considerable.

Traslado de cálculos a la GPU.

Además de los datos que cada hilo utiliza de modo individual, hay una serie de datos que deben calcularse para cada par de subdominios víctima-activo, es decir, para cada bloque de hilos (los que participan en la misma integral de superficie). Por ejemplo, la distancia entre ambos subdominios o el número de puntos que se deben utilizar en la integral. Hay que recalcar que estos datos deben calcularse, no simplemente leer las matrices de la geometría. El planteamiento de la cuestión es análogo al de la sección inmediatamente anterior: en la GPU los accesos a los datos no son alineados (en el caso más general), por lo que se secuencializarán en gran medida. La diferencia es que, en este caso, todos los hilos que participan en la misma integral de superficie realizarán el mismo cálculo. Si se calcula en la CPU, no se calcula un conjunto de datos para cada hilo, sino para cada bloque de hilos, lo que supone menos trabajo. Debido a esta diferencia, los tiempos de ejecución probablemente mostrarán menos diferencias entre ambas alternativas que en la sección anterior.

Si se opta por llevar estos cálculos a la GPU hay, al menos, dos formas de hacerlo. Una de ellas supone crear un nuevo *kernel* que haga los cálculos, con un hilo por cada juego de datos que haya que generar. Los datos generados por este *kernel* deben almacenarse temporalmente (en memoria global) para ser leídos por el siguiente *kernel*, el que calcula las integrales de superficie. La Figura 4-11

muestra esta relación. El espacio total es del orden de 30 Bytes por cada par de subdominios. Como referencia, una geometría de 1500 subdominios necesitaría unos 70 MB para estos datos.

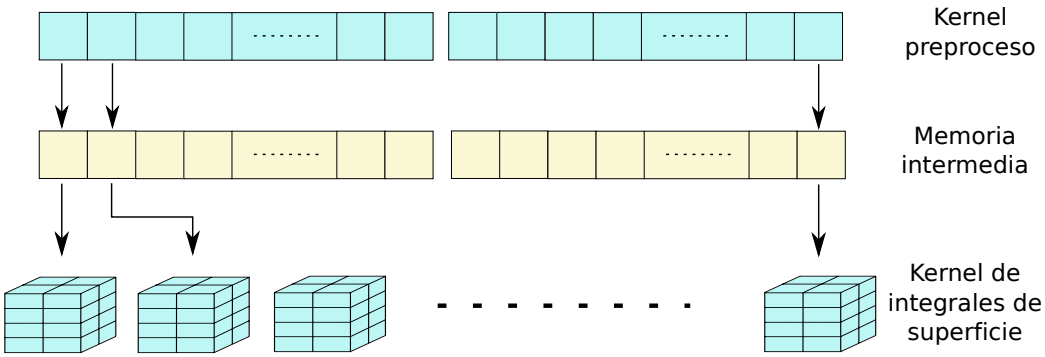


Figura 4-11. Preproceso de datos.

Como esta cantidad de espacio no es determinante (y además, puede ser liberada al terminar la ejecución de las integrales de superficie), la segunda forma de calcular estos datos en la GPU, que no necesita almacenamiento temporal, no supone una gran variación con respecto a la primera. En este caso se trata de que, dentro del mismo *kernel* que calcula las integrales de superficie, cada hilo calcule los datos (lo que implica que varios hilos calcularán el mismo juego de datos). La ventaja con respecto a la primera forma es, como se ha mencionado, que no se necesita almacenamiento temporal, y, además, que no es necesario lanzar un *kernel* adicional, lo cual puede ser beneficioso ya que lanzar un *kernel* conlleva un recargo en tiempo (el tiempo que el *runtime* tarda en prepararlo y ponerlo en ejecución).

Como se menciona más arriba, el almacenamiento temporal necesario para hacer llegar los datos de un *kernel* al siguiente no es muy significativo, y las mediciones de tiempo muestran que el recargo

por el lanzamiento del *kernel* adicional tampoco lo es. La sencillez del diseño de la aplicación recomienda elegir la primera opción de cálculo de datos en la GPU considerada: utilizar un *kernel* adicional. La discusión queda reducida, pues, a comparar esta opción con la de realizar los cálculos en la CPU. El resultado va a ser, nuevamente, favorable al traslado del cálculo a la GPU.

4.4.4 Organización de los bloques de hilos.

El manual de programación para CUDA de NVIDIA [45] pone un gran énfasis en que el número de hilos por cada bloque debe ser lo suficientemente grande como para que la GPU tenga margen de maniobra cuando se encuentra un acceso a memoria global. Lo que se pretende es que, al llegar la ejecución de un hilo a un acceso a memoria global (considerablemente lento para las velocidades de los *cores* y los accesos a registros y memoria compartida), la GPU disponga de otros hilos sobre los que seguir trabajando mientras llegan los datos pedidos. De esta manera, los retardos de acceso a memoria global se ven enmascarados con la ejecución de otros hilos. Si no hay hilos suficientes, el trabajo se debe detener hasta que lleguen los datos leídos.

En la primera versión de nuestra aplicación para GPU, se asigna un bloque de hilos a cada integral de superficie. Un sencillo cálculo permite averiguar que, si cada hilo del bloque realiza la evaluación de la función de Green en uno de los puntos de la malla bidimensional sobre la que se realiza la integral, las integrales de 10 puntos necesitarán 200 hilos (10 puntos x 10 puntos x 2 subparches activos), las de 4 puntos necesitarán 32 y las de 2, 8. Así, las integrales de 10 puntos se puede pensar que tienen un número razonable de hilos por

bloque, dado que el máximo es 512 (NVIDIA recomienda, al menos 192). Pero en el caso de las integrales de 4 y 2 puntos, el número de hilos es claramente insuficiente.

La facilidad de programación nos llevó a diseñar los bloques de esa manera, pero para conseguir rendimiento es preciso aprovechar mejor la capacidad de cada bloque de hilos. En esta sección se evalúa cómo afecta al rendimiento un mal diseño de los bloques de hilos de un *kernel*. Para ello, se va a comparar el tiempo de ejecución de la versión de CUDA con un bloque de hilos para cada integral de superficie con otra en la que se compactan los bloques originales. Esencialmente se trata de que cada nuevo bloque de hilos calcule varias de las integrales que antes se calculaban con uno sólo. O, en otras, palabras, de diseñar nuevos bloques de hilos, formados por varios de los bloques antiguos. En esta línea, los nuevos bloques se podrían llamar "Súper Bloques", para distinguirlos de los antiguos. Es decir, cada "Súper Bloque" contiene varios de los bloques originales. Al grupo de hilos que calcula una integral de superficie lo llamaremos "paquete de hilos" de aquí en adelante. Evidentemente esta es una terminología propia que utilizamos para distinguir unos de otros. No está basada en ningún término "oficial" de NVIDIA.

Así pues, teniendo en cuenta el número de hilos que necesita cada integral de superficie, se crean tres tipos de "SuperBloques": de 200 hilos (para integrales de 10 puntos), de 16 paquetes de 32 hilos cada uno (para integrales de 4 puntos: 32 x 16 = 512) y de 64 paquetes de 8 hilos cada uno (para integrales de 2 puntos: 64 x 8 = 512). En el caso de las integrales de 4 y de 2 puntos se aprovecha al máximo el número de hilos de que dispone un bloque de hilos en CUDA

(nuestros "SuperBloques" ahora). En el caso de las integrales de 10 puntos, agrupar 2 en un "SuperBloque" supone más trabajo de diseño de la aplicación que mejoras en el rendimiento, ya que son muy poco frecuentes. De forma análoga se procede para el cálculo del acoplo capacitivo. La Figura 4-12 muestra cómo sería el agrupamiento de paquetes de hilos forma gráfica. Se muestra un caso caso hipotético (no presente en nuestra aplicación) de agrupación de paquetes de 16 hilos.

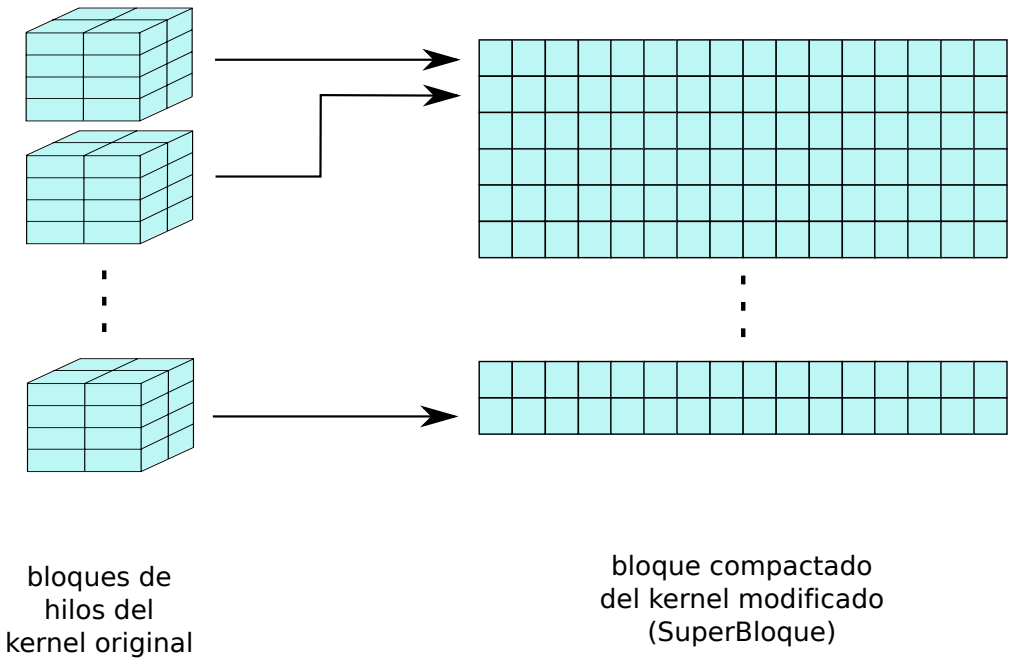


Figura 4-12. Formación de "SuperBloques".

Al usar este nuevo diseño del *kernel* (en "SuperBloques"), el rendimiento se ha de ver afectado por varios factores: por un lado, aprovechar mejor las posibilidades de los bloques en cuanto a los hilos que contienen debe suponer una mejora clara del rendimiento. Por otro lado, es aconsejable lanzar *kernels* diferentes para cada tipo

de SuperBloque, lo que conllevará algún tipo de retardo. Si no se hace así, un SuperBloque único que se adaptara al tipo de integral de superficie requeriría un salto condicional al principio (al estilo del del *kernel* de la versión inicial) para ejecutar un código u otro. Este salto también supondría una pérdida de rendimiento. La experiencia obtenida del apartado anterior indica que lanzar un número reducido de *kernels* no tiene un efecto demasiado malo sobre el rendimiento, por lo que, para mantener la sencillez de la estructura del código se opta por separar las integrales en *kernels* diferentes.

El último factor que se ha de considerar, con respecto al rendimiento, tiene que ver con que cada SuperBloque debe saber qué par de subdominios le corresponde. Para esto, hay que tener en cuenta:

- el número de puntos de integración en la cuchilla determina cuántos paquetes de hilos hacen falta para cada par de subdominios víctima-activo (es decir, cuántas integrales de superficie).
- el número de puntos de cada integral de superficie determina cuántos hilos tiene el correspondiente paquete (es decir, si esa integral de superficie se calcula en un *kernel* o en otro).

Para facilitar la contabilidad y permitir que cada paquete de hilos de cada SuperBloque sepa qué está calculando, se crean listas. En concreto, una para cada *kernel*. Por medio de esa lista, los paquetes de hilos pueden saber en qué par de subdominios víctima-activo se encuadra su cálculo, y qué integral de superficie (de qué punto de la cuchilla) les corresponde calcular a ellos. El flujo de datos entre CPU y GPU para todo esto se muestra en la Figura 4-13.

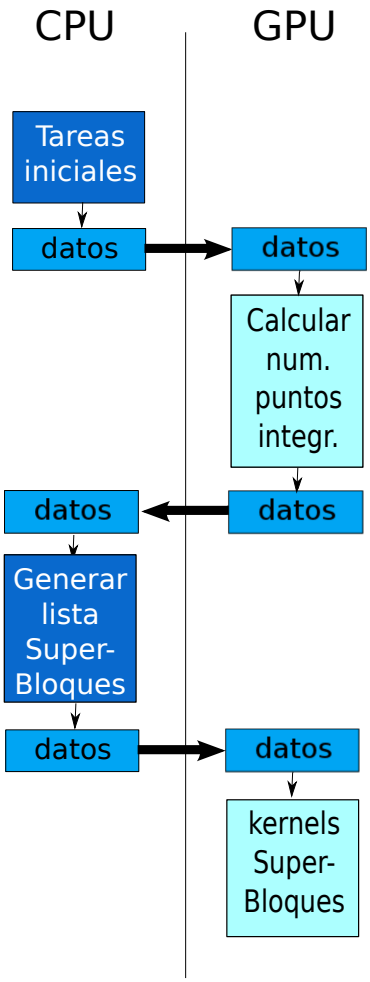


Figura 4-13. Flujo de datos para preparar "SuperBloques"

Después de las tareas de inicialización, ejecutadas en la CPU, se transfieren los datos de la geometría y demás constantes a la GPU. En ese momento se lanza el *kernel* que, entre otros datos, calcula los puntos de integración de cada integral, que son devueltos a la CPU. Ésta, a partir de ellos, obtiene las listas y las envía a la GPU. A partir de ese momento se empiezan a lanzar los *kernels* de las integrales de superficie.

Este aumento del tráfico con la GPU debe tener su efecto en el rendimiento. Sin embargo, los resultados obtenidos, como se muestra en el capítulo 7, indican que organizar los *kernels* de esta manera es muy ventajoso para la aplicación. En general, se puede aceptar la conclusión de que los bloques de hilos deben aprovecharse todo lo posible.

4.4.5 Reducción del almacenamiento intermedio.

Al plantearnos el diseño de nuestra aplicación en la GPU, hemos recurrido a una directriz utilizada muy frecuentemente en la programación: subdividir el trabajo en tareas. Este planteamiento se ha intensificado a medida que se exploraba el espacio de diseño de la misma, de forma que el resultado final es una serie de *kernels*, razonablemente específicos, que van dando los pasos necesarios para finalmente generar la matriz de impedancias Z.

Cuando dos *kernels* realizan, cada uno, un paso dentro de un procedimiento computacional, es frecuente que los datos que genera uno deban ser procesados por el otro. Esto ocurre también con nuestra aplicación: el primer *kernel* calcula los datos procesados a partir de las características de la geometría que después utilizan los *kernels* posteriores, los que calculan las integrales de superficie, y los resultados de éstos son, a su vez, procesados por el *kernel* que realiza las integraciones a lo largo de las cuchillas.

Los datos que pasan de un *kernel* a otro deben hacerlo, en la variante más eficaz, a través de la memoria global. Y esto limita el tamaño del problema que se puede tratar, ya que ese espacio que se

utiliza para almacenamiento temporal se está restando del que se podría utilizar para el objetivo final de los cálculos: la matriz Z.

De todos los datos que se almacenan de forma temporal en la GPU, el que tiene un peso mayor y, por tanto, limita más el tamaño de la matriz Z que se puede tratar, es la estructura que almacena los resultados de las integrales de superficie para que puedan ser utilizados al integrar sobre las cuchillas. Los demás datos temporales no suponen demasiado recargo, pero una de las optimizaciones posibles sería reducir o eliminar la necesidad de ese espacio temporal.

La Figura 4-14 muestra esquemáticamente la situación: los *kernels* que calculan las integrales de superficie (de los que se muestra sólo uno, pero que son varios) dejan sus resultados en un almacenamiento intermedio, de donde lo toma el que calcula las integrales de línea sobre las cuchillas. Éste, a su vez, genera la matriz Z. Cada paquete de hilos (en la figura, una fila del SuperBloque) tiene uno sombreado, que es el que se encarga de almacenar los datos de todo el paquete de hilos.

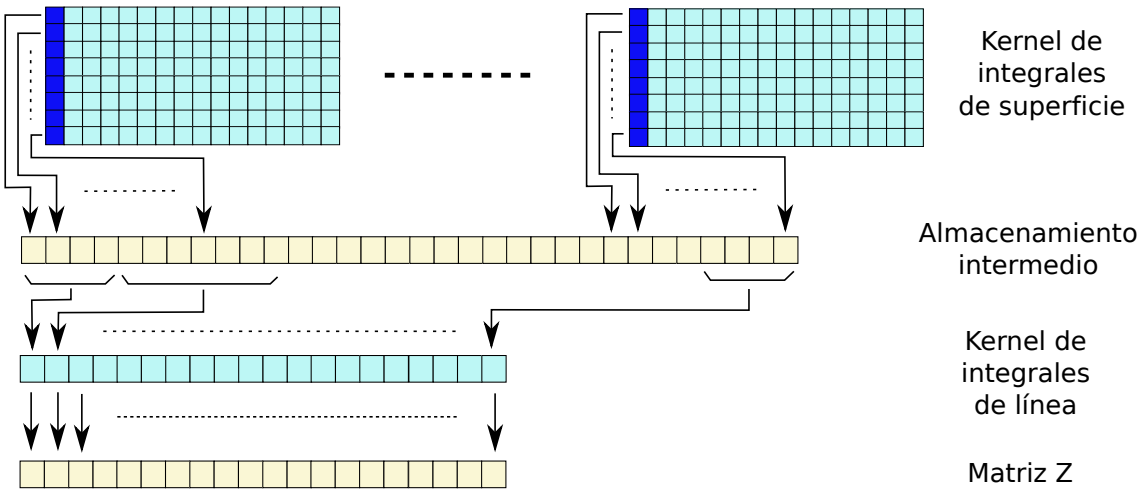


Figura 4-14. Situación anterior.

Esta sección se dedica a tratar la cuestión del almacenamiento temporal en la GPU. El objetivo es reducirlo, de forma que el problema que se pueda tratar sea mayor. Y dado que la mayor parte del almacenamiento temporal de la aplicación (en la GPU) se dedica a los resultados de las integrales de superficie, lo que se busca es eliminar su necesidad. Para ello, es preciso que los *kernels* de integrales de superficie almacenen sus resultados directamente en la matriz Z.

Cada elemento de la matriz Z es la suma ponderada, según el método de Gauss, de los resultados de las integrales de superficie. Para que cada paquete de hilos almacene su resultado directamente en la matriz Z, lo único que hay que hacer es ponderar el valor de esa integral y sumarlo al valor del elemento correspondiente de la matriz Z. La primera parte es sencilla: cada paquete de hilos sabe el punto de la cuchilla que le corresponde, por lo que puede calcular su coeficiente de Gauss y multiplicarlo por la integral que ha calculado. El problema surge a la hora de hacer la suma: cada elemento de Z es la suma de los resultados de varios paquetes de hilos, y es muy posible que más de uno intente actualizar el mismo valor al mismo tiempo, ya que seguramente se ejecuten a la vez. Si no se toman precauciones, puede ocurrir que el resultado de la suma no sea correcto. Ésto último es lo que muestra la Figura 4-15.

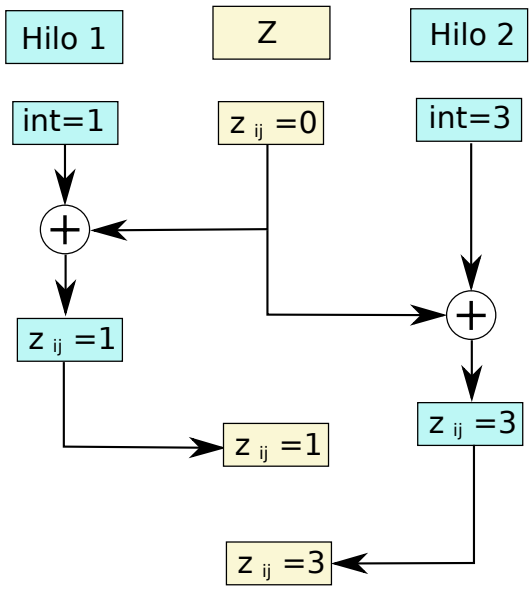


Figura 4-15. Actualización incorrecta.

En la figura, cada hilo (que representa al hilo que se encarga de actualizar el valor calculado por el paquete) tiene el resultado de su integral (dato "int"), y debe sumarlo al elemento (i,j) de la matriz Z. Si el hilo 2 lee el valor de Z antes de que el hilo 1 lo actualice, y lo almacena después de él, el resultado final es incorrecto. Para solucionar esto, es preciso introducir algún mecanismo de sincronización, de forma que el hilo 2 no pueda leer el valor de Z hasta que esté actualizado, como muestra la Figura 4-16. En ella, se impide la lectura del elemento de Z por parte del hilo 2 hasta que el hilo 1 lo actualiza, con lo que el resultado es correcto.

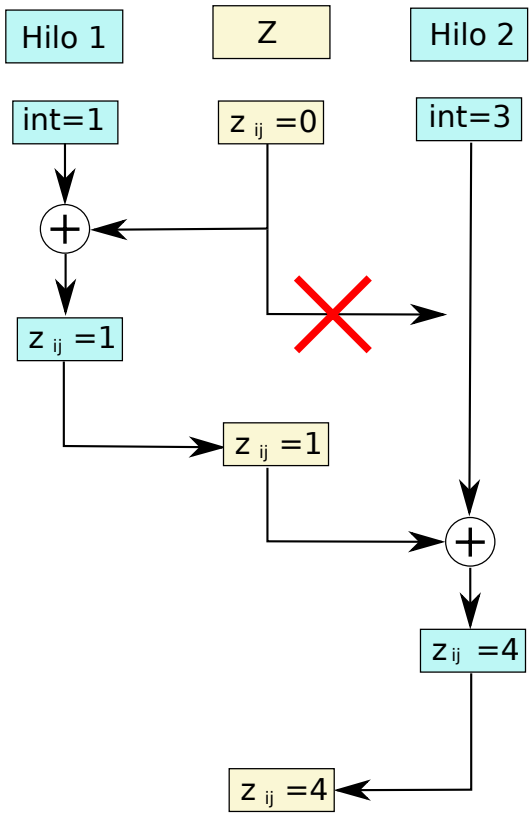


Figura 4-16. Actualización correcta (con sincronización).

Esta forma de actualizar valores se denomina "actualización atómica". Las operaciones atómicas para números en coma flotante aparecen a partir de la "computing capability" 2.0. Durante gran parte del desarrollo de esta tesis, la única forma de realizar una actualización atómica de un dato en coma flotante fue utilizar la función `atomicCAS` (*atomic Compare-And-Swap*, comparación e intercambio atómicos). Esta función lee un valor de una posición de memoria, lo compara con otro que le es proporcionado, y si coinciden, lo sustituye por un segundo valor que también se le proporciona. Además, la función trabaja con números enteros (no había versión para números en coma flotante), por lo que hay que hacer una pequeña adaptación.

La forma de usarla es leer el valor que se quiere actualizar, incrementarlo (en un registro del hilo), y luego intentar actualizarlo. Sólo se realizará la actualización si el valor leído anteriormente coincide con el leído al intentar actualizar. Si no, se debe repetir el proceso hasta que tenga éxito. Además, dado que la función trabaja con enteros, para el caso de datos en coma flotante (el nuestro) es necesario convertir el valor leído (en formato entero) a coma flotante, y realizar la conversión inversa al actualizar. Todo esto está descrito en la guía de programación de CUDA [45].

La Figura 4-17 muestra cómo queda la aplicación al eliminar el almacenamiento intermedio. Se ha eliminado el *kernel* que realiza la integral de linea sobre la cuchilla, y los *kernels* de las integrales de superficie actualizan directamente la matriz Z. Como se puede ver, varios paquetes de hilos actualizan el mismo elemento de Z, que es lo que hace necesaria la sincronización.

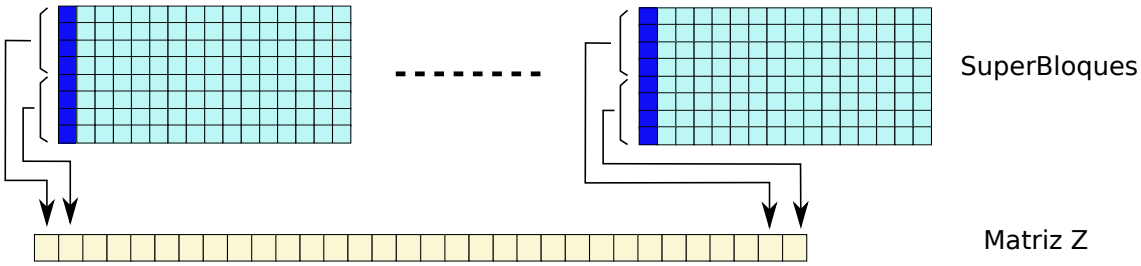


Figura 4-17. Situación actual.

La transformación del código para que las integrales de superficie se almacenen actualizando directamente la matriz Z tiene dos resultados: por un lado se evita dedicar un espacio de la memoria global de la GPU a resultados intermedios, lo que libera memoria para procesar geometrías más grandes. Pero, por otro se produce una

pérdida de rendimiento, ya que habrá hilos que se queden esperando a poder actualizar el valor correspondiente de la matriz Z. En el capítulo de resultados se muestran valores para ambos efectos: cuánto más grande puede ser la matriz, y qué pérdida de rendimiento se produce. A la hora de optar por eliminar o no el almacenamiento intermedio debe tenerse en cuenta qué es más importante: si el tiempo de cálculo o el tamaño de la geometría. En este sentido, hay que tener en cuenta que eliminar el almacenamiento intermedio puede resultar más eficiente que no hacerlo si la geometría a tratar es grande: si no cabe entera en la GPU, puede ser necesario dividirla en trozos, y eso sí que ralentiza el proceso. En este caso, poder procesarla en un solo trozo, aunque con un método más lento, puede ser la forma más eficiente de tratar el problema.

4.5 Conclusiones.

De todo el trabajo empleado para el desarrollo de la aplicación en este capítulo es posible extraer algunas conclusiones interesantes. Cuando se crea una aplicación utilizando CUDA, es preciso tener en cuenta la forma en la que se ejecucta en la GPU. Así, es conveniente diseñar la aplicación de forma que los hilos de la GPU realicen tareas lo más parecidas posible. El ideal es que se ejecuten exactamente las mismas instrucciones pero con datos diferentes. Esto no es siempre posible, por lo que, si las tareas asignadas a los hilos difieren demasiado, hay que plantearse si agrupar las tareas en *kernels* separados cuyos hilos sí sean suficientemente parecidos. Es lo que se ha hecho aquí en un primer paso con las integrales de superficie: se han separado en tres tipos distintos, uno para cada número de puntos de integración.

También se debe trasladar a la GPU todo el trabajo posible que cumpla las condiciones necesarias (es decir, si es paralelo, si todos los hilos de GPU van a hacer el mismo trabajo, etc). La forma en la que los hilos de GPU realicen los accesos a memoria es una consideración de orden secundario: en general, si el número de hilos es suficiente, es preferible llevar el trabajo a la GPU.

Por último, es muy aconsejable crear bloques con el mayor número de hilos posible. De esta manera se enmascaran las esperas de los hilos al acceder a memoria, por ejemplo. Si hay suficientes hilos, mientras unos esperan el resto puede ir avanzando trabajo útil que de otra forma (si no estuvieran), habría que hacer después.

5 Aceleración del Método de las Funciones Base Características.

5.1 Introducción.

En este capítulo se describe la implementación en CUDA del código que aplica el Método de las Funciones Base Características (CBFM, '*Characteristic Base Function Method*'), tal y como se describió en el Capítulo 3. El CBFM está pensado para transformar la matriz de acoplos del sistema en otra mucho más reducida, pero no especifica cómo se debe obtener esa matriz. En nuestra implementación, la matriz se obtiene por el Método de los Momentos (MoM, '*Method of Moments*'), aprovechando que se dispone del código que lo hace (descrito en el Capítulo 4).

El CBFM está especialmente indicado para geometrías grandes que deben procesarse por bloques. Sin embargo, con el objetivo de utilizarlo como referencia, se incluye en primer lugar, en el apartado 5.2, una versión del método que procesa toda la geometría en un solo bloque. A continuación, ya en el apartado 5.3, se describe la implementación más general del CBFM.

5.2 Aplicación del CBFM a geometrías en un único bloque.

Si la geometría que se está analizando tiene un tamaño que permite que sea procesada por la GPU en un único bloque, el proceso es más sencillo y corresponde a lo descrito en el apartado 3.2.1. Una vez obtenida la matriz de impedancias del Método de los Momentos, se obtiene el espectro de ondas planas (PWS, '*Planar Wave*

Spectrum') y se resuelve el sistema de la ecuación (3.10) para obtener las corrientes inducidas por cada onda del espectro. A continuación se aplica la ortogonalización, por medio de la SVD, como se indica en la ecuación (3.41), y de ahí se obtienen las Funciones Base Características (CBFs, '*Characteristic Basis Functions'*). De ellas, se retienen únicamente aquellas que tienen un Valor Singular asociado mayor. En concreto, se filtran los Valores Singulares que sean menores a 1/500 veces el mayor de ellos.

Las CBFs se pueden poner en función de las funciones de base y de prueba del Método de los Momentos, tal como indican las ecuaciones (3.42) y (3.43). Partiendo de estas ecuaciones, y tal y como se explica en el apartado 3.2.1, la matriz de impedancias de las CBFs se puede obtener a partir de la matriz de impedancias del MoM. La ecuación (3.47) muestra la relación entre ambas en forma matricial:

$$ZR = U^{*}ZU$$

La matriz U es el resultado de aplicar la SVD a las corrientes que induce en la geometría el PWS, una vez filtradas las CBFs según su Valor Singular asociado. La expresión U*, como se indicó en su momento, es la transpuesta conjugada de U. La Figura 5-1 muestra el pseudocódigo de todo el proceso.

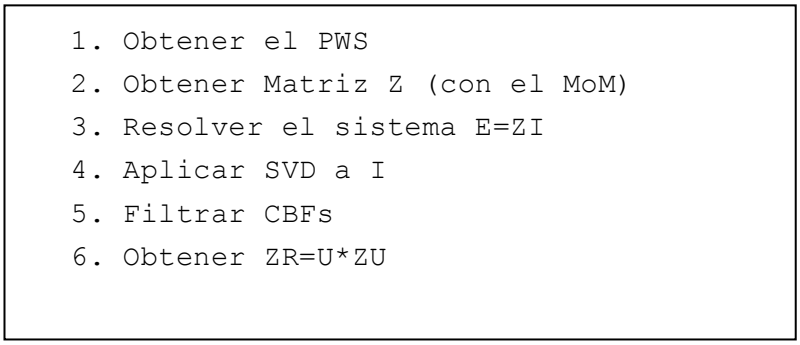


Figura 5-1. Algoritmo del CBFM para geometría en un único bloque.

El PWS se obtiene en la CPU de la misma forma que en la aplicación original, y la matriz Z del Método de los Momentos se calcula siguiendo los pasos descritos en el Capítulo 4. La parte novedosa está a partir de este punto. En primer lugar, es preciso resolver el sistema de la ecuación (3.10) utilizando como excitaciones las obtenidas del PWS y la matriz Z del MoM. El sistema se puede resolver de cualquier manera válida, y en este trabajo aprovechamos que existe una versión de las subrutinas de álgebra lineal LAPACK ('*Linear Algebra PACKage*') [51] adaptadas a CUDA, denominadas CULA [50]. Entre ellas se encuentra una, `culaDeviceCgesv`, que resuelve un sistema lineal de ecuaciones con coeficientes complejos, que es la adecuada a este paso. La rutina requiere los datos en la memoria de la GPU, y, dado que la matriz Z se ha calculado precisamente ahí, no es necesario sacarla para seguir con el proceso. Lo único necesario es trasladar a la GPU las excitaciones del PWS e invocar la rutina. El resultado es la matriz J de la ecuación (3.41), y se genera en la propia GPU.

El siguiente paso es aplicar la SVD a la matriz J. De nuevo es posible hacer uso de una rutina de CULA, en este caso `culaDeviceCgesvd`, que realiza esta tarea. La rutina tiene como resultado la matriz U y un vector de valores singulares S. Este vector se debe analizar para encontrar el valor singular a partir del cual se realiza el descarte, el valor umbral o de corte. El convenio de esta función es que los valores del vector S están ordenados de mayor a menor, por lo que lo único que hay que hacer es recorrerlo en orden. Como se ha indicado ya, se utiliza un factor de 500 como límite, de forma que los valores que sean más de 500 veces más pequeños que el valor singular mayor se descartan. Esta búsqueda es más eficiente (y más sencilla de programar) en la CPU, por lo que el vector S se transfiere de la memoria de la GPU a la de la CPU para procesarlo. El coste de esta operación es desdeñable en comparación con el resto de tareas del proceso, por lo que merece la pena hacerla así.

Una vez localizado el valor singular de corte, la matriz U (que se ha mantenido en la GPU) se modifica de la forma correspondiente. Dado que las CBFs son vectores columna, el paso consiste en ignorar las columnas a partir de la correspondiente al valor singular de corte, el umbral, como indica la Figura 5-2. Por simplificar la notación, desde este momento se utilizará el nombre U para referirse a la matriz U modificada, ya que la matriz U completa no se utiliza más.

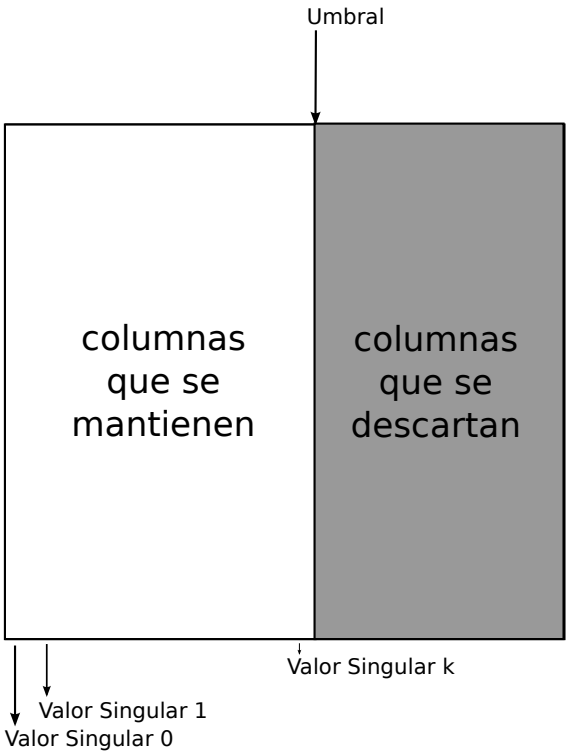


Figura 5-2. Matriz U.

La matriz U gobierna las transformaciones entre CBFs y subdominios. Específicamente, la ecuación (3.41) describe cómo se obtiene la matriz ZR a partir de la matriz Z por medio de la matriz U, por lo que lo único que queda por hacer es realizar las multiplicaciones de matrices. Para ello se utiliza la librería cuBLAS [71], que es una adaptación a CUDA de las Subrutinas de Álgebra Lineal Básica (BLAS, '*Basic Linear Algebra Subroutines*') [49]. Entre estas subrutinas se encuentra una, `cublasCgemm`, que multiplica matrices de números complejos en la propia GPU. Se usa dos veces, una para cada producto de matrices, y el resultado es la matriz ZR.

A partir de aquí, la matriz ZR está lista para ser utilizada de la forma que se necesite. Puede ser retirada de la GPU y almacenada para cálculos posteriores, o utilizarse inmediatamente. Por ejemplo, si

se quiere analizar el efecto de una onda sobre la geometría, lo único que hay que hacer es convertir las excitaciones que provoca esa onda al espacio de las CBFs por medio de la matriz U:

$$E' = U^* E \tag{5.1}$$

Donde, de nuevo U* es la transpuesta conjugada de U. Una vez obtenida la matriz E', se resuelve el sistema:

$$E' = Z' J' \tag{5.2}$$

Y se obtiene la matriz de corrientes J′. Para obtenerlas en función de los subdominios, se aplica la transformación inversa a (5.1):

$$J = UJ' \tag{5.3}$$

Todo esto se puede hacer en la propia GPU utilizando las rutinas ya mencionadas. Para resolver el sistema lineal se utiliza `culaDeviceCgesv`, y para los productos de matrices `cublasCgemm`.

5.3 Aplicación del CBFM a geometrías en varios bloques.

Si la geometría es demasiado grande para ser tratada en un solo bloque, es preciso dividirla en varios. El proceso de cada uno de los bloques es muy parecido al descrito en el apartado anterior, pero tiene particularidades que tienen que ver con el hecho de que cada bloque no es un ente aislado, sino que forma parte de una única geometría. En este apartado se describen los pasos que se han de dar con este tipo de geometrías.

Cada bloque resultante de la división de la geometría va a ser procesado, como se ha dicho, de una forma muy parecida a como se

describe en el apartado 5.2. Pero, para tener en cuenta que todos los bloques forman parte de una única geometría, a la hora de calcular la matriz del MoM se va a ampliar cada bloque con los subdominos de otros bloques que sean contiguos a los bordes del mismo, hasta una profundidad determinada. Con esta modificación, lo que se consigue es mantener la continuidad eléctrica entre bloques, a pesar de que se procesen de forma independiente. El bloque así modificado se denomina *bloque extendido*, para distinguirlo del bloque *inicial* u *original*, y los subdominios añadidos forman la *franja*. La Figura 5-3 muestra todo esto de forma gráfica.

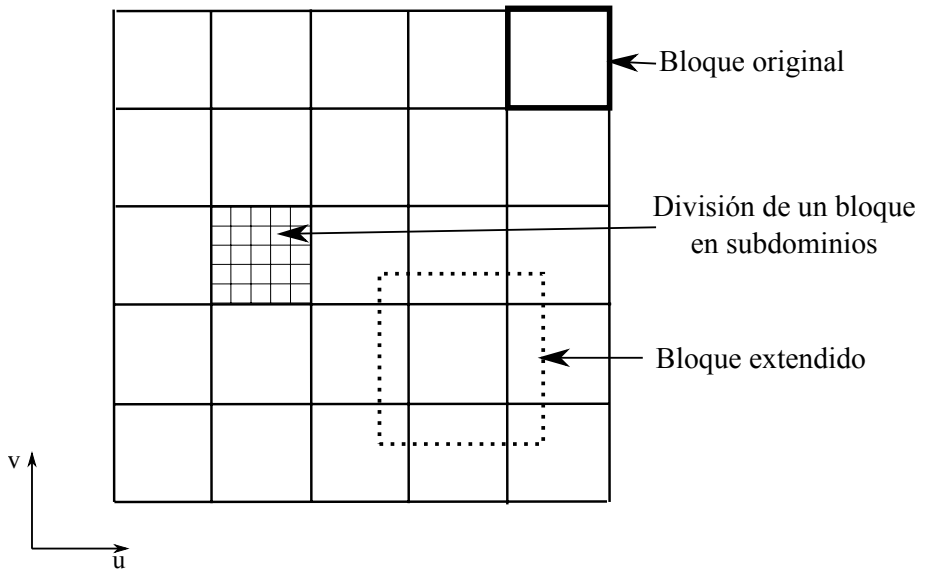


Figura 5-3. Bloques original y extendido.

Una vez calculada la matriz del MoM para el bloque extendido, se obtienen las excitaciones del PWS que corresponden a los subdominios del mismo, y se resuelve el sistema de la ecuación (3.41), pero con el bloque extendido. Las corrientes que se obtienen ya incorporan la continuidad eléctrica, por lo que se pueden retirar las

correspondientes a la franja. Desde este punto, el proceso es igual que en el apartado anterior: se aplica la SVD a las corrientes y se obtiene la matriz U de conversión al espacio de las CBFs.

El hecho de procesar la geometría en bloques también introduce, consecuentemente, una división en submatrices de la matriz general del sistema. Las submatrices corresponden a acoplos de unos bloques con otros. Y esta subdivisión también se manifiesta en la matriz de acoplos reducida. La ecuación (3.48) describe la división en submatrices de la matriz reducida. La matriz del MoM tiene una estructura similar.

Cuando se procesa en varios bloques, hay que calcular las matrices de acoplos de todos los bloques con todos los demás. La ecuación (3.47) es aplicable a acoplos de un bloque consigo mismo. Si los bloques activo y víctima son diferentes, la transformación debe tener esto en cuenta. La expresión más general es:

$$ZR_{ij} = U_i^* Z_{ij} U_j \tag{5.4}$$

Es decir, para obtener ZR_{ij} son necesarias las matrices U_i y U_j, que se obtienen cuando se calculan ZR_{ii} y ZR_{jj}, por lo que no es necesario obtenerlas otra vez. Esto sugiere un orden en el procesamiento de las submatrices de la matriz global: en primer lugar se procesan las submatrices que corresponden a acoplos de un bloque consigo mismo (que son las que están situadas en la diagonal) y se obtienen además las matrices U_i. Una vez hecho esto, se procesan las submatrices que están fuera de la diagonal. El algoritmo general lo describe la Figura 5-4.

Como se puede ver, se tratan de forma diferente las submatrices de la diagonal y todas las demás. Para las primeras (1), se obtiene

(1.1) la matriz de acoplos extendida del bloque (ZE_{kk}) según el MoM (de la misma forma que hasta ahora), únicamente teniendo en cuenta la franja. La matriz se genera en la GPU. A continuación (1.2) se obtienen las corrientes inducidas (matriz JE_k), para lo que es necesario utilizar la rutina `culaDeviceCgesv` ya mencionada, que calcula la solución del sistema lineal:

$$EE_k = ZE_{kk} JE_k \tag{5.5}$$

De la matriz JE_k se eliminan las corrientes correspondientes a los subdominios de la franja (1.3), lo que da como resultado la matriz J_k. Esto se hace en la propia GPU, por medio de un *kernel*, y se dejan los resultados también en la GPU. El resto es igual al proceso de la geometría en un solo bloque: se le aplica a J_k la SVD con la rutina `culaDeviceCgesvd` (1.4), con lo que se obtiene la matriz U_k y el vector de valores singulares S_k. Éste último se lleva a la CPU para encontrar el valor singular de corte y con él se transforma la matriz U_k completa en la matriz U_k filtrada (1.5). Esta matriz se utiliza para transformar Z_{kk} (obtenida (1.6) de retirar de ZE_{kk} los elementos correspondientes a subdominios de la franja por medio de un *kernel* en la GPU) en ZR_{kk} (1.7) y se reserva para su uso posterior. En la implementación actual las matrices U_k se dejan en la GPU, ya que no ocupan demasiado espacio, pero si fuera necesario podrían descargarse a la CPU.

```
;preparar datos
.
.
;generar submatrices de la diagonal
1. Para cada bloque de la geometría:
   1.1 Calcular la matriz de acoplo extendida ZE_kk
   1.2 Calcular las corrientes inducidas extendidas JE_k
   1.3 Extraer J_k de JE_k
   1.4 Aplicar la SVD a J_k para obtener U_k
   1.5 Aplicar el umbral a los vectores de Uk
   1.6 Extraer Z_kk de ZE_kk
   1.7 Obtener ZR_kk a partir de Z_kk y U_k
.
.
;generar submatrices fuera de la diagonal
2. Para cada submatriz fuera de la diagonal
   2.1 Calcular la matriz de acoplo original Z_kl
   2.2 Obtener ZR_kl a partir de Z_kl, U_k y U_l
```

Figura 5-4. Algoritmo general del CBFM.

El proceso de las submatrices que no están en la diagonal es mucho más sencillo. En primer lugar (2.1) se obtiene la matriz de acoplos, Z_{kl}, sin necesidad de añadir la franja, por el método explicado en el capítulo 4. Una vez obtenida, y utilizando las matrices U_i correspondientes, se obtiene la submatriz reducida (2.2), de acuerdo a la ecuación (5.4). Para los productos de matrices, se vuelve a utilizar la rutina `cublasCgemm`.

Una vez obtenidas todas las submatrices reducidas, tanto las que están en la diagonal como las demás, se ensambla la matriz completa. Si las submatrices reducidas se habían descargado a la CPU, se vuelven a enviar a la GPU, y allí se ensamblan con un *kernel*.

El proceso es mucho más sencillo que el que proporciona la matriz de acoplos del MoM, en parte porque es posible utilizar rutinas de álgebra lineal, y en parte porque el resto de operaciones son triviales: eliminar elementos de matrices o ensamblarlas. Encontrar el valor singular de corte en el vector de valores singulares S es una tarea compleja de realizar en la GPU, dado que tiene una naturaleza fundamentalmente secuencial, por lo que se ha preferido descargar el vector a la CPU y procesarlo ahí. El tiempo empleado en este paso es despreciable en comparación con el resto de pasos del proceso. No obstante, si fuera necesario, se podría diseñar un algoritmo que lo hiciera en la GPU, al estilo de las operaciones de reducción explicadas en el Capítulo 4.

El CBFM, tal cual lo hemos implementado en este trabajo, incluye la aplicación del MoM para obtener la matriz de acoplos de subdominios. Esta fase es la más exigente en cuanto a tiempo de proceso, tanto ejecutándolo en la CPU como si se ejecuta en la GPU, lo que justifica la gran inversión en tiempo y esfuerzo dedicados a ella.

6 Utilización de varias GPUs.

6.1 Introducción

En este capítulo se presentan los trabajos realizados al objeto de adaptar la aplicación para la utilización de varias GPUs. De entre las distintas opciones posibles se ha elegido la más sencilla de programar, teniendo en cuenta además el rendimiento que es posible obtener.

Cuando se dispone de varias GPUs y se pretende utilizarlas todas, es posible optar por varias líneas de trabajo: en primer lugar, se puede adaptar el código para repartir el trabajo entre ellas de forma "manual", es decir, seleccionando expresamente por programa para cada bloque de trabajo en qué GPU se va a ejecutar. Una segunda opción es utilizar una capa de software que cree varios procesos, para que cada uno de ellos gestione una GPU. Si se opta por esta última alternativa, es preciso elegir entre una comunicación entre los procesos llevada a cabo a través de mensajes, como se hace en el entorno MPI ('*Message Passing Interface*'), o por variables de memoria compartidas, que es la forma utilizada por OpenMP ('*Open Multi-Processing*').

El objetivo de este capítulo no es realizar un estudio de todas las opciones disponibles para la multiprogramación, sino simplemente adaptar nuestra aplicación para que pueda utilizar múltiples GPUs. Esta es la razón por la que se ha optado por la solución más cómoda, que es OpenMP. El siguiente apartado explica, si bien superficialmente, las características principales de OpenMP y por qué estas características hacen más sencilla la adaptación de nuestra aplicación. A continuación, en dos apartados adicionales, se describen las dos versiones adaptadas de la aplicación que se han creado.

6.2 OpenMP.

OpenMP [72] es un entorno de multiprogramación en sistemas con memoria compartida. Por medio de directivas de compilación, variables de entorno y algunas rutinas, permite crear programas con varios hilos de ejecución a partir de una versión secuencial. Los hilos creados pueden comunicarse entre sí por medio de variables compartidas, lo que hace el recargo de comunicación muy pequeño. Una de sus principales ventajas, y el factor decisivo para este trabajo, es su facilidad de uso: simplemente especificando qué partes del programa se han de ejecutar en paralelo, y qué datos se comparten y cuáles otros deben ser privados para los hilos de ejecución, se puede transformar una aplicación secuencial en una multi-hilo.

La forma de trabajar en OpenMP es tomar una aplicación secuencial y marcar, por medio de una serie de indicadores, qué regiones de la misma se pueden ejecutar en paralelo. OpenMP proporciona diversas opciones en cuanto a cómo se paralelizan las zonas marcadas, y también en cuanto a cómo se reparte el trabajo entre los diferentes hilos que se crean.

Entre las zonas de un programa, los bucles son los principales candidatos a la paralelización. Puesto que un bucle marca una zona de código que debe ejecutarse repetidas veces, es lo más natural plantearse la ejecución paralela de ese código. Esto es relativamente directo siempre que sus iteraciones sean independientes. En caso contrario no es imposible hacerlo, pero son precisas algunas transformaciones del código, tanto mayores cuanto más interdependientes sean las iteraciones.

Nuestra aplicación está estructurada en torno a dos bucles principales, como se describió en el capítulo 5: el bucle que procesa

las submatrices de la diagonal principal de la matriz de acoplos, y el que procesa el resto de submatrices. Ambos caen dentro de la primera categoría: las iteraciones son independientes entre sí. De hecho, la única dependencia en el código es la que impone el CBFM: para procesar las submatrices que están fuera de la diagonal, es preciso haber procesado antes las que sí están en la diagonal, ya que proporcionan las matrices de transformación entre representación de subdominios y representación de CBFs. Pero, salvo que esta dependencia provoca la división de la aplicación en los dos bucles mencionados, por lo demás las iteraciones de ambos bucles son independientes entre sí.

Las zonas de código que se han de ejecutar en paralelo serán procesadas por los distintos hilos que se creen. El programador tiene libertad para especificar cuántos hilos se crean para cada zona paralelizada. El trabajo se repartirá entre los hilos que se creen. Por ejemplo, en un bucle, no es necesario crear tantos hilos como iteraciones. Si se crean menos, las iteraciones se reparten de forma más o menos homogénea entre ellos. Si se crean más, simplemente habrá hilos que no hagan nada. Para asignar iteraciones a hilos existen diversas opciones, tanto estáticas (en el momento de la compilación) como dinámicas (en el tiempo de la ejecución).

Una vez definidas las zonas con ejecución en paralelo y el número de hilos de cada una, es preciso definir qué datos de la aplicación serán compartidos por los hilos, y qué datos serán privados para cada uno de ellos. Existen diversas categorías de datos privados, atendiendo fundamentalmente a si toman el valor inicial de la aplicación principal o si, por el contrario, este valor inicial debe ser asignado por cada hilo, y también con respecto a qué ocurre con sus

valores cuando termina la ejecución de los hilos. La categoría que se asigne a cada dato depende de la propia lógica de la aplicación y de el tipo de región paralela en la que se usa.

Por último, si es necesario, se pueden incorporar puntos de sincronización entre hilos. En el caso de nuestra aplicación, esto no es necesario, pero en ocasiones puede serlo si es preciso que unos hilos realicen alguna tarea antes que otros, que se produzca algún intercambio de datos, o que todos los hilos lleguen a determinado punto del programa antes de que cualquiera pueda continuar. La sincronización debe utilizarse con precaución ya que lleva asociada de forma inherente una pérdida de rendimiento.

En los subapartados siguientes se describen algunas de las características bosquejadas aquí con algo más de detalle. OpenMP está adaptado tanto a FORTRAN como a C, por lo que, para cada una de las características descritas a continuación existe una versión con sintaxis de C y otra con sintaxis de FORTRAN. Dado que en nuestra aplicación OpenMP se aplica a una función escrita en FORTRAN, es la sintaxis que se muestra, pero es necesario mencionar que, en cada caso, también existe la correspondiente para C.

6.2.1 Regiones paralelas.

La forma de trabajar con OpenMP es partiendo de una aplicación secuencial preexistente. No es necesario (aunque se puede hacer) diseñar la aplicación de forma paralela desde el principio. El programador analiza el código y decide qué partes del mismo son susceptibles de ser ejecutadas en paralelo. Para indicárselo al

compilador, la marca con las directivas `parallel` y `end parallel`.

Cuando el programa principal se encuentra una región etiquetada como paralela, se crea un grupo de hilos ('*a team of threads*', en terminología inglesa) para ejecutarla en paralelo. Las regiones paralelas se pueden anidar, de forma que, si un hilo de un grupo de hilos se encuentra otra región paralela, creará otro grupo de hilos para tratar esa nueva región paralela encontrada. El resto de hilos de su grupo de hilos también puede encontrarse la misma región paralela, creando cada uno el correspondiente nuevo grupo de hilos para procesarla. Implícitamente asociada al final de una región paralela se encuentra una sincronización llamada *de barrera* ('*barrier synchronization*'), de forma que ningún hilo pueda pasar de ella hasta que todos han llegado hasta ahí.

Al marcar una zona de código como región paralela es posible especificar características especiales sobre ella. Lo más habitual es que, justo a continuación del marcador de región paralela, se incluyan directivas que especifiquen cómo se reparte el trabajo entre los hilos y qué datos son paralelos y qué datos son privados. Es importante tener en cuenta que marcar una región como paralela solamente crea un grupo de hilos, pero, si no se especifica nada más, los hilos realizarán exactamente el mismo trabajo, por lo que no se consigue ninguna mejora. Es preciso indicar cómo se reparte el trabajo entre los hilos. Esta forma de repartir el trabajo puede tener un efecto considerable en la eficiencia de la paralelización. Las construcciones más importantes de reparto de trabajo son:

- *bucles* (directiva `do`): las iteraciones se ejecutan en paralelo, repartiéndose entre los hilos creados. Es la más utilizada, y admite diversos criterios de reparto de trabajo entre los hilos.
- *secciones* (directiva `section`): se utiliza cuando varias porciones de código, que en el caso más general, realizan tareas distintas, se pueden ejecutar en paralelo. Cada una de ellas es ejecutada por un hilo. Evidentemente, es necesario que no existan relaciones de precedencia entre los bloques.
- *ejecución única* (directiva `single`): dentro de una región paralela se especifica una porción de código que debe ejecutar un único hilo. No es posible especificar cuál de los creados; se usa únicamente para que no se ejecute la porción de código múltiples veces.
- *procesamiento de arrays* (directiva `workshare`): se utiliza únicamente en FORTRAN, y está relacionada con las operaciones de arrays de este lenguaje, sentencias que se aplican a todas (o algunas de) las componentes de un array. Si se utiliza, el trabajo que implica la sentencia se reparte entre los hilos creados.

La Tabla 6-1 resume los distintos tipos de región paralela. Cada una de ellas admite un subconjunto de directivas adicionales relativas a las características de los datos que utiliza (cuáles son públicos, cuáles privados, etc). Un ejemplo de región paralela se muestra en la Figura 6-1. La directiva en la línea 1 indica que comienza una región paralela en forma de bucle (directiva `do`). Esto hace que se creen tantos hilos como se haya especificado anteriormente, bien por variables de entorno, bien llamando a la función OpenMP correspondiente. El bucle propiamente dicho comienza en la línea 5,

que es por donde empiezan a ejecutar los hilos creados. Por omisión, los datos utilizados en la región son privados, como indica la línea 2, por lo que se crea una copia los que se usen. Será compartida la variable `ListaSubDominios`, como indica la línea 3. La línea 4 símplemente reitera que las variables `IndiceBloque`, `tid` y `nh` son privadas. El final de la región paralela lo indica la directiva de la línea 9.

En este trabajo, por la naturaleza de la aplicación que se desarrolla, la fórmula de reparto de trabajo más indicada es la relativa a bucles. Los dos bucles principales de la aplicación serán paralelizados y sus iteraciones repartidas entre los hilos creados. A cada hilo le corresponderá una GPU de forma fija durante la duración del bucle. Con esta construcción es posible gestionar las varias GPUs presentes en el sistema de forma sencilla y eficiente.

Tabla 6-1. Tipos de región paralela.

tipo de región	**descripción**
bucles	todos los hilos ejecutan el mismo código
secciones	hilos distintos ejecutan bloques distintos
ejecución única	sólo un hilo ejecuta la región
arrays	se reparte el trabajo entre los hilos (instrucciones de arrays en FORTRAN)

```
1  !$omp parallel do
2  !$omp default (private)
3  !$omp shared (ListaSubDominios)
4  !$omp private (IndiceBloque,tid,nh)
5  do IndiceBloque=1,NumeroBloques
6       tid=OMP_GET_THREAD_NUM()
7       nh=OMP_GET_NUM_THREADS()
        ...
        ...
8  end do
9  !$omp end parallel do
```

Figura 6-1. Ejemplo de región paralela.

6.2.2 Asignación de iteraciones a hilos.

En los bucles paralelizados, las iteraciones del bucle se reparten entre los hilos creados, y existen diversas formas de asignar iteraciones a hilos. En esta sección se explican las más utilizadas.

- *estática*: las iteraciones se agrupan en bloques homogéneos de iteraciones consecutivas, tantos como hilos, y se asigna un bloque a cada uno de forma secuencial. Puesto que el número de iteraciones no tiene por qué ser múltiplo del número de hilos, es posible que los últimos hilos tengan menos iteraciones que los primeros.
- *dinámica*: las iteraciones se agrupan en bloques, como en el tipo anterior, pero se asignan a hilos de forma dinámica, a medida que los hilos los piden. Por el mismo motivo que el tipo anterior, puede ocurrir que los últimos bloques tengan menos iteraciones que los primeros.

- *guiada*: se trata de una forma muy parecida a la anterior, pero en este caso el tamaño del bloque no es constante, sino que es proporcional al número de iteraciones que quedan por asignar. De esta forma, el trabajo que se reparte va siendo menor cada vez que se asigna un bloque, y el ajuste es más fino.
- *seleccionada en tiempo de ejecución*: permite, por medio de variables de entorno, fijar el método de asignación. El método ha de ser uno de los anteriores. La ventaja es que no es necesario tomar la decisión en el momento de compilar el programa, sino que se pueden tener en cuenta factores que surjan sobre la marcha.

Cada una de las fórmulas de reparto disponibles (resumidas en la Tabla 6-2) se puede configurar por medio de determinados parámetros. Por ejemplo, la dinámica y la guiada permiten especificar el número mínimo de iteraciones asignadas en cada bloque.

Tabla 6-2. Estrategias de planificación

planificación	**características**
estática	reparto fijo de bloques de tamaño fijo.
dinámica	reparto dinámico de bloques de tamaño fijo.
guiada	reparto dinámico de bloques de tamaño variable.
en ejecución	selección del método en tiempo de ejecución.

El objetivo de este trabajo, como se ha explicado más arriba, no es explorar el espacio de configuración de OpenMP, sino realizar una gestión sencilla de las distintas GPUs presentes en el sistema, por lo

que se utilizan las características necesarias para este objetivo. En este orden de cosas, en este trabajo se han probado dos de las técnicas: la estática y la guiada. Se ha prescindido de la dinámica porque cuando el número de iteraciones a repartir no es demasiado grande, como es el caso de nuestra aplicación, es muy parecida a la guiada, y la única diferencia es el ajuste dinámico del número de iteraciones en cada bloque que proporciona ésta última, cuestión interesante en nuestro caso. La asignación en tiempo de ejecución se ha desestimado porque la carga de trabajo de nuestra aplicación es relativamente predecible (en tipo y cantidad), por lo que no es necesario evaluar las condiciones de ejecución en el momento de realizar el reparto de carga.

6.2.3 Asignación de los datos.

Una vez se han determinado las regiones paralelas y se ha especificado cómo se ha de repartir el trabajo entre los distintos hilos, es preciso definir cómo se asignan los datos a los hilos. En otras palabras, qué datos van a ser compartidos por todos los hilos, y qué datos van a ser específicos (privados) de cada hilo. En general, los hilos utilizan variables definidas en el programa principal, y lo único que hay que hacer es especificar, en la cabecera de la región (a continuación del marcador de región paralela) qué datos se utilizan en la región y de qué tipo son. Esto no es una redeclaración, ni se crean datos nuevos. Simplemente se avisa al compilador de las características de los datos que usa la región.

Los datos compartidos (directiva `shared`) proporcionan un mecanismo de comunicación entre hilos, dado que las modificaciones que un hilo haga pueden ser observadas por todos los demás, pero

también exigen una sicronización en el acceso si varios hilos pueden modificarlos. Los datos compartidos que no son modificados representan los datos de entrada a la región paralela.

Los datos privados (directiva `private`) son aquellos que pueden ser modificados por los hilos, pero cuyas modificaciones no es necesario (o conveniente) que sean observados por los demás hilos. En general, cada hilo proporciona una parte de los resultados de la región paralela, y para obtenerlos utiliza algunas variables intermedias. Tanto unos como otras serán datos privados si no es necesario que los reciban los demás hilos. Cuando los datos privados son utilizados previamente fuera de la región paralela, se crea una copia local para cada hilo, que es sobre la que el hilo trabaja.

Además de estas dos categorías genéricas, datos compartidos y privados, existen otras que permiten precisar con algo más de detalle el uso que se va a dar a los datos. A continuación se describen algunas de los más importantes:

- `lastprivate`: se trata de datos privados que deben mantener su valor después de la ejecución de la región paralela. Dado que para los datos privados se crea una copia local, cuando el hilo termina los datos desaparecen. Para aquellos casos en los que esto no deba ser así, se utiliza esta categoría. El valor que se traslada fuera de la región paralela es el de la última escritura desde un punto de vista lógico del programa. Por ejemplo, en un bucle, el valor que escriba en el dato la última iteración (aunque en la ejecución paralela la última iteración puede no ser la última que se ejecute, el valor retenido por el dato es éste).

- `firstprivate`: es, en cierto sentido, lo contrario de la categoría anterior. En este caso es un dato privado que debe retener el valor que tenía fuera de la región paralela antes de que lo reciban los hilos. En otras palabras, son datos privados inicializados fuera de la región paralela.

Dado que puede ocurrir que la cantidad de datos que manejen los hilos sea muy grande, para evitar largas listas de etiquetado de datos, existe la directiva default, que permite asignar categorías por omisión. Las categorías que se pueden asignar con esta directiva son private, shared y none (ninguna). De esta manera, si la mayoría de los datos que usa una región paralela son compartidos, se les asigna esta categoría con la directiva default y sólamente es necesario especificar aquellos que son privados. La Tabla 6-3 resume las categorías de datos respecto a su compartición.

Tabla 6-3. Compartición de datos.

tipo	**descripción**
`shared`	compartido
`private`	privado sin inicializar
`firstprivate`	privado inicializado
`lastprivate`	privado, retiene el último valor asignado más alla del fin de la región paralela

6.2.4 Sincronización.

Todo sistema de memoria compartida, y OpenMP lo es, debe tener mecanismos de sincronización que permitan, si es preciso, ordenar los accesos a los datos compartidos. En el caso más sencillo,

se trata simplemente de asegurar que no se leen datos que aún no han sido creados. En los casos más complejos es posible establecer precedencias entre los hilos para que partes de la zona paralela se ejecuten en un orden determinado.

Las regiones paralelas llevan implícita una sincronización de barrera al final, de forma que ningún hilo pueda superar ese punto hasta que se produzca la llegada de todos los demás, y en nuestra aplicación no es necesaria ninguna más. Sin embargo, para dar una visión completa de OpenMP, se enumeran a continuación algunas de las construcciones que incluye.

- *Barrera* (directiva `barrier`): se trata de un punto del programa que todos los hilos deben alcanzar antes de que ninguno de ellos pueda proseguir. Para evitar que la aplicación se quede bloqueada, es necesario asegurarse al crear el programa de que todos los hilos llegan a todas las barreras a las que tienen que llegar, ya que es fácil cometer errores en este sentido que "cuelguen" el programa.
- *Ejecución ordenada* (directiva `ordered`): dentro de un bucle paralelizado se puede especificar la ejecución ordenada de un bloque de código dentro de cada iteración para asegurarse de que los hilos lo ejecutan por orden de iteración. Sólo se aplica al bloque ordenado, no al resto del código de la iteración.
- *Sección crítica* (directiva `critical`): proporciona la posibilidad de serializar la ejecución de una zona de código, de forma que sólo un único hilo se encuentre ejecutándola en cada instante de tiempo.
- *Asignación atómica* (directiva `atomic`): tiene un sentido parecido a la opción anterior, pero en este caso aplicada a una

única sentencia de asignación. Es una alternativa más eficiente que la sección crítica, aunque su simplicidad puede no ser válida para todas las situaciones.

- *Ejecución por hilo maestro* (directiva `master`): cada grupo de hilos que se crea para una región paralela tiene un hilo maestro (que es uno de ellos) al que se le pueden asignar tareas especiales por medio de la directiva `master`.

6.3 Adaptación del algoritmo a varias GPUs.

El objetivo de este capítulo es documentar cómo se ha utilizado OpenMP para gestionar las distintas GPUs que pueden estar presentes en el sistema. En apartados anteriores se han descrito brevemente algunas de las características esenciales de OpenMP, y en éste y el siguiente se explica cuáles se han utilizado y cómo. El objetivo, como se ha mencionado ya, no es hacer un estudio exhaustivo de cómo aprovechar OpenMP para sacar el máximo partido a los *cores* de la CPU, ya que esa línea de trabajo está completamente fuera del ámbito de esta tesis. La misión principal que se encomienda a OpenMP en este trabajo es facilitar la gestión de varias GPUs.

Dada la facilidad que ofrece OpenMP para la creación de hilos de CPU, se trata de una opción mejor que programarla "a mano" en la aplicación, ya que ésto último conllevaría mantener información de contabilidad respecto a qué iteración de qué bucle se ha asignado a qué GPU, y sería necesario continuamente seleccionar la tarjeta objetivo de las operaciones CUDA, tanto lanzar *kernels* como enviar y recibir datos. Con OpenMP se fija la GPU una vez al principio de cada hilo y no es necesario seleccionarla otra vez.

De todas las características que ofrece OpenMP, sólo algunas son necesarias para lo que se pretende hacer. Los dos bucles principales de la aplicación, el que procesa las submatrices en la diagonal de la matriz de impedancias, y el que procesa las demás, se van a paralelizar con la directiva `do`, de forma que se repartan las iteraciones entre diferentes hilos. Las dos versiones que se han desarrollado difieren en cuanto a cómo se realiza ese reparto. A continuación se describen ambas versiones.

6.3.1 Versión 1 para varias GPUs.

Con respecto al reparto de trabajo, en la Versión 1 se van a probar dos opciones: crear un hilo por cada GPU, y crear dos hilos por cada GPU. La primera opción tiene por objetivo simplemente asignar el trabajo a las GPUs con hilos independientes. La segunda busca aprovechar al máximo cada GPU, de forma que esté siempre trabajando: cuando termine el trabajo de un hilo puede recibir el que le asigne el segundo, y vuelta a empezar. La asignación de iteraciones a hilos será la estática, y más adelante, en la Versión 2, se probará la guiada.

Con respecto a la asignación de los datos, se procura que sean compartidos la mayor cantidad de ellos posible. De esta forma se evita duplicarlos innecesariamente, con lo que se ahorra en memoria. Sólo algunos deben ser privados, ya que los hilos deben generarlos o modificarlos, o bien, como es el caso más general, son datos diferentes para cada hilo, aunque no sean modificados por éste. Por seguridad, y para evitar los llamados "efectos laterales" de la programación (accesos a variables globales no intencionados), se declaran todos privados, y se definen específicamente algunos como compartidos

(todos los que se pueda), ya que pueden ser compartidos sin peligro. Por ejemplo, son compartidos los datos de la geometría, la información sobre los subdominios (posiciones, derivadas, etc), la lista de subdominios en cada región y en cada región extendida, las excitaciones (para la parte del CBFM), etc.

Por otra parte, son privados datos como la lista de subdominios que deben procesarse en cada iteración del bucle, ya que obviamente ésta es diferente en cada una.

La aplicación debe incorporar algunas pequeñas modificaciones adicionales para tratar varias GPUs. Por ejemplo, es preciso que los datos de la geometría se cargen en todas las GPUs que vayan a participar en la ejecución. Esto no es demasiado costoso en términos de tiempo, así que no supone un recargo adicional de consideración.

Una vez preparada cada GPU, la aplicación sigue su curso normal, y cuando llega al bucle que procesa las submatrices en la diagonal, reparte el trabajo entre los hilos creados (que pueden ser en número uno por cada GPU o dos por cada GPU, según la variante de la Versión 1 que se use). Éstos generan la información pertinente a la submatriz que procesan y seleccionan la GPU que les corresponde. Las GPUs se asignan a hilos de forma consecutiva: el hilo 0 tiene la GPU 0, el hilo 1 la GPU 1 y así sucesivamente. Cuando se terminan las GPUs, se vuelve a empezar por la 0 (para los casos en los que haya más hilos que GPUs). La operación de selección de GPU está incluida entre las funciones de CUDA, y la asignación se mantiene mientras el hilo no seleccione otra. Una vez realizada la operación de selección, todas las operaciones de GPU que lance el hilo se ejecutan sobre la GPU seleccionada.

Desde ese momento, los hilos se dedican a lanzar los *kernels* que calculan la matriz de impedancias que les corresponde, siguiendo idéntica secuencia a lo ya descrito en los Capítulos 4 y 5. A medida que los hilos terminan con una iteración del bucle empiezan con la siguiente que tengan asignada, si la hay. Cuando todos terminan, la aplicación sigue avanzando y acomete el segundo bucle principal, el que procesa las submatrices fuera de la diagonal, operando de la misma manera. El ensamblado final de la matriz reducida se lleva a cabo por el hilo principal del programa.

6.3.2 Versión 2 para varias GPUs.

La Versión 2 de OpenMP funciona de manera muy parecida a la Versión 1, con la única diferencia de que la asignación de iteraciones a hilos se hace de manera dinámica. En concreto, se han probado las dos políticas más comunes: guiada y dinámica. La primera admite un parámetro, k, que indica el número mínimo de iteraciones asignadas a cada hilo. Si éste es 1, no hay límite inferior en este aspecto.

En la política guiada, se ha utilizado únicamente un k de 1. La razón de esto es que para un número pequeño de iteraciones, como es el caso del bucle principal de la aplicación, limitar por abajo el número de las mismas asignadas puede dar lugar a ineficiencias. Cuando se llega al final de la ejecución del bucle, cosa que ocurre relativamente pronto, se pueden asignar tantas iteraciones al mismo hilo que el resto no tenga trabajo que hacer. Utilizar un k mayor es más interesante en bucles con más iteraciones que los de esta aplicación, de forma que el recargo de asignar iteraciones a hilos se pague menos veces.

La política dinámica no ha producido un resultado digno de mención, ya que no es muy diferente a la política estática utilizada en la Versión 1. La razón es parecida a lo expuesto en el párrafo anterior: las iteraciones son pocas relativamente, por lo que no hay gran diferencia entre asignarlas de forma estática y de forma dinámica.

Del resto de características que ofrece OpenMP no se ha hecho uso de ninguna. Especialmente de las relativas a la sincronicación: las incluidas por omisión en la paralelización de bucles son suficientes para esta aplicación. Dado el tipo de paralelismo presente en ella, con total independencia entre iteraciones del mismo bucle, no es necesario coordinar la ejecución de los hilos. La única dependencia que hay que respetar es que se ejecuten todas las iteraciones del primer bucle antes de avanzar al segundo, ya que éste necesita de las matrices de transferencia que se obtienen en el primero. De hecho, incluso sería posible plantearse un esquema en el que, a medida que se van obteniendo estas matrices, se permita poner en ejecución los hilos del segundo bucle que las utilizan, cosa que podría ocurrir antes de que se terminaran de calcular todas ellas. Sin embargo, el trabajo necesario para implementarlo no obtendría demasiada recompensa, dados los resultados obtenidos con las versiones presentadas, especialmente la Versión 2, por lo que se ha optado por no llevarlo a cabo.

6.4 Justificación de la elección de OpenMP.

En este capítulo se han descrito las características principales de OpenMP de una forma un tanto superficial. OpenMP ofrece una gran variedad de opciones y posibilidades a la hora de crear una aplicación

paralela de memoria compartida. Además, se han mencionado otras opciones disponibles al respecto de la tarea a realizar, que era la utilización de varias GPUs para acelerar nuestra aplicación, como programar de forma manual la asignación de trabajo a las mismas o la utilización de otro entorno paralelo como MPI. En este apartado se intenta justificar la elección de OpenMP para esta tarea.

En primer lugar, es preciso recalcar la sencillez de uso de OpenMP. Se trata de una herramienta que únicamente requiere que el compilador respete sus directivas (como era el caso del utilizado en esta tesis). Una vez garantizado eso, el procedimiento es realmente simple: se analizan las zonas que presentan paralelismo aprovechable, se decide cómo se va a repartir el trabajo entre los hilos creados, y qué datos deben ser privados de cada hilo y cuáles deben ser compartidos, etiquetando cada uno de forma correspondiente. En nuestro caso ni siquiera ha sido necesario incluir sincronizaciones entre los hilos, dado el paralelismo entre ellos. Además, con OpenMP ha sido posible repartir el trabajo entre los hilos de forma dinámica para aprovechar mejor las GPUs.

Un segundo factor a tener en cuenta es la facilidad y la robustez con la que se cuenta para, desde un hilo determinado, manejar una GPU. Basta con seleccionarla al principio de la ejecución del hilo para que, desde ese momento, todo el trabajo (todas las operaciones en general, incluyendo lecturas y escrituras de memoria de GPU) que lanza el hilo se dirija a la GPU seleccionada. Este mecanismo hace que sea tremendamente sencillo llevar la gestión de varias GPUs.

Una ventaja más que ofrece OpenMP es la sencillez en cuanto a la repartición de datos: no es preciso crear múltiples copias de los datos

privados, ni repartir los compartidos. Todo lo gestiona el compilador de forma invisible para el programador.

Si se compara OpenMP con las otras dos opciones disponibles, el resultado es que OpenMP es más apropiado para la tarea que se plantea en este capítulo que MPI porque libera al programador en gran medida de la gestión de los datos en cada hilo: simplemente se etiqueta cada uno como privado o público y OpenMP se encarga. No es necesario crear copias de los privados ni distribuir los públicos, como en MPI. Además, para una máquina como la que se utiliza, la memoria compartida es mucho más rápida que el envío de mensajes, no sólo más cómoda de programar.

Por otro lado, OpenMP facilita considerablemente la labor de la gestión de varias GPUs con respecto a hacerlo de forma manual. Hacerlo de esta última manera habría supuesto tener que añadir código para gestionar qué se envía a cada GPU, y arbitrar en qué momento se hace, ya que mientras la GPU trabaja la CPU generalmente espera. Eso es sencillo con una única GPU, pero con varias, para no incurrir en pérdidas de tiempo sería necesario algún tipo de espera con sondeo permanente para ver cuál termina, y a continuación enviarle más trabajo. Todo esto desaparece al crear varios hilos, cada uno pendiente de una GPU.

Como resumen, OpenMP proporciona todo lo que se necesita con un coste en trabajo y en tiempo de ejecución menor que cualquiera de las alternativas, por lo que es la opción clara para una aplicación con una estructura tan sencilla como la nuestra. Esta estructura ha hecho que no haya sido necesario utilizar a fondo las características de programación paralela de OpenMP, pero esto es una ventaja

adicional de este entorno: no hace falta un gran dominio del mismo para empezar a obtener grandes resultados.

7 Resultados.

7.1 Introducción

En este capítulo se presentan los resultados de los programas descritos en los apartados anteriores. Dado que el objetivo principal de todo el trabajo es conseguir una aceleración en los tiempos de ejecución que justifique el enorme esfuerzo de convertir completamente aplicaciones ya existentes a la nueva plataforma de programación, el capítulo se centra fundamentalmente en analizar las mejoras de rendimiento de las diferentes versiones. Sin embargo, antes de considerar las mejoras obtenidas, es preciso comprobar que los resultados son correctos. Para ello se utilizan dos criterios: por un lado, se comparan las corrientes inducidas por una onda incidente en la geometría que obtienen tanto la versión original de la aplicación como la versión de CUDA que se esté validando. Por otro lado, se comparan los resultados obtenidos al calcular la sección rádar con el programa original y con las versiones aceleradas. Como se verá en las gráficas, con ambos criterios los resultados son prácticamente idénticos a los de la versión original, por lo que se puede aceptar la precisión de las versiones de la GPU.

Una vez validadas las versiones de GPU, se puede pasar a evaluar su rendimiento. En primer lugar se presentan los resultados para la obtención de la matriz del Método de los Momentos (MoM). Se utilizan para esto dos geometrías elementales, un diedro y una esfera, y se mide la mejora que las sucesivas versiones ofrecen en el cálculo de la matriz.

A continuación, se presenta un estudio parecido, pero para la matriz reducida del método de las Funciones Base Características

(CBFM). En una primera fase, tratando la geometría en un único bloque (es decir, aplicado a una geometría pequeña), y después, aplicando el método a una geometría dividida en varios bloques (lo que permite que sea mayor). Para el CBFM, las geometrías que se utilizan fundamentalmente son más realistas: un avión y una cavidad cobra.

Por último, se presentan resultados de las mejoras obtenidas cuando se usan varias GPUs. Por limitaciones en el equipo del que disponemos para la realización de este trabajo, el máximo número de GPUs utilizadas ha sido 3. Sin embargo, estos resultados son suficientemente interesantes como para mostrarlos, ya que abren una nueva vía de mejoras para las aplicaciones desarrolladas.

Para todas estas medidas se utiliza fundamentalmente una máquina que denominaremos 'Máquina 1'. Sus características son: procesador Intel i3, 8 GB de memoria RAM, y una GPU TESLA C2075 con 6 GB de memoria RAM. La máquina tiene un sistema operativo Windows 7 de 64 bits. En cuanto a las bibliotecas de código, se utiliza el CUDA Toolkit versión 3.1, que incluye una versión de las biliotecas cuBLAS, y la biblioteca de CULA 2.1. Para la versión secuencial de la aplicación, se utiliza la biblioteca Intel Math Kernel Library.

En algunas ocasiones (especialmente para ejecutar las versiones aceleradas que utilizan varias GPUs) se utiliza otra máquina, que denominaremos 'Máquina 2'. Ésta dispone de un procesador Intel i3, 8 GB de memoria RAM, y hasta 3 tarjetas GPU TESLA C1060 con 4 GB de memoria RAM. El software es el mismo que en la Máquina 1. En los experimentos en los que se utiliza la Máquina 2, esto se indica expresamente. Para el resto, se ha utilizado la Máquina 1.

7.2 Geometrías.

Para la evaluación, tanto de la precisión de los programas que corren en la GPU como de la mejora en el rendimiento de los mismos, se han utilizado una serie de geometrías que se describen a continuación. En primer lugar, para las versiones iniciales que únicamente aplican el MoM, se utilizan dos geometrías sencillas: un diedro (Figura 7-1) y una esfera (Figura 7-2).

En el caso del diedro, se trata de dos placas cuadradas de cuatro metros de lado que están dispuestas en planos perpendiculares. La frecuencia utilizada con esta geometría es de 1.5e8 s^{-1}, por lo que el tamaño eléctrico de la misma es de 2 lambdas. Si el factor de discretización es de 10 divisiones por lambda, el número de subdominios está alrededor de 1500.

En el caso de la esfera, tiene un radio de 1 metro, y se utiliza igualmente una frecuencia de 1.5e8 s^{-1}. De esta manera, la geometría tiene un tamaño eléctrico de 0.5 lambdas. Con un factor de discretización de 10 divisiones por lambda, tiene del orden de 1500 subdominios.

Estas dos geometrías son sencillas pero razonablemente generales, y se utilizan dos tamaños eléctricos diferentes para tratar con una variedad de rangos algo mayor.

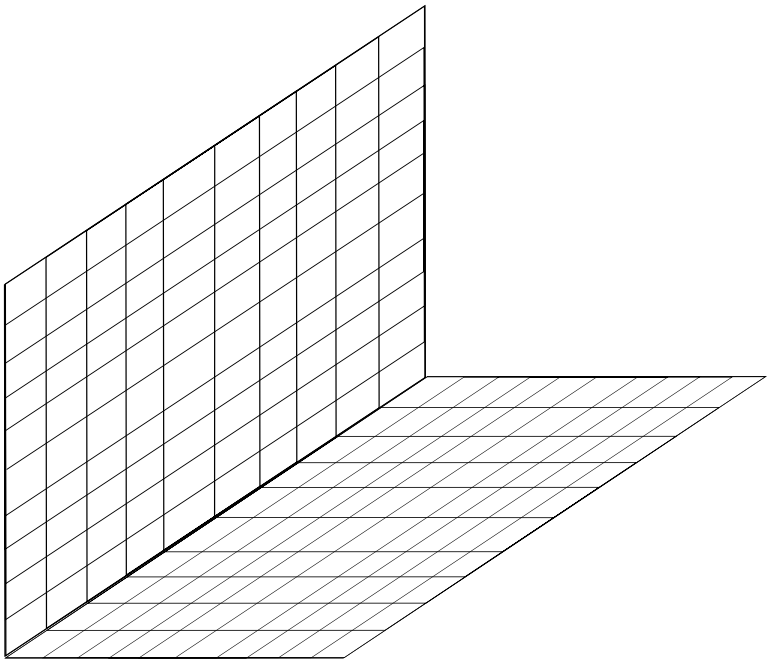


Figura 7-1. Diedro

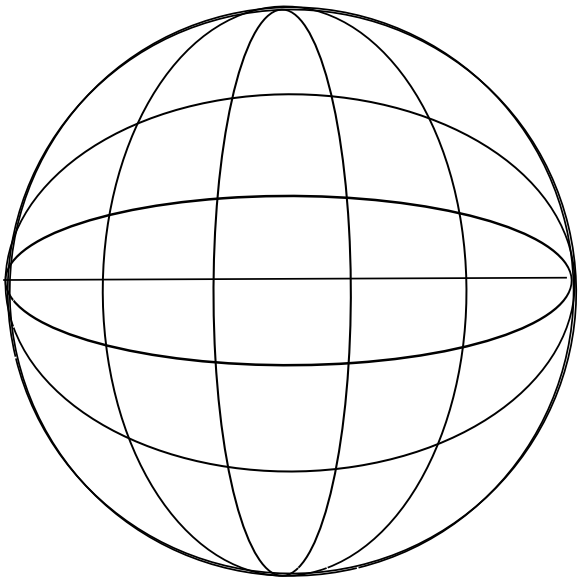


Figura 7-2. Esfera

Para hacer estudios más realistas se ha optado por dos geometrías más complejas: un avión (Figura 7-3) y una cavidad Cobra (Figura 7-4). En este caso, como muestran las figuras, las geometrías tienen muchos más planos. En el caso del avión, le damos dos utilidades. Por un lado, para procesarla como un único bloque cuando se necesita una geometría grande. Para ello, utilizamos una frecuencia de 21e8 s^{-1}, con lo que el tamaño eléctrico es de unas 4 lambdas, y el número de subdominios es de unos 3500 si se usa un factor de discretización de $\lambda/10$. El segundo uso que le damos es cuando queremos procesarla en varios bloques. En este caso se puede utilizar una frecuencia mayor: 28e8 Hz, y se pueden obtener unos 7000 subdominios.

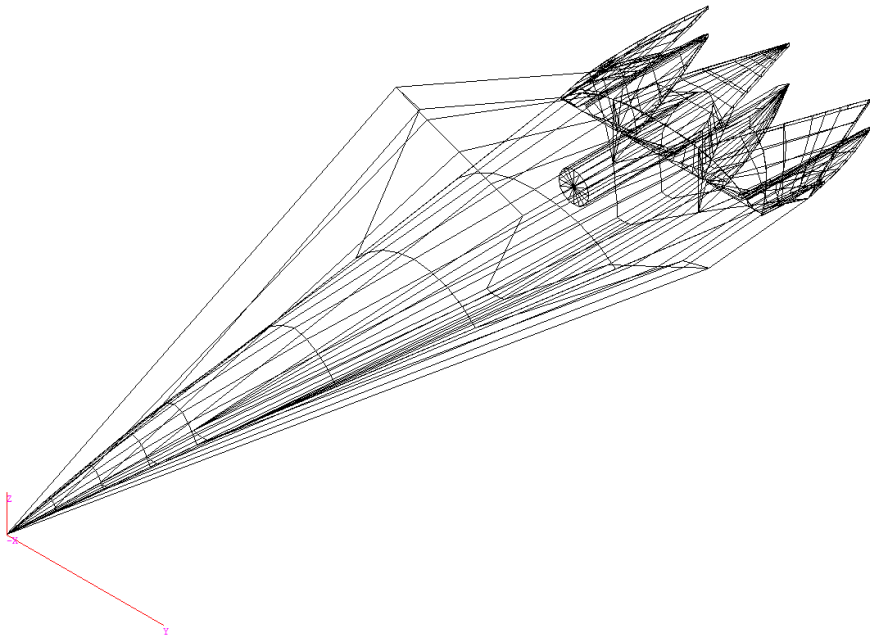


Figura 7-3. Avión

La cavidad cobra la empleamos únicamente para procesarla en varios bloques. Debido a esto, con ella utilizamos una frecuencia de 40e8 Hz, y con un factor de discretización de $\lambda/10$ se obtienen unos 7000 subdominios. El tamaño eléctrico es de unas 6 lambdas.

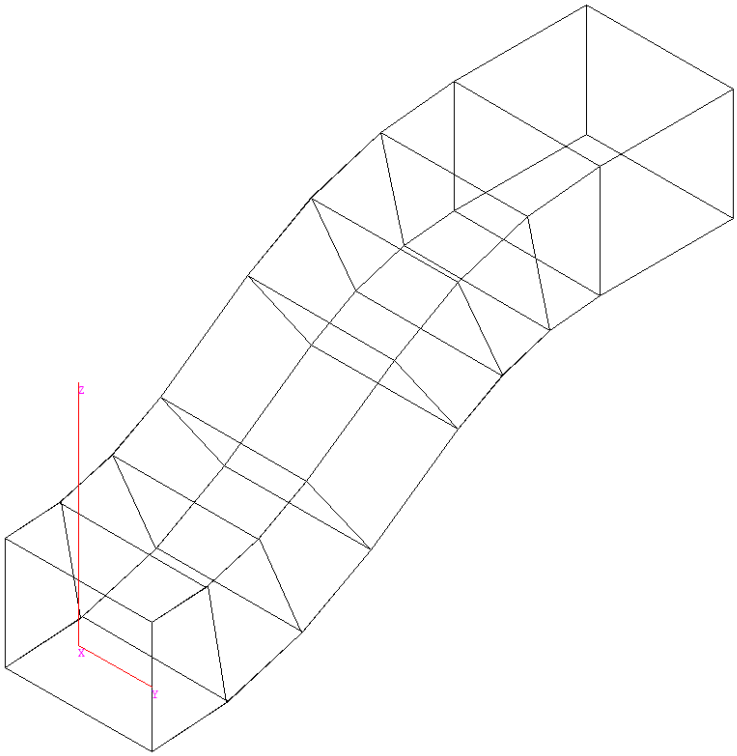


Figura 7-4. Cavidad Cobra

7.3 Validación de los resultados.

En esta sección se estudia la precisión de los resultados producidos por las versiones de CUDA. En primer lugar, para las geometrías sencillas (el diedro y la esfera) se miden las corrientes inducidas en la geometría por una onda incidente plana. Las corrientes se calculan utilizando la matriz de impedancias obtenida de la aplicación del método de los momentos. De esta manera, si las corrientes inducidas proporcionadas por las versiones de CUDA coinciden con las que

calcula la aplicación original, se puede concluir que las versiones de CUDA son correctas.

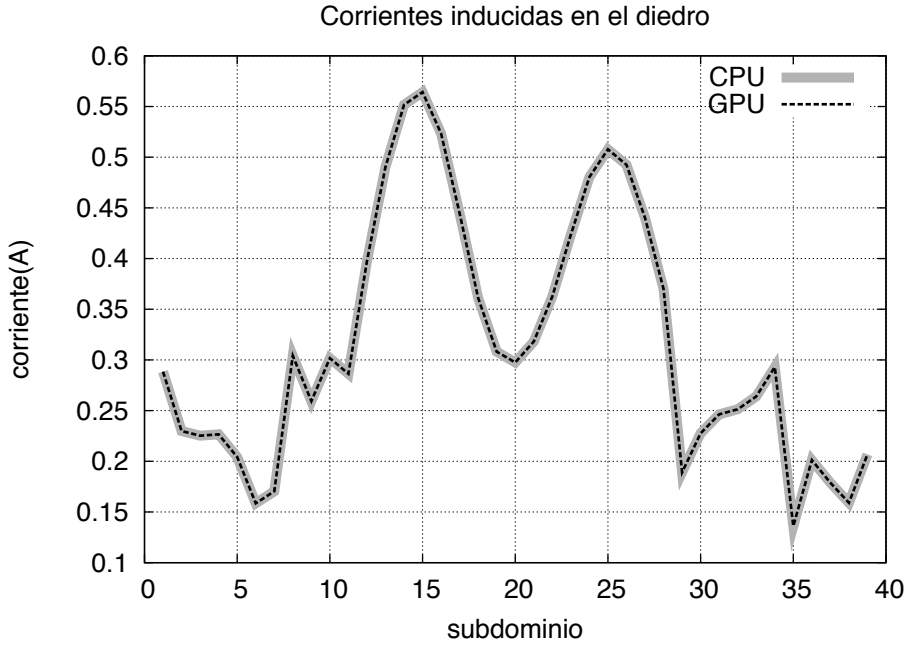


Figura 7-5. Corrientes inducidas en el diedro

La Figura 7-5 muestra un ejemplo de los resultados obtenidos. Se trata de las corrientes calculadas en el diedro para una onda plana incidente en el mismo para $\theta = 70$ grados y $\varphi = 40$ grados. La gráfica muestra las intensidades, medidas en amperios, obtenidas en los subdominios situados en un corte transversal del diedro. El trazo grueso continuo representa los datos de la aplicación original (etiquetada 'CPU') y el trazo fino discontinuo los datos de la versión de CUDA (etiquetada 'GPU').

La gráfica de la Figura 7-5 se obtuvo utilizando la versión base del capítulo 4 para el MoM, pero da resultados similares con todas las optimizaciones. Esto es razonable, ya que la parte de cálculo es esencialmente idéntica en las cuatro, y las optimizaciones sólo afectan a cómo se organizan los cálculos y no a cuáles son éstos.

Como se puede apreciar en la figura, los resultados son prácticamente iguales. Esto es así para todas las direcciones de incidencia de la onda plana y para todos los subdominios de la geometría. Se muestra únicamente una gráfica a modo de ejemplo de todos los casos.

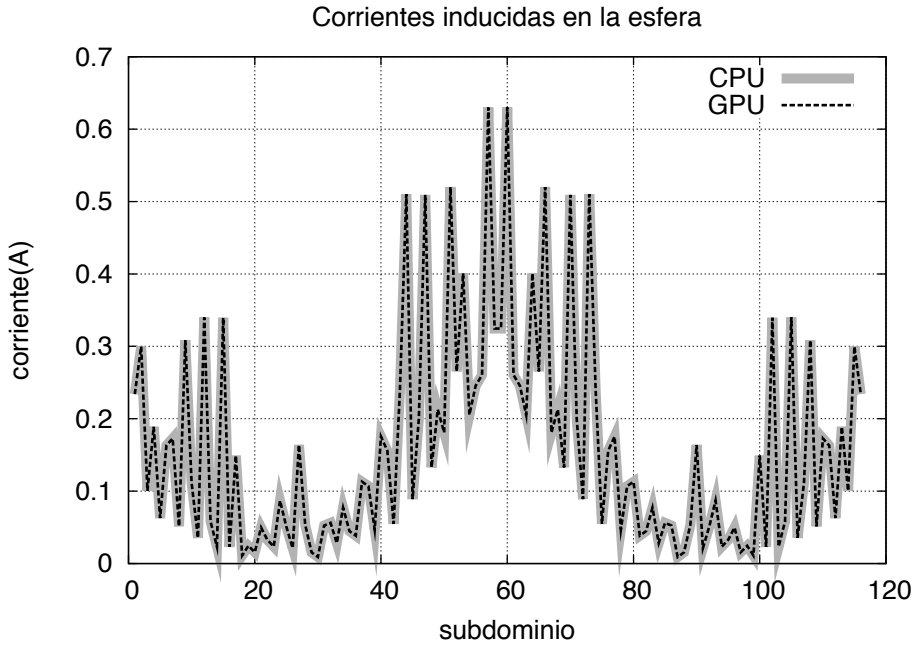


Figura 7-6. Corrientes inducidas en la esfera

La Figura 7-6 muestra los resultados obtenidos de un experimento similar utilizando como geometría la esfera. La versión de la aplicación utilizada es la misma que para la Figura 7-5. De nuevo los resultados son idénticos para la versión original y la acelerada.

Para medir la fiabilidad de los resultados suele ser más habitual utilizar la sección rádar (RCS). Para las geometrías más complejas se ha utilizado la RCS como criterio de validación. A continuación se muestran los resultados que se han obtenido para ambas geometrías. Las ondas incidentes son un conjunto de ondas polarizadas en θ, cada una con una dirección de incidencia. Las direcciones están espaciadas

regularmente, de forma similar a lo que ocurría con el espectro de ondas planas (PWS).

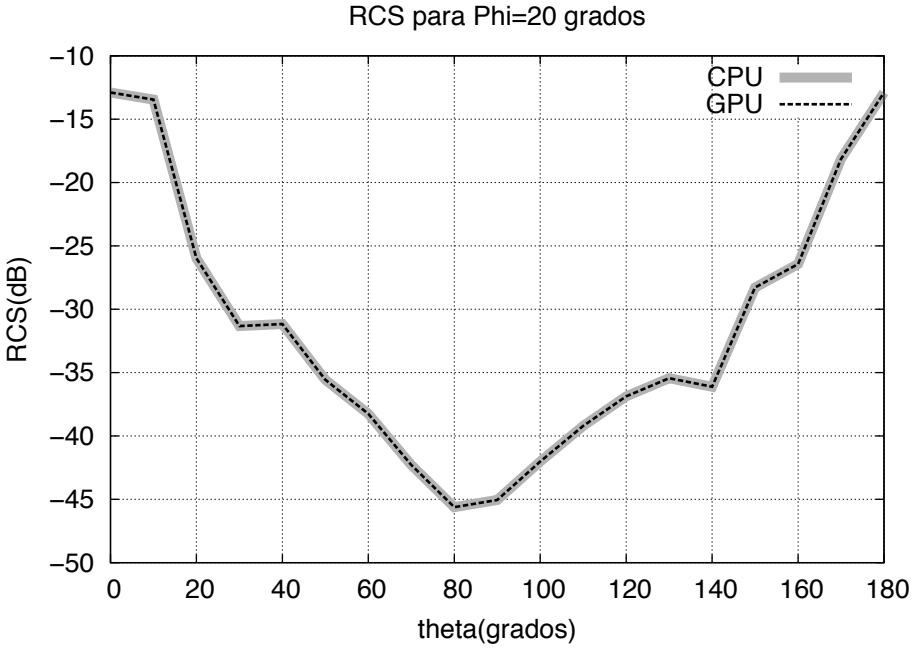


Figura 7-7. RCS para el avión

La Figura 7-7 muestra el resultado en el caso del avión para una de las ondas incidentes, en concreto, los valores medidos en un plano con $\varphi = 20$ a intervalos de 10 grados, es decir, para todos los valores de θ (desde 0 a 180). Igual que en las gráficas anteriores de esta subsección, se muestran en trazo grueso continuo los resultados calculados con la aplicación original, y en trazo fino discontinuo los equivalentes en la versión acelerada. En este caso, la versión acelerada primero obtiene la matriz reducida del CBFM, y la RCS se obtiene a partir de ella.

Como se puede observar, los resultados son, de nuevo, prácticamente idénticos. Ocurre lo mismo con la cavidad Cobra, cuyos resultados se muestran en la Figura 7-8. La configuración del

experimento es la misma que para el avión, y la gráfica muestra los datos obtenidos en las mismas condiciones.

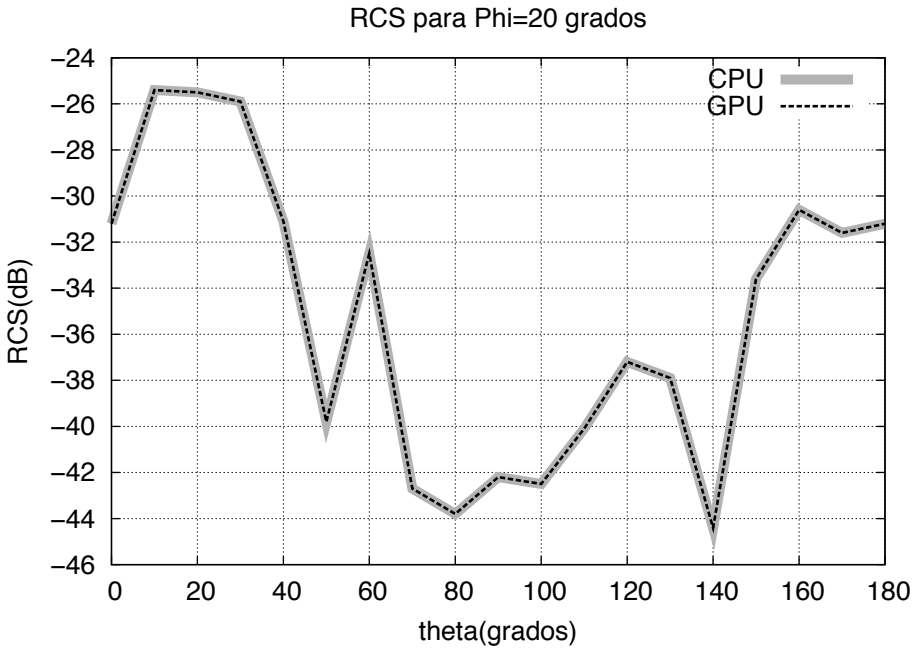


Figura 7-8. RCS para la cavidad Cobra

Las gráficas mostradas son únicamente ejemplos de todos los datos obtenidos. Son un subconjunto muy pequeño de ellos, pero son representativos. Tanto si se calculan las intensidades inducidas en la geometría, como si se calcula la RCS, las versiones desarrolladas como parte de este trabajo proporcionan resultados prácticamente idénticos a los que se obtienen con la aplicación secuencial de partida. Los resultados han de ser los mismos si los cálculos también lo son. Los datos obtenidos validan la corrección de las versiones desarrolladas.

7.4 Aceleración de la obtención de la matriz del Método de los Momentos.

Una vez validadas las versiones de CUDA en cuanto a la precisión de los resultados obtenidos, en esta sección se presentan las mediciones de tiempos realizadas para comparar su rendimiento. A lo largo del capítulo se utilizan de forma indistinta los términos *ganancia* y *aceleración*. En ambos casos representan la proporción en la que una versión con una mejora es más rápida que otra versión sin esa mejora. Es decir:

$$ganancia = \frac{t_{base}}{t_{mejora}}.$$

Dado que la mejora suele reducir el tiempo de ejecución, la ganancia o aceleración suele ser mayor que 1. De esta manera, una ganancia o aceleración de 2 supone que el tiempo de ejecución con la mejora es la mitad que el tiempo original. Una ganancia de 2 también se describe en ocasiones como una ganancia del 200%.

En las gráficas que se muestran se presentan las aceleraciones obtenidas por las sucesivas técnicas y características de CUDA estudiadas en el capítulo 4 para la aplicación que obtiene la matriz de impedancias del MoM. Los tiempos medidos son únicamente los que conlleva el cálculo de dicha matriz, y se excluyen los análisis de la geometría realizados previamente al cálculo propiamente dicho. Es decir, básicamente se mide el tiempo de ejecución de la parte de la aplicación que utiliza la tarjeta gráfica. Los cálculos previos tienen un peso pequeño con respecto al total de la aplicación. Sí se incluyen los tiempos para convertir los datos del formato de FORTRAN al de C, y lo que se tarda en transferir los datos de la geometría a la tarjeta gráfica.

Las geometrías que se utilizan son el diedro y la esfera, descritas en el apartado 7.2. Para variar en cierta medida la cantidad de subdominios que maneja el programa se utilizan varios factores de discretización. Esto permite analizar el rendimiento de la apliación en un rango mayor de casos, lo que a su vez permite extraer conclusiones interesantes.

En concreto, en el diedro se utiliza como factor de discretización $\lambda/10$, y a partir de este valor se van utilizando geometrías con menos subdominios, reduciendo ese factor hasta $\lambda/6$. Geometrías menores que esas ya dan un número de subdominios demasiado reducido, por lo que no se usan. En el caso de la esfera, el factor de discretización va desde $\lambda/15$ a $\lambda/10$. Con este procedimiento se consigue fácilmente poder utilizar un rango interesante del número de subdominios sin necesidad de tener que generar una geometría distinta para cada caso. Además, se busca encontrar el tamaño del problema que satura la tarjeta, es decir, la geometría mayor que se puede tratar en un único bloque. Esto ocurre con el factor de discretización mayor, $\lambda/10$ para el diedro y $\lambda/15$ para la esfera. La frecuencia utilizada es la misma en todos los casos, 1.5e8 Hz.

Tanto la versión original de la aplicación como las diferentes versiones de CUDA se compilan con todas las opciones de optimización que admite el compilador, Visual Studio Professional versión 9. En el caso de las versiones de CUDA, también se activan las opciones de optimización para el código de la GPU que permite el compilador específico para éste. Tanto en la aplicación original como en las versiones de CUDA las medidas generales de tiempo se hacen utilizando la función `system_clock`, que en la plataforma en la que se desarrollan las pruebas proporciona una precisión de diezmilésimas

de segundo. En el caso de las versiones de CUDA, las medidas se toman desde la parte de la aplicación escrita en FORTRAN. Sin embargo, para otras medidas de tiempo más específicas de estas versiones, por ejemplo, medidas del tiempo de ejecución de algunos *kernels* concretos, o de fases de proceso de datos en la CPU, se utiliza la función `clock()` de C, que permite medir la duración de un intervalo en '*ticks*' del sistema. Estas tomas de tiempo están insertadas en la parte de la aplicación que controla el lanzamiento de los *kernels*, y está escrita en C. Tanto la función `system_clock` como la función `clock()` tienen tiempos de ejecución despreciables, por lo que no alteran la medición del tiempo de ejecución de las aplicaciones. Todas las medidas de esta sección se han realizado sobre la Máquina 1.

7.4.1 Versión inicial.

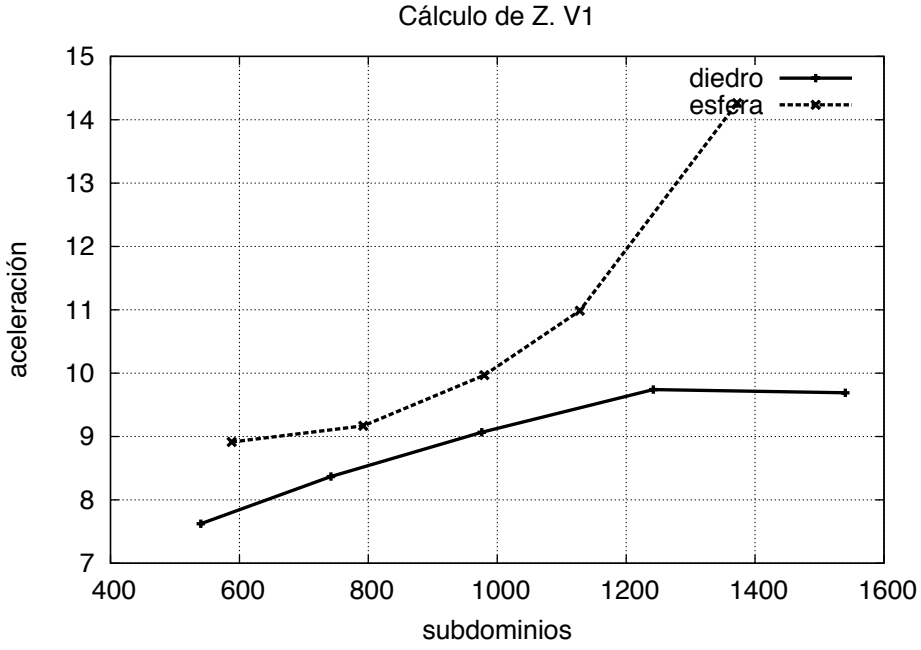


Figura 7-9. Aceleración obtenida con la Version inicial.

La Figura 7-9 muestra la aceleración obtenida con la primera versión de CUDA, medida como *el tiempo de la aplicación original dividido entre el tiempo de la aplicación acelerada*. En otras palabras, una aceleración de 2 (o del 200%) significa que la aplicación acelerada se ejecuta en la mitad de tiempo que la original. La línea continua corresponde al diedro y la discontinua a la esfera. Como se puede observar, la aceleración crece con el tamaño del problema. Este comportamiento se va a mantener a lo largo de toda la sección. Cuanto mayor es la geometría, mayor es el trabajo (el cálculo de la matriz de impedancias tiene complejidad del orden de n^2, con n la dimensión del problema), por lo que la GPU tiene más donde ganar. Para las versiones más pequeñas del problema (entre 500 y 600 subdominios), la ganancia es relativamente pequeña (aunque con otras tecnologías como MPI u OpenMP una ganancia de 7 no es nada desdeñable, para CUDA es muy reducida).

La línea que corresponde al diedro se estabiliza a partir de los 1200 subdominios, mientras que la de la esfera se mantiene creciendo hasta el final de la gráfica. Desgraciadamente, la memoria de la tarjeta no permite procesar geometrías mayores que las utilizadas en la gráfica, por lo que no es posible saber en qué punto se estabiliza la gráfica de la esfera (si es que lo hace), o si la del diedro vuelve a crecer para una tamaño mayor. En cualquier caso, los resultados son discretos para la tecnología CUDA, en la que aceleraciones de 100 no son extrañas.

Tabla 7-1. Tiempos para la Versión inicial.

Diedro					
Subd.	1540	1242	976	742	540
Ganancia	9.69	9.74	9.07	8.37	7.62
Total (s)	3.15	2.09	1.37	0.81	0.48
Transf. (s)	0.09	0.11	0.09	0.08	0.09
Parámetros (s)	1.28	0.83	0.55	0.31	0.53
Kernels (s)	1.62	1.05	0.64	0.36	0.19
Liberar (s)	0.16	0.11	0.09	0.06	0.05
Esfera					
Subd.	1372	1128	980	792	588
Ganancia	14.26	10.99	9.97	9.17	8.91
Total (s)	2.64	2.25	1.83	1.05	0.58
Transf. (s)	0.11	0.11	0.10	0.09	0.09
Parámetros (s)	1.05	1.05	0.97	0.40	0.20
Kernels (s)	1.34	0.98	0.67	0.49	0.25
Liberar (s)	0.14	0.11	0.09	0.00	0.05

La Tabla 7-1 muestra los tiempos medidos para la versión inicial en ambas geometrías. La primera línea de cada una de ellas indica el número de subdominios de cada ejecución. A continuación se muestran la ganancia obtenida, el tiempo total de ejecución, el tiempo empleado en transferir a la GPU los datos de la geometría (transf.), el tiempo empleado en el cálculo de datos específicos de los *kernels* (parámetros), el tiempo de ejecución de los propios *kernels* (*kernels*), y el tiempo necesario para liberar la memoria dinámica requerida tanto en la CPU como en la GPU (liberar).

La tabla permite extraer conclusiones interesantes para posteriores mejoras. Por ejemplo, el tiempo en transferir a la GPU los datos de la geometría se mantiene prácticamente constante de manera independiente del tamaño de la misma. Esto es una de las razones

por las que la ganancia aumenta con el tamaño del problema. Por otra parte, calcular los datos específicos que van a procesar los *kernels* ocupa un tiempo equiparable al de la ejecución de los propios *kernels*. De hecho, para tamaños de geometría menores pueden llegar a ser incluso mayores.

Los datos medidos permiten apuntar la dirección de posibles mejoras. Por un lado, la ejecución de los *kernels* lleva prácticamente tanto tiempo como todo el resto de tareas. Esto quiere decir que hay que mejorar el diseño de los *kernels*, pero también hay que buscar optimizar la parte de la aplicación que no se ejecuta en la GPU si se pretende una mejora equilibrada del tiempo de ejecución.

El cálculo de los parámetros que utilizan los *kernels* incluye operaciones como obtener las posiciones de los subdominios en la geometría y el número de puntos de integración de las integrales de superficie. Esta información se puede extraer de los datos referentes a la geometría que ya se han introducido en la GPU, por lo que es posible generarlos por medio de uno o varios *kernels* específicos, como se explica en el capítulo 4.

Con respecto al tiempo de ejecución de los *kernels*, el trabajo que tienen es relativamente fijo, es decir, no se puede eliminar. Lo único que queda es organizarlo de una manera más eficiente. Como se explicó en el Capítulo 4, la primera versión para CUDA consta de un único *kernel* que calcula todas las integrales de superficie, independientemente del número de puntos. Esto crea un desequilibrio entre los bloques que calculan integrales con pocos puntos y los que utilizan muchos puntos, por lo que una optimización interesante sería separar estos cálculos en tantos *kernels* diferentes como puntos de

integración se utilizan. En el caso de esta aplicación, dado que se utilizan 2, 4 ó 10 puntos, se crearán tres *kernels* diferentes.

Por último, es interesante observar cómo cuanto más trabajo tiene la aplicación, mayor es la mejora. De hecho, se ha llevado a cabo un experimento en el que el número de puntos de integración es fijo y el máximo posible, es decir, 10. La Figura 7-10 muestra los resultados para el diedro. Para la esfera ocurre algo parecido. Como se ve, la ganancia es considerablemente mayor, y se acerca a 90 para la geometría más grande. El hecho de usar únicamente 10 puntos de integración en todas las integrales de superficie tiene dos consecuencias: por un lado, la cantidad de trabajo es considerablemente mayor, por lo que la GPU tiene más donde mejorar a la CPU. Por otro, todos los hilos de todos los bloques realizan la misma tarea, por lo que el trabajo está equilibrado entre todos ellos. El efecto conjunto es la espectacular mejora que se obtiene. Desgraciadamente, el tiempo de ejecución total es también mucho mayor en este caso, por lo que no es una opción viable, y se ha incluido aquí únicamente para ilustrar la necesidad de homogeneizar en lo posible el trabajo de los bloques de un *kernel*.

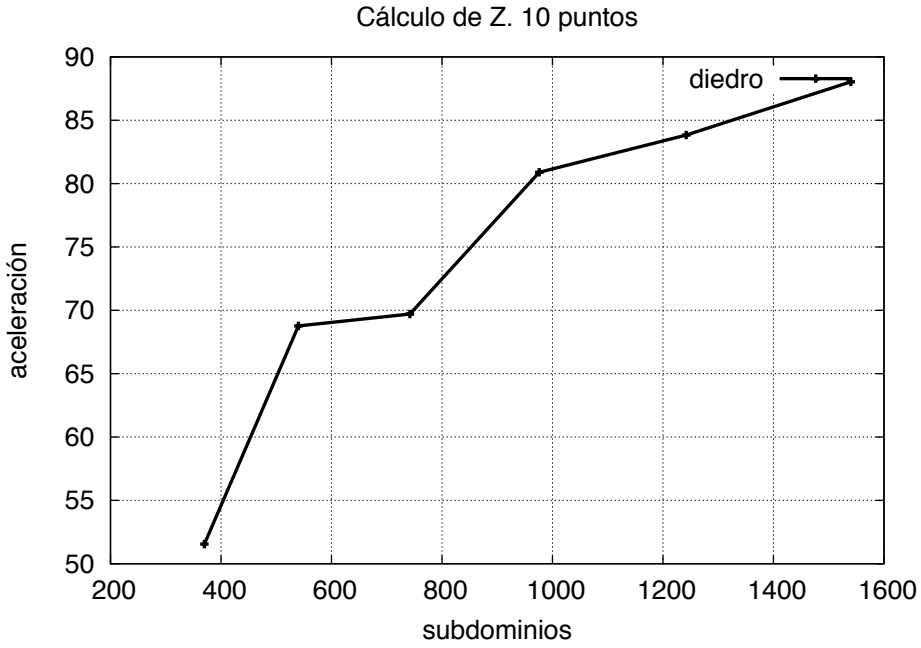


Figura 7-10. Aceleración utilizando 10 puntos de integración.

7.4.2 Reducción de la cantidad de parámetros del *kernel*

La primera cuestión que se trató en el capítulo 4, una vez que se disponía de una versión operativa y correcta de la aplicación, era si es preferible que la CPU prepare los datos para los *kernels*, toda vez que a ella no le afectan los accesos alineados, o si, por el contrario, es más eficiente que cada hilo de la GPU los obtenga por sí mismo accediendo a memoria global. La Tabla 7-2 y la Tabla 7-3 muestran las medidas de tiempo para ambas posibilidades, tanto para el diedro como para la esfera. Las referencias a "base" corresponden a la versión inicial de CUDA, donde se preparan en la CPU. Las referencias a "mejorada", por el contrario, a la versión en la que se obtienen en la GPU.

Se muestran tiempos únicamente para las partes afectadas del programa: el tiempo total de ejecución ("Total"), el tiempo empleado en la GPU en generar los parámetros de los *kernels* ("Parms"), y el tiempo en ejecutar el *kernel* ("*Kernel*"). Éste último se desglosa en el tiempo total del *kernel* ("Total"), el de enviar los parámetros a la GPU ("Copia"), y el tiempo específicamente para ejecutar el *kernel* ("*Kernel*). Además, al final de cada tabla, se muestra la ganancia que obtiene la reducción de parámetros con respecto a la versión base, tanto para el total de la aplicación como para los dos aspectos afectados: la generación de parámetros ("Parms") y el envío a la GPU ("Copia").

Tabla 7-2. Tiempos de ejecución para el diedro.

Diedro							
SDs			154	124	976	742	540
Total		base	3,15	2,09	1,37	0,81	0,48
		modificada	2,47	1,56	1,02	0,62	0,39
Parms		base	1,28	0,83	0,55	0,31	0,53
		modificada	0,78	0,51	0,31	0,19	0,09
Kernel	Total	base	1,62	1,05	0,64	0,36	0,19
		modificada	1,53	0,91	0,56	0,33	0,17
	Copia	base	0,39	0,23	0,16	0,08	0,05
		modificada	0,15	0,10	0,06	0,03	0,02
	Kerne	base	0,93	0,55	0,33	0,19	0,09
		modificada	0,93	0,53	0,32	0,19	0,09
Aceleración							
Total			1,28	1,34	1,34	1,30	1,24
Parms			1,64	1,62	1,74	1,67	5,64
Copia			2,65	2,39	2,60	2,52	2,88

Tabla 7-3. Tiempos de ejecución para la esfera.

Esfera							
SDs			137	11	980	792	588
Total		base	2,64	2,2	1,83	1,05	0,58
		modificada	1,99	1,3	1,04	0,70	0,44
Parms		base	1,05	1,0	0,97	0,40	0,20
		modificada	0,64	0,4	0,32	0,20	0,11
Kernel	Total	base	1,01	0,7	0,52	0,39	0,18
		modificada	0,83	0,5	0,42	0,27	0,14
	Copia	base	0,30	0,3	0,16	0,15	0,06
		modificada	0,12	0,0	0,06	0,03	0,02
	Kernel	base	0,72	0,4	0,36	0,23	0,12
		modificada	0,71	0,4	0,36	0,23	0,13
Aceleración							
Total			1,32	1,6	1,76	1,49	1,33
Parms			1,64	2,4	3,04	1,97	1,79
Copia			2,51	3,75	2,50	4,94	3,87

Lo primero que se puede apreciar es una gran mejora general en la aplicación. Las ganancias obtenidas están, en todos los casos, por encima del 120%, llegando en determinados casos, a superar el 160%. Teniendo en cuenta que estas ganancias no tienen contrapartida (salvo el trabajo adicional de programación), son muy interesantes. Las ganancias en las tablas se definen, como se ha mencionado ya al comienzo de este capítulo, así:

ganancia = (tiempo sin mejora) / (tiempo con mejora).

La mejora proviene de dos aspectos que ya se mencionaron en el capítulo 4. Por un lado, se reduce el tiempo en generar los parámetros, ya que la CPU no tiene que generar la misma cantidad de ellos que antes. La mejora en este apartado es muy buena, y supera el 160% en todos los casos, con algunos incluso superando el 200%. Es muy interesante apreciar que esta descarga de la CPU no

viene acompañada de un recargo en la GPU que la cancele: el tiempo empleado en el *kernel* propiamente dicho es práctiamente idéntico en ambas versiones. Esto es indicativo de que los accesos adicionales necesarios para obtener los datos, a pesar de no ser alineados, no son suficientes (ni suficientemente importantes) como para afectar al tiempo de ejecución.

El segundo aspecto, muy relacionado con el primero, es la reducción en el tiempo que se emplea en enviar esos parámetros a la GPU (entrada "Copia"). Al ser una estructura de datos con menos elementos, el envío es más rápido. La reducción en tiempo de ejecución para esta parte de la aplicación es aún mayor, ya que las ganancias superan holgadamente el 200%, llegando incluso a valores mucho mayores, del orden de 300% e incluso, en un caso, acercándose al 500%.

Los datos mostrados más arriba comparan dos versiones donde se utiliza la GPU. En los apartados siguientes, para dar una idea de cómo la característica que se estudia en cada apartado consigue mejorar el rendimiento de la aplicación frente a la versión secuencial (sin GPU), se van a ir añadiendo las correspondientes gráficas de ganancia o aceleración. La Figura 7-11 es la correspondiente a este apartado: muestra la que se obtiene si se reduce el número de datos que se envía al *kernel*. Se puede apreciar, si se compara con la Figura 7-9, que la ganancia ha mejorado de forma perceptible.

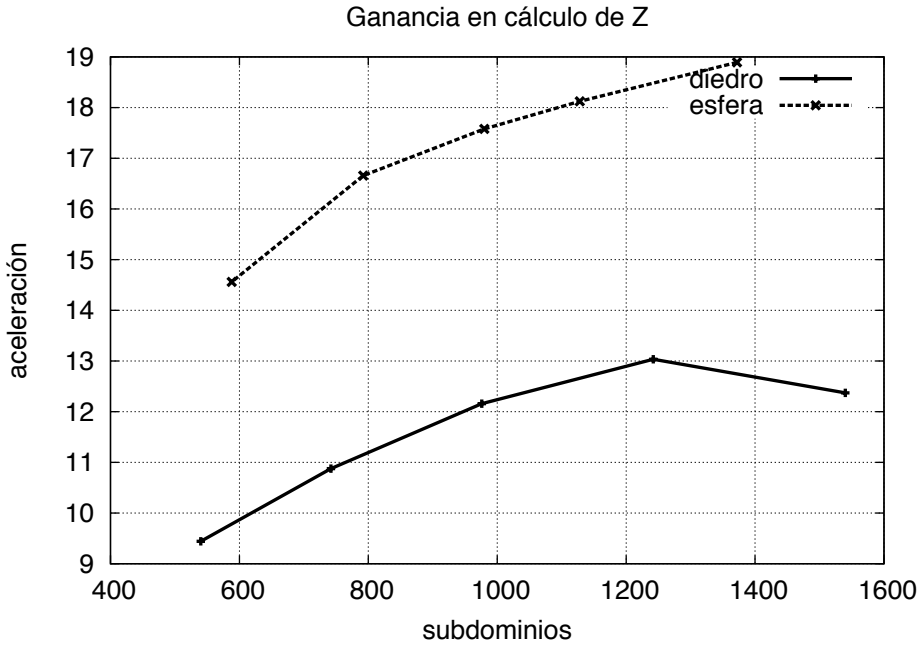


Figura 7-11. Aceleración frente a la versión secuencial.

La conclusión de este apartado es que es beneficioso aprovechar las posibilidades de paralelismo que ofrece la GPU en casi cualquier situación, incluso aunque hacerlo suponga generar una serie de accesos a memoria. La mejora proviene de, por una parte, la ejecución paralela (hilos en ejecución paralela) de los accesos a datos, en la que los accesos no alineados no son un factor muy perjudicial, y por otra, de la reducción del tráfico entre CPU y GPU. Como efecto lateral beneficioso, además, no es necesario dedicar tanto espacio en la GPU para almacenar datos, ya que se usan nada más generarse.

7.4.3 Traslado de cálculos a la GPU

En este caso se trata de trasladar a la GPU cálculos cuyos resultados comparten todos los hilos de cada bloque del *kernel* que calcula las integrales de superficie. La discusión está en si el hecho de

crear un *kernel* adicional para calcularlos se compensa con la descarga de trabajo para la CPU. Los resultados se muestran en las tablas Tabla 7-4 y Tabla 7-5. De nuevo figuran solamente los apartados de la aplicación que se ven afectados por esta comparación: el tiempo total del *kernel* ("Total"), el de la generación de parámetros para los *kernels* ("Parms"), y el de ejecución de los *kernels* ("*Kernels*"), éste último descompuesto a su vez en las fases de copia de datos a GPU ("Copia"), y ejecución del *kernel* ("*Kernels*"). En la versión mejorada este último tiempo incluye la ejecución de ambos *kernels*. Además se muestra la aceleración total de la aplicación y la parcial del cálculo de parámetros para los *kernels*.

Tabla 7-4. Tiempos de ejecución para el diedro.

Diedro							
SDs			1540	1242	976	742	540
Total		base	2,47	1,56	1,02	0,62	0,39
		modificada	2,33	1,51	1,01	0,68	0,47
Parms		base	0,78	0,51	0,31	0,19	0,09
		modificada	0,50	0,32	0,20	0,11	0,06
Kernels	Total	base	1,53	0,91	0,56	0,33	0,17
		modificada	1,55	0,92	0,56	0,33	0,18
	Copia	base	0,15	0,10	0,06	0,03	0,02
		modificada	0,16	0,10	0,07	0,04	0,02
	Kernels	base	0,93	0,53	0,32	0,19	0,09
		modificada	0,94	0,54	0,33	0,19	0,10
Aceleración							
en el total			1,06	1,03	1,01	0,92	0,82
en Parms			1,56	1,57	1,58	1,68	1,62

Tabla 7-5. Tiempos de ejecución para la esfera.

Esfera							
SDs			1372	1128	980	792	588
Total		base	1,99	1,36	1,04	0,70	0,44
		modificada	1,88	1,36	1,08	0,78	0,53
Parms		base	0,64	0,43	0,32	0,20	0,11
		modificada	0,40	0,27	0,20	0,14	0,08
Kernels	Total	base	1,18	0,80	0,59	0,37	0,22
		modificada	1,21	0,81	0,62	0,39	0,22
	Copia	base	0,12	0,08	0,06	0,03	0,02
		modificada	0,13	0,09	0,08	0,05	0,02
	Kernels	base	0,71	0,48	0,36	0,23	0,13
		modificada	0,72	0,48	0,36	0,23	0,14
Aceleración							
en el total			1,06	1,01	0,97	0,90	0,82
en Parms			1,61	1,55	1,58	1,44	1,40

Se puede ver que, en este caso, el efecto es más modesto. En el caso de la aplicación completa, no se obtiene más allá de un 106% de mejora, e incluso puede apreciarse un efecto negativo para problemas pequeños. Esto es debido a que el recargo de lanzar un *kernel* adicional, que es más o menos constante, tiene más peso relativo en un problema pequeño que en uno más grande, donde la ganancia en otras partes de la aplicación pueden compensarlo más fácilmente. Como criterio práctico, para geometrías menores de 1000 subdominios puede resultar aconsejable no hacer el traslado a la GPU, y dejar que se encargue la CPU. Hay que mencionar que en esta comparación, la versión base ya incluye la mejora del apartado anterior, por lo que las mejoras de este apartado (donde las hay) se pueden sumar a las del anterior. La Figura 7-12 representa la ganancia que se obtiene al aplicar, de forma conjunta, las técnicas de este apartado y las del anterior, frente a la versión secuencial de la aplicación.

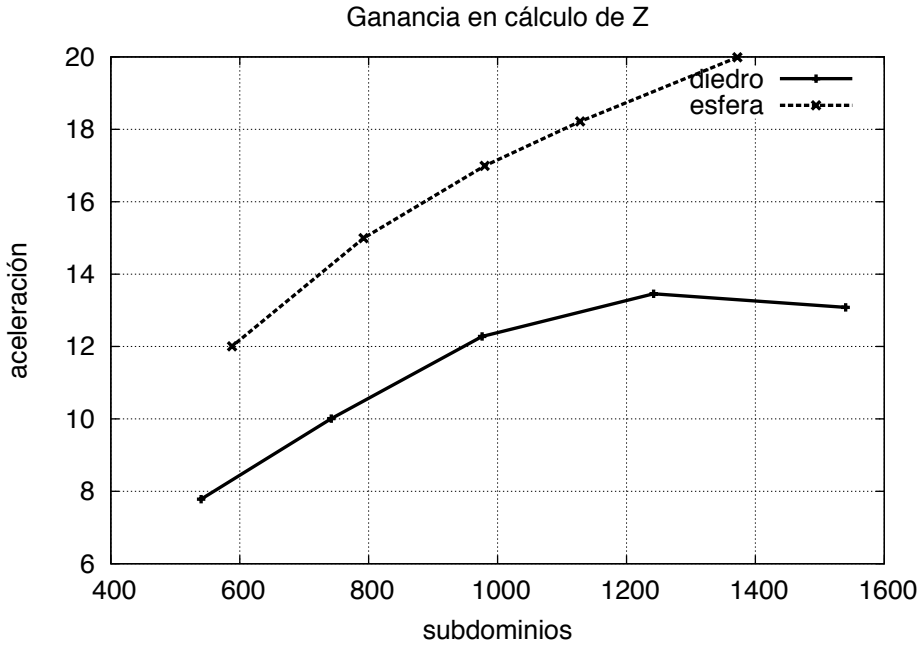


Figura 7-12. Aceleración frente a la versión secuencial.

7.4.4 Organización de los bloques de hilos.

En este apartado se evalúa cómo influye en el tiempo de ejecución de la aplicación el hecho aprovechar en lo posible la capacidad de los bloques de hilos. La versión base incorpora el traslado a la GPU de todos los cálculos discutidos en los apartados anteriores, pero con la organización del *kernel* de la versión inicial, mientras que la mejorada está modificada con respecto a la versión base en que se introducen los "SuperBloques" mencionados en el Capítulo 4 y los cálculos asociados necesarios, fundamentalmente la gestión de las listas de paquetes de hilos.

Las tablas Tabla 7-6 y Tabla 7-7 muestran los tiempos de los kernels de las integrales de superficie para el cálculo del acoplo inductivo y del capacitivo. En el caso de la versión mejorada, el tiempo de los kernels incluye tanto la ejecución del kernel en sí, como

la gestión asociada de las listas, por lo que, además, se desglosan estos dos tiempos. Por último, se muestran las aceleraciones obtenidas en el total de la aplicación, en la ejecución de los *kernels* de las integrales de superficie del inductivo y en la de los del capacitivo. Para simplificar, en el resto de esta sección se indica "*kernel* inductivo" o "*kernel* capacitivo", pero hay que tener en cuenta que se hace referencia a las integrales de superficie exclusivamente.

Tabla 7-6. Tiempos de ejecución para el diedro.

Diedro						
SDs		1540	1242	976	742	540
base	*kernel* inductivo	0,94	0,53	0,33	0,19	0,10
modificada	ks. induc. (total)	0,52	0,34	0,20	0,11	0,08
	gestión listas	0,13	0,08	0,06	0,03	0,02
	kernels (SBs)	0,25	0,17	0,09	0,05	0,05
base	*kernel* cap.	0,40	0,24	0,14	0,08	0,04
modificada	k. cap. (total)	0,09	0,06	0,05	0,02	0,02
	gestión listas	0,08	0,05	0,03	0,02	0,00
	kernel	0,02	0,02	0,02	0,00	0,02
Aceleración						
en el total		2,03	1,78	1,65	1,45	1,31
en k. inducc.		3,76	3,11	3,67	3,80	1,60
en k. capac.		4,50	3,00	2,50	----	1,00

Tabla 7-7. Tiempos de ejecución para la esfera.

Esfera						
SDs		1372	1128	980	792	588
base	*kernel* inductivo	0,71	0,48	0,36	0,23	0,14
modificada	ks. induc. (total)	0,50	0,35	0,27	0,17	0,09
	gestión listas	0,10	0,07	0,05	0,03	0,02
	kernels (SBs)	0,28	0,20	0,16	0,11	0,06
base	*kernel* cap.	0,30	0,20	0,16	0,09	0,05
modificada	k. cap. (total)	0,09	0,06	0,05	0,03	0,02
	gestión listas	0,06	0,04	0,03	0,02	0,02
	kernel	0,02	0,02	0,02	0,02	0,00
Aceleración						
en el total		1,75	1,56	1,61	1,43	1,26
en k. induc.		2,54	2,40	2,25	2,09	2,33
en k. capac.		4,50	3,00	2,50	1,50	----

La aceleración global de la aplicación oscila entre el 130% y el 200% para el diedro, y valores un poco inferiores para la esfera, entre el 126% y el 175%. A la vista de estos resultados, queda completamente justificado seguir las directrices de NVIDIA a este respecto, y aprovechar en lo posible los bloques de hilos. Esto es así incluso en un caso como el nuestro, en el que hacerlo obliga a un cómputo adicional en la CPU como el de las listas.

En este punto es muy importante hacer énfasis en el hecho de que los SuperBloques hacen exactamente el mismo trabajo que los *kernels* a los que sustituyen. Ni lo reducen ni lo aumentan. Y esta carga de trabajo la llevan a cabo de una forma considerablemente más eficiente. Las entradas de las tablas en los apartados de aceleración dejan ver este fenómeno. Para el acoplo inductivo, las ganancias están en torno al 300% con el diedro y al 200% con la esfera. En el capacitivo es aún mayor, pero esto se ve afectado por el hecho de que esta parte de la aplicación ocupa muy poco tiempo, por

lo que se da una cierta irregularidad en los resultados. Además, tiene muy poco peso sobre el cálculo global, así que su mejora no es significativa para la aplicación.

La Figura 7-13 compara el rendimiento de la versión CUDA con SuperBloques y el de la versión secuencial. Se puede ver que, con esta forma de organizar los bloques, se produce un despegue en el rendimiento de la GPU. Las características anteriores mejoran el rendimiento en una medida mucho menor a como lo hace el hecho de diseñar convenientemente un *kernel*.

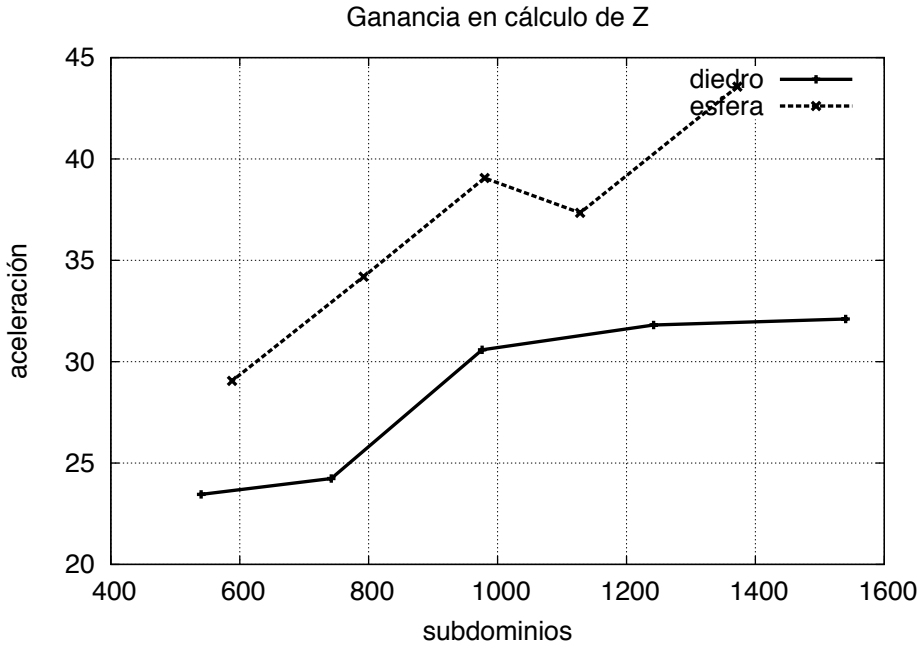


Figura 7-13. Aceleración frente a la versión secuencial.

Como conclusión, este apartado ilustra una de las enseñanzas más importantes de esta tesis: un diseño descuidado de la aplicación puede desaprovechar los recursos de la GPU y presentar rendimientos considerablemente inferiores a los posibles, como era el caso en la aplicación inicial. La forma de funcionar de una GPU es

completamente diferente a la de una CPU tradicional, por lo que es preciso invertir en su estudio y en el de la organización de los cálculos en ella. Los frutos que se pueden recoger merecen el esfuerzo.

A modo de resumen, en las siguentes gráficas (Figura 7-14 y Figura 7-15) se muestran las ganancias que se pueden obtener si se hace uso de lo explorado hasta el momento. Ambas representan la ganancia obtenida frente a la versión secuencial. Se etiqueta como "base" la versión inicial, como "reduc." la versión que reduce la cantidad de parámetros enviados al *kernel*, como "calc." la que traslada parte del cálculo de los parámetros a la GPU, y, por último como "SuperBs" la que utiliza SuperBloques. Como se puede observar, la versión base tiene un rendimiento relativamente bueno (reducir a un séptimo - para problemas un poco grandes ya - el tiempo de ejecución es, en general un rendimiento más que generoso, pero para CUDA no es espectacular). La reducción de parámetros, sea por envío, sea por cálculo, dobla la mejora obtenida por la versión base. Pero el salto cualitativo se obtiene con la utilización de SuperBloques, ya que el aprovechamiento de todos los hilos de los bloques es el factor que hace que un *kernel* explote al máximo los recursos computacionales de la GPU.

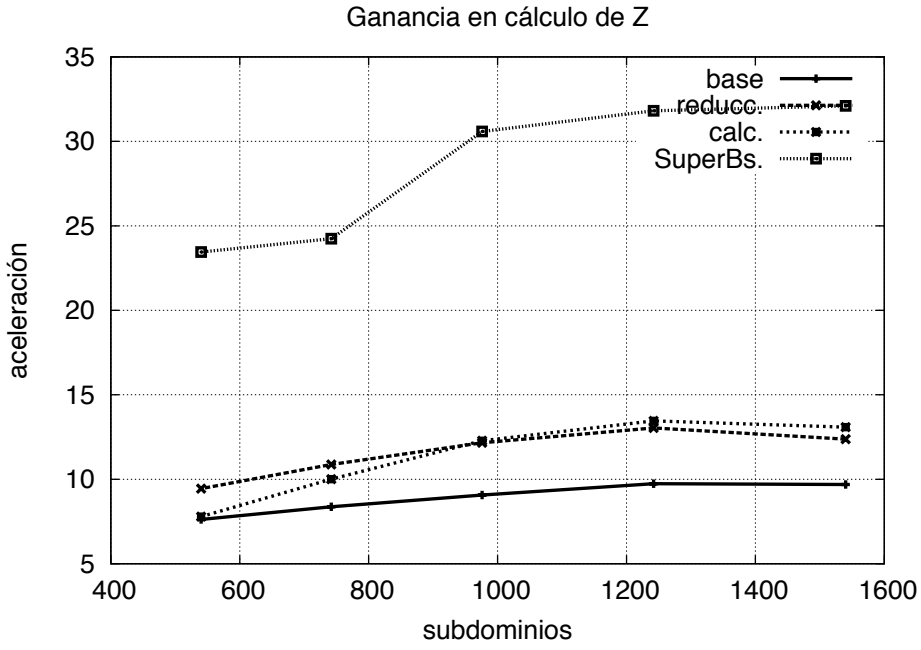


Figura 7-14. Ganancias en el diedro.

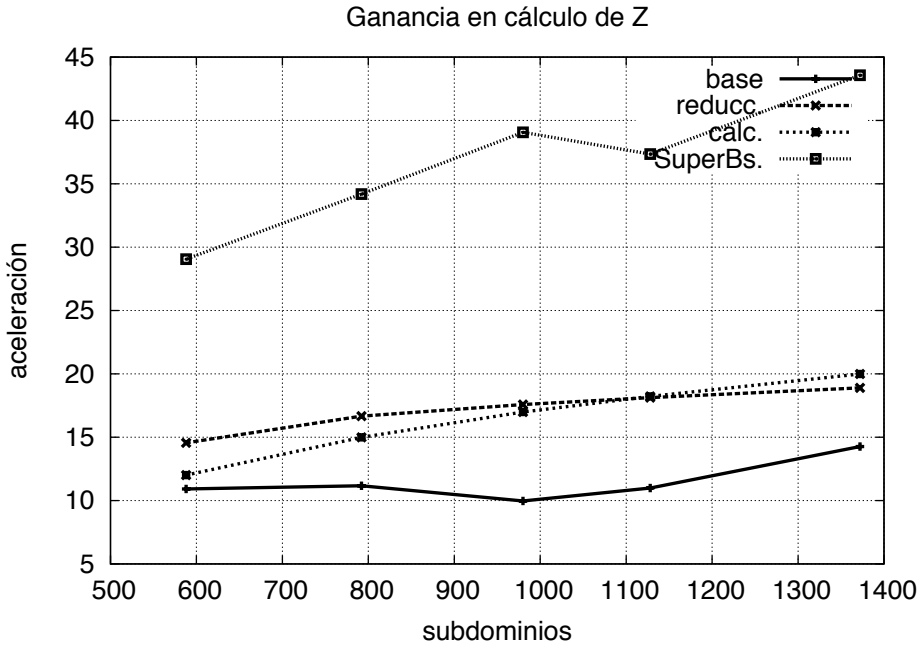


Figura 7-15. Ganancias en la esfera.

7.4.5 Reducción del almacenamiento intermedio.

El último de los experimentos relacionados con el MoM es la eliminación de la necesidad de un almacenamiento temporal entre las integrales de superficie y las de línea. Para conseguir esto, los *kernels* encargados de las primeras acumulan sus resultados sobre la matriz Z reproduciendo el cálculo de las segundas, haciendo a éstas innecesarias. De esta forma, no se han de guardar esos resultados para ser procesados luego. Las tablas Tabla 7-8 y Tabla 7-9 muestran los tiempos para la versión base (con almacenamiento intermedio) y la modificada (sin él). Ambas implementaciones utilizan todas las mejoras vistas hasta este momento (por ejemplo, los SuperBloques). Se incluyen en las tablas el tiempo total de la aplicación, y los empleados en cada tipo de *kernel* (el tiempo total, incluyendo copias a GPU, ejecución, etc).

Tabla 7-8. Tiempos de ejecución para el diedro.

Diedro						
SDs		1540	1242	976	742	540
Total	base	0,95	0,64	0,41	0,28	0,16
	modificada	1,28	0,83	0,56	0,34	0,19
kernel inductivo	base	0,52	0,34	0,20	0,11	0,08
	modificada	0,89	0,58	0,36	0,22	0,11
kernel capacitivo	base	0,09	0,06	0,05	0,02	0,02
	modificada	0,09	0,06	0,05	0,02	0,00
k. integral línea	base	0,05	0,03	0,02	0,02	0,00

Tabla 7-9. Tiempos de ejecución para la esfera.

Esfera						
SDs		1580	1372	1128	980	792
Total	base	1,12	0,86	0,66	0,47	0,34
	modificada	1,53	1,17	0,84	0,64	0,45
kernel inductivo	base	0,66	0,50	0,35	0,27	0,17
	modificada	1,13	0,86	0,61	0,45	0,30
kernel capacitivo	base	0,11	0,09	0,04	0,05	0,03
	modificada	0,10	0,08	0,06	0,05	0,03
k. integral línea	base	0,05	0,04	0,00	0,02	0,02

A diferencia de apartados anteriores, en este lo que ocurre es que la modificación introducida no lleva consigo una mejora en el rendimiento de la aplicación, sino más bien lo contrario: se puede observar cómo los tiempos totales de ejecución de la aplicación son consistentemente mayores en la versión modificada. La diferencia no es enorme, generalmente se mueve en torno a unas décimas de segundo, pero puede llegar a ser considerable. Es el caso de las geometrías grandes, tanto para diedro como para esfera.

Eliminar el almacenamiento intermedio se consigue con un coste: el de que los hilos que almacenan resultados parciales lo hacen en zonas de memoria compartida con otros hilos, como se explica en el apartado 4.4.5. La sincronización necesaria para asegurar la corrección de los resultados obliga a que, cuando un hilo está accediendo a un elemento de Z para actualizarlo, todos los demás que trabajan sobre el mismo hilo deben esperar. Esos retardos se van acumulando, y al final acaban influyendo sobre el tiempo total de la aplicación. Los tiempos de ejecución en los demás apartados de la misma no sufren variación. Esto se refiere a todos en general, y

especialmente a la gestión de las listas. El aumento de tiempo de ejecución que aparece en las tablas se debe exclusivamente a la sincronización de los hilos que actualizan los resultados en la matriz Z.

El interés de esta implementación radica en que, si se elimina el almacenamiento intermedio, se pueden procesar geometrías más grandes. La Tabla 7-10 muestra los tamaños máximos que caben en la GPU con ambas implementaciones.

Tabla 7-10. Tamaños máximos que admite la GPU.

Diedro	
base	2232
modificada	4522
incremento	203%
Esfera	
base	2884
modificada	4384
incremento	152%

Los tamaños máximos obtenidos muestran cierta disparidad entre ambas geometrías. La causa de esto es que el espacio en memoria global que una geometría necesita depende de los puntos de integración que se utilizan en las cuchillas, ya que cada punto requiere una integral de superficie, y, por lo tanto, datos que trasladar al *kernel* para trabajar. Puesto que el número de puntos en la cuchilla depende de la cercanía entre los subdominios cuyo acoplo se calcula, geometrías más "concentradas" necesitarán más espacio que otras más "extensas", siempre comparando geometrías con número de subdominios parecido.

Independientemente de este hecho, que es consecuencia del método de análisis electromagnético, lo que sí es cierto es que, a cambio de una pérdida de rendimiento, eliminar el almacenamiento intermedio permite aumentar considerablemente los tamaños (medidos en números de subdominios) de las geometrías analizadas. Cuando las geometrías se hacen grandes, hacerlo permite tratarlas en un sólo bloque. En estos casos, la mejora en el rendimiento radica en que no eliminar el almacenamiento intermedio obligaría a tratarlas en dos o más bloques, lo que supone una penalización aún mayor que la que supone sí hacerlo e incurrir en los retardos que impone la sincronización.

Como conclusión de este apartado, si las geometrías bajo análisis son pequeñas, es más interesante mantener la secuencia de *kernels* original. Pero, a partir de un determinado tamaño (aquel que llena la memoria de la GPU), el mejor rendimiento lo da eliminar el último *kernel* para no utilizar almacenamiento intermedio. Si la geometría cabe entonces completa en la GPU, mucho mejor. Pero, aun si no es así, y hay que procesarla por fragmentos, éstos podrán ser mayores (y por lo tanto menos), con lo que el tiempo de proceso total es menor.

7.5 Aceleración del Método de las Funciones Base Características.

En esta sección se presentan los resultados de la aceleración de Método de las Funciones Base Características (CBFM). Dado que el objetivo principal del CBFM es reducir el tamaño del problema, además de las gráficas de aceleración se incluyen otras en las que se muestra cuánto se reduce el problema. Si se trata la geometría en un único bloque, los resultados serán parecidos a otros trabajos sobre el

CBFM ([11], [67], [68]), pero la particularidad de que en éste se usan GPUs para el cálculo (que tienen una memoria más reducida, por lo que tienen que trabajar con bloques más pequeños) hace interesante mostrar cómo se reduce el problema al tratarlo por bloques.

Los criterios y los métodos para la medición de tiempos son los mismos que los del apartado anterior. Se utilizan todas las optimizaciones del compilador disponibles, tanto para la versión original secuencial como para la versión acelerada, y la máquina sobre la que corren los programas es la Máquina 1. Las gráficas de aceleración comparan el tiempo necesario para obtener la matriz reducida del CBFM, lo cual incluye el cálculo de la matriz de impedancias del MoM. De esta manera, la aceleración obtenida es una especie de promedio global de todo el proceso. Cuando las geometrías son muy grandes, la memoria limitada de la GPU hace necesario dividir la geometría en bloques. Esta división no se ha aplicado en el caso de la versión original, sólo en la versión acelerada. El hecho de dividir en bloques causa una pérdida de rendimiento, pero se ha considerado más adecuado realizar la partición sólo en la versión acelerada, ya que la CPU no lo necesita, y aplicarla sería, en este caso, obligar a la CPU a competir con una limitación artificial que no tiene en realidad.

Las geometrías utilizadas para este apartado son, fundamentalmente, el avión y la cavidad Cobra, descritas en el apartado 7.2. Se pretende trabajar con geometrías mayores. De forma parecida a lo que se hace en la sección 7.4, se utilizan varios factores de discretización para variar la cantidad de subdominios de los que constan las geometrías.

7.5.1 Geometría en un único bloque.

En esta subsección se muestran los resultados de la aceleración del CBFM cuando la geometría se puede procesar en un único bloque. El CBFM está pensado para atacar una geometría grande en varios bloques, y dado que al dividirla de esta manera se pierde rendimiento, se pueden considerar los resultados obtenidos aquí como un óptimo de referencia. La Figura 7-16 muestra la aceleración para el caso del avión. Dado que la aplicación acelerada puede procesar geometrías de hasta unos 3500 subdominios de una vez, éste es el máximo que se ha utilizado.

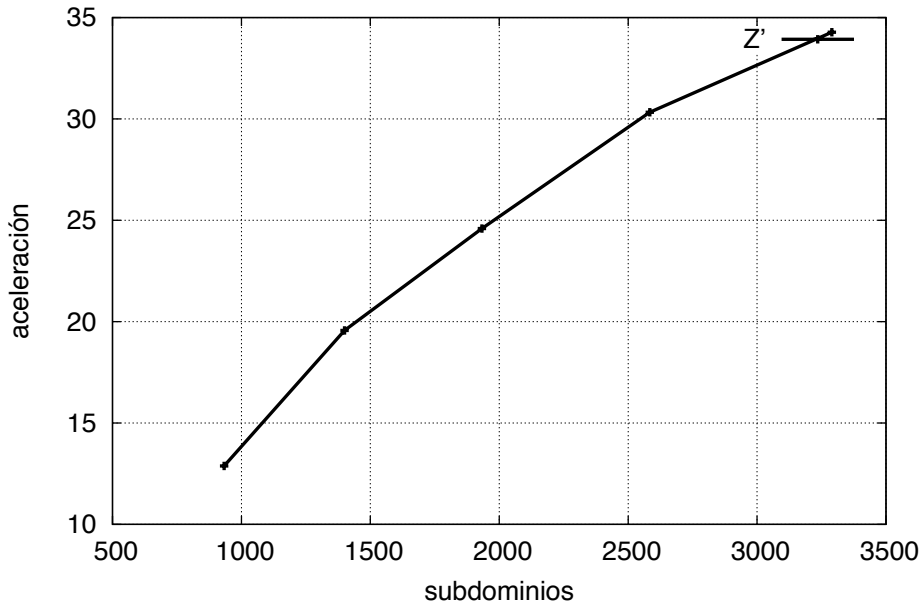


Figura 7-16. Aceleración del CBFM para el avión en un solo bloque

Como se puede ver, cuanto mayor es la geometría, mejor es el rendimiento obtenido. Esto es análogo a lo que ocurre con la aceleración del MoM, y la razón es la misma. Con una geometría

mayor hay más trabajo y más margen para mejorar. Además, los tiempos de inicialización y transferencia se mantienen prácticamente constantes para todos los tamaños, por lo que el peso relativo sobre el total es menor para los mayores.

Por otro lado, es interesante ver que la aceleración es algo menor que la conseguida en el caso del MoM con geometrías parecidas en cuanto al tamaño. Esto es porque las operaciones del CBFM son menos paralelas que el MoM. En concreto, la solución de un sistema de ecuaciones lineales o la descomposición en valores singulares, que son los pasos principales para el CBFM.

La Figura 7-17 muestra la medida en la que el CBFM reduce el problema para el caso del avión tratado en un solo bloque. Por ejemplo, el punto de abscisa mayor corresponde a una reducción de 3290 subdominios a 90 CBFs, por lo que el factor es 36.5. El factor de reducción es lo que hace interesante al CBFM y la razón principal de su uso. Sin embargo, al igual que en el caso de la aceleración, cuando se procese la geometría en varios bloques este factor será menor, y la reducción menos ventajosa.

A la hora de considerar la mayor geometría que es posible tratar con el CBFM, el factor limitador es la solución del sistema de ecuaciones lineales en el espacio de las CBFs. Esto es así porque el resto del proceso se puede particionar en bloques, pero, puesto que en esta tesis consideramos un método de solución directa, el sistema se debe resolver de una vez. Nuestras mediciones indican que el sistema mayor que admite la GPU tiene del orden de 13000 incógnitas. Teniendo en cuenta la máxima reducción que se consigue para un bloque, que es del orden de 36.5, esto implicaría que el método puede tratar una geometría de 450000 subdominios. Esto es el máximo

teórico, sin embargo, ya que, al tratar la geometría en varios bloques, el factor de reducción del problema puede disminuir.

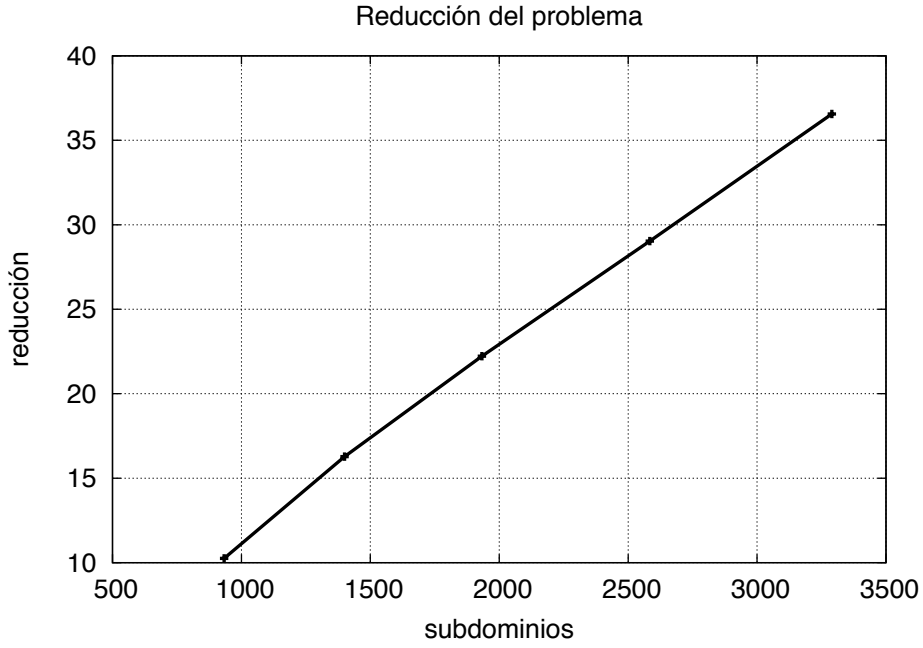


Figura 7-17. Reducción del problema con el CBFM en un solo bloque

7.5.2 Geometría en varios bloques.

Cuando la geometría se hace muy grande, es preciso tratarla por bloques, ya que no caben todos los datos en la memoria. Esto es así tanto para la aplicación original secuencial como para la versión de CUDA, aunque en ésta última, dado que la memoria de la que dispone es menor, los trozos en los que se divide han de ser menores.

El método de partición está incluido en la aplicación original, y es una de las partes que no se ha trasladado a CUDA, por lo que no es posible modificarlo. Explicado de forma sencilla, a partir del origen de coordenadas, se crean cajas con forma de ortoedro. Todos los subdominios que se encuentren dentro del mismo ortoedro se asignan

al mismo bloque. Es posible especificar las dimensiones del ortoedro, pero no el punto a partir del que se empiezan a formar, por lo que no se tiene un control completo sobre cómo se particiona la geometría.

Como se ha explicado en las subsecciones anteriores, el hecho de particionar la geometría provoca una pérdida de rendimiento con respecto a tratarla en un solo bloque. Sin embargo, generalmente es la única forma de procesarla, por lo que es un mal necesario. En esta sección nos interesa averiguar cuánto se pierde en el proceso.

Las dos figuras siguientes muestran los resultados de este análisis. En ambos casos, tanto para el avión como para la cavidad Cobra, la aceleración se mide como la relación entre el tiempo de cálculo de la versión original (secuencial, sin usar CUDA), que no divide la geometría en bloques, y la versión acelerada, que sí lo hace. El no particionar la geometría en la aplicación original tiene dos razones: por un lado, es la forma de analizar el efecto de la división en bloques. Comparando la aceleración en este caso con los datos que se obtenían en el apartado anterior se puede estimar este efecto. Por otro lado, la CPU no tiene las limitaciones de espacio que sí tiene la GPU, por lo que es justo dejar que explote uno de sus puntos fuertes.

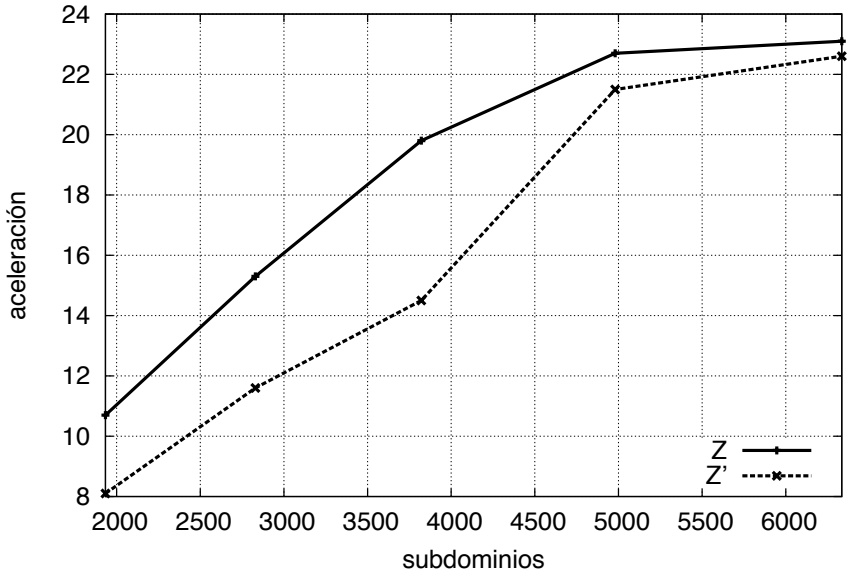


Figura 7-18. Aceleración del cálculo de Z y ZR (Z') para el avión en varios bloques

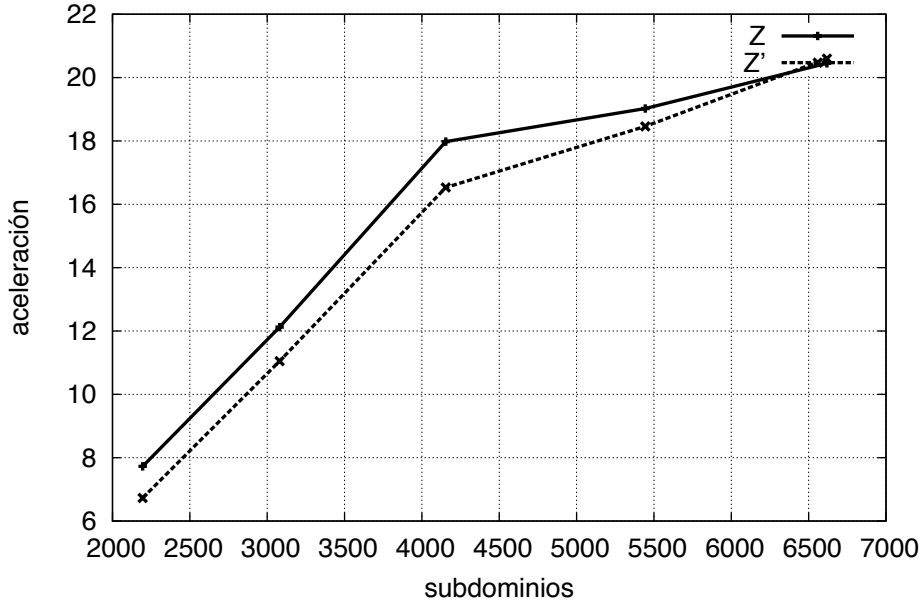


Figura 7-19. Aceleración del cálculo de Z y ZR (Z') para la cavidad cobra en varios bloques.

La Figura 7-18 muestra la aceleración que se obtiene al calcular la matriz de impedancias Z y la matriz reducida ZR para el avión aplicando una partición en bloques sin optimizar. Como se puede apreciar de la comparación con la Figura 7-16, la aceleración se reduce considerablemente. Se ha optado por incluir los datos para la matriz de impedancias Z para que se observe que el cálculo de la misma también se ve afectado por la división de la geometría en bloques. La partición que se ha realizado no busca optimizar, y es la misma para todos los tamaños de la geometría. La Figura 7-19 muestra los mismos resultados para la cavidad Cobra, y presenta unos valores muy parecidos al caso del avión.

A la hora de hacer la partición, la única limitación realmente es que los bloques quepan en la GPU. Como se ha demostrado en el apartado 7.4, en general, cuanto mayor es la geometría, mejor el rendimiento. Y esto se debería poder aplicar a cada bloque de la geometría de forma individual: cuanto mayores sean los bloques, mayor debería ser la aceleración respecto a la ejecución secuencial. De esta manera, una partición más adecuada sería aquella que intentara hacer los bloques lo más grandes posibles, respetando siempre que quepan en la GPU. Para comprobar esto, se realiza un experimento en el que se mantiene fijo el factor de discretización y se mide cómo se comporta la aceleración cuando se varía el tamaño de los bloques o regiones de la geometría.

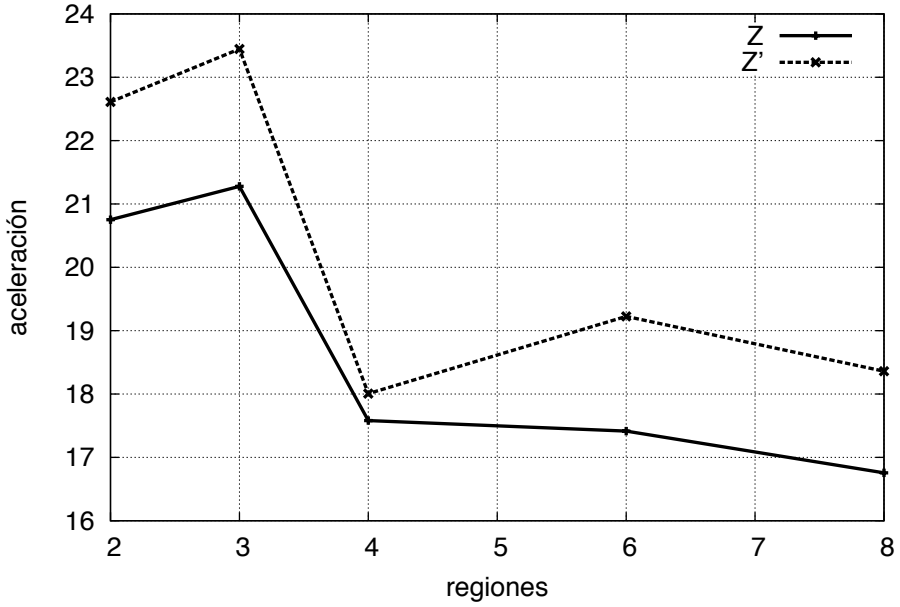


Figura 7-20. Aceleración para distinto número de regiones o bloques.

La Figura 7-20 muestra el resultado para el avión. Para el factor de discretización mayor, lo que proporciona unos 6300 subdominios, se varía el tamaño de los bloques (y, por lo tanto, el número de los mismos). Lo que se observa valida la hipótesis: el rendimiento aumenta a medida que el número de bloques disminuye, es decir, al hacerse éstos más grandes. El valor mayor de la aceleración se consigue con 2 ó 3 bloques.

También es interesante ver cómo afecta el tamaño de los bloques a la reducción del problema. Esto se muestra en la Figura 7-21 para el avión. En este caso, de nuevo se obtiene que cuanto mayores son los bloques, mayor es la reducción del problema.

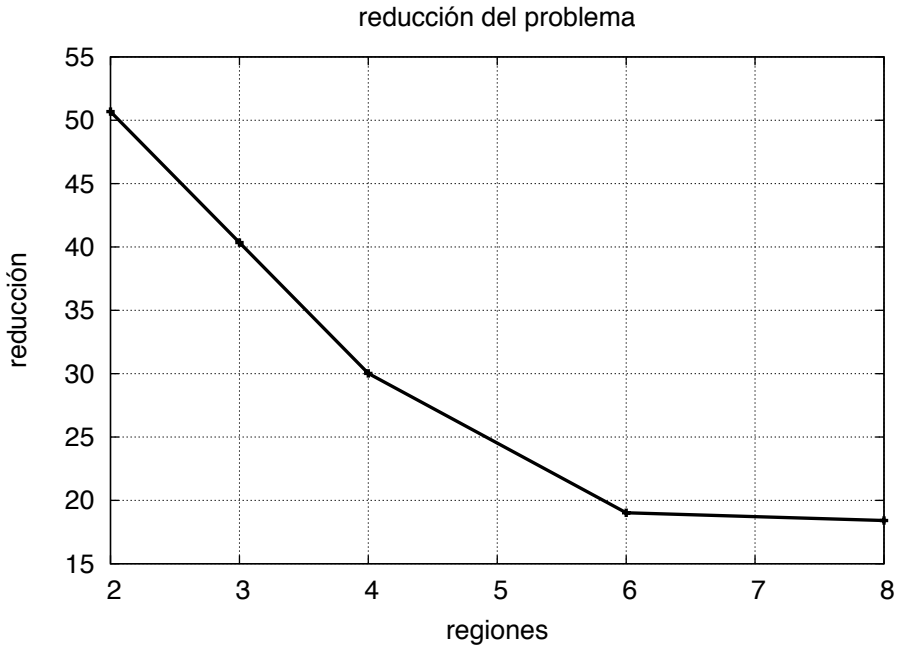


Figura 7-21. Reducción del problema del avión, para diferente número de regiones

La conclusión de este estudio es que, para conseguir una aceleración interesante, es preciso seleccionar el tamaño de los bloques para que sea el mayor posible. Este parámetro vendrá determinado por la capacidad de memoria de la GPU. Este valor proporcionará, además de una mayor aceleración, una mayor reducción del problema. Para comprobar esto, se elige, para cada factor de discretización, la división en regiones más adecuada, que será aquella que tenga las regiones más grandes. Las dos gráficas siguientes muestran los resultados para el avión:

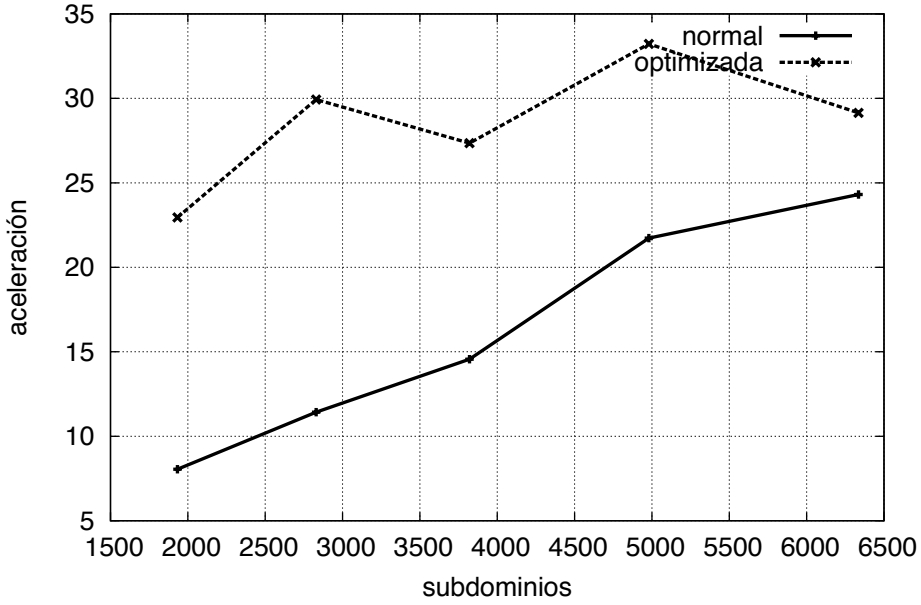


Figura 7-22. Aceleración para dos formas de dividir en bloques. Cálculo de ZR.

En la Figura 7-22, una de las gráficas, la correspondiente a la versión normal, es la misma que ya se mostró en la Figura 7-18. La versión optimizada es claramente superior. El aspecto de dientes de sierra es consecuencia de la optimización: para un tamaño mayor, la partición en bloques puede dar un resultado de peor calidad que para un tamaño menor de la geometría. En cualquier caso, la aceleración es superior en al menos 5 puntos, que es el caso de menor diferencia entre ambas versiones.

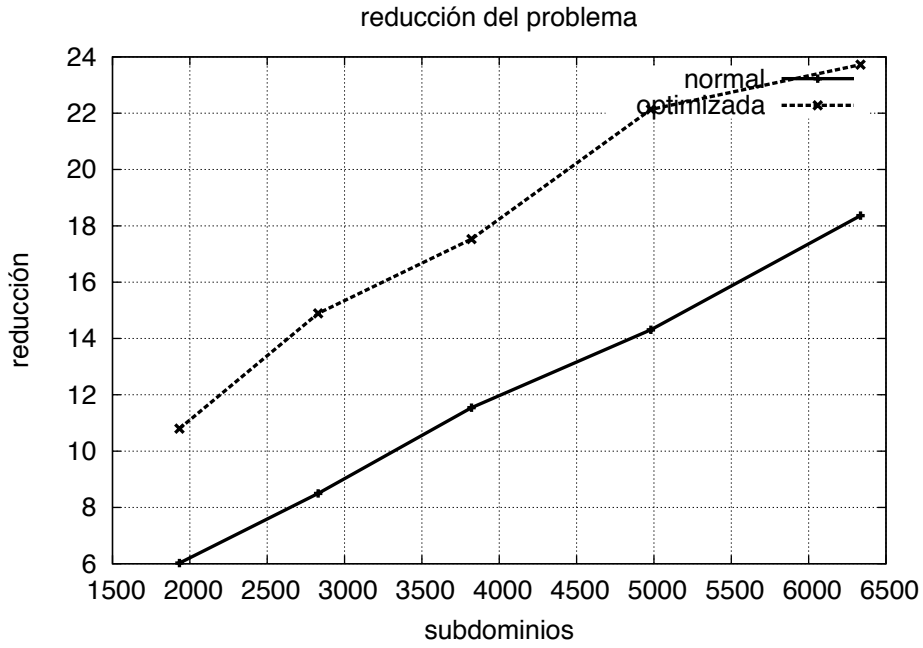


Figura 7-23. Reducción del problema para dos formas de dividir en bloques.

Con respecto a la reducción del problema, ocurre igual. El CBFM reduce mejor (de forma relativa) si tiene más subdominios sobre los que optimizar, y esto es precisamente lo que ocurre si se hacen los bloques lo más grandes posible. En estas condiciones, y con los mejores datos de la Figura 7-23, la mayor geometría tratable (sobre el caso del avión) tendría 310000 subdominios.

Como conclusión de este apartado se podría sacar que, a la hora de dividir una geometría en bloques para que pueda ser procesada por la GPU, es interesante invertir en la forma de particionarla. Si se consigue ajustar el tamaño de los bloques al máximo que puede procesar la GPU, las ventajas obtenidas pueden llegar a ser considerables. Si el método de particionado no genera bloques homogéneos en cuanto al número de subdominios que contienen, y puesto que el factor decisivo va a ser el tamaño del mayor de ellos, es

posible que no se obtenga el máximo partido de la GPU. La forma de solucionar esto es un método de particionado que cree bloques de tamaño parecido, para que todos ellos exploten al máximo la capacidad de la GPU y el CBFM reduzca el problema lo más posible. Si el método de particionado no es modificable, como en el caso de este trabajo, aún así es posible mejorar tanto el rendimiento de la aplicación paralela como la capacidad de reducción del problema del CBFM ajustando el tamaño de los bloques de la geometría de forma individualizada. Los resultados que se obtienen de esta manera justifican el esfuerzo extra.

7.6 Utilización de varias GPUs.

En este apartado se muestran los resultados obtenidos de las distintas versiones de la aplicación creadas para utilizar múltiples GPUs. La tecnología CUDA no impone ninguna limitación al respecto del número de tarjetas que se pueden utilizar. Sin embargo, los requisitos que plantean las tarjetas en cuanto a la alimentación hacen inviables equipos con más de 4 tarjetas conectadas a la misma placa base. Existen placas base que integran, además de la CPU o las CPUs, una o dos tarjetas CUDA, y que están diseñadas para que se puedan montar varias placas base en un mismo equipo, con lo que la cantidad de tarjetas es mucho mayor. Sin embargo, el precio de estos equipos es también muy superior a uno de uso corriente.

La máquina sobre la que corren las pruebas cuyos resultados se muestran aquí es la Máquina 2 que se menciona en el apartado 7.1 Es una solución intermedia entre un PC de tipo medio y uno de esos equipos con múltiples placas base: se trata de una máquina con 3 GPUs conectadas a una placa base con una única CPU. Tiene un

procesador Intel Xeon E5520, con 8GB de memoria RAM, y 3 GPUs C1060 de 4GB de memoria RAM de vídeo.

Con respecto a las pruebas realizadas, la geometría considerada a lo largo de todo este apartado es el avión. Se utiliza un factor de discretización de 10, lo que da unos 6300 subdominios, y se consideran distintas divisiones en bloques: 2, 3, 4, 6 y 8.

7.6.1 Versión 1.

En primer lugar, se muestran los resultados de ejecutar la Versión 1 de OpenMP. Esta versión no tiene ningún tipo de control sobre cómo se asignan las partes de la matriz de impedancias a los hilos de CPU. Se consideran en primer lugar tantos hilos como GPUs, y se varía el número de GPUs desde 1 hasta 3. La Tabla 7-11 muestra los tiempos de ejecución medidos.

Tabla 7-11, Tiempos (s) de ejecución de la Versión 1 de OpenMP

Nº de Bloques	**2**	**3**	**4**	**6**	**8**
1 GPU	68.43	70.09	81.68	84.12	87.96
2 GPUs	39.78	47.53	43.26	42.96	45.16
3 GPUs	40.31	34.80	40.75	41.47	42.09

Los resultados obtenidos no son sorprendentes: salvo algún caso aislado (3 bloques con 2 GPUs), se cumple que, cuantos menos bloques (es decir, cuanto mayores son los bloques), menor es el tiempo necesario para procesar. Por otro lado, cuantas más GPUs, también es menor el tiempo.

Algún dato sí es interesante resaltar. Con 2 bloques, el tiempo para 3 GPUs no mejora el tiempo para 2 GPUs. Esto es razonable,

ya que la tercera GPU no se utiliza en la práctica: la diagonal de la matriz de impedancias tiene 2 bloques, y también hay 2 submatrices fuera de la diagonal. Dada la estructura de la aplicación (explicada en el Capítulo 5), no se puede obtener más rendimiento.

También es interesante observar que es más favorable en muchos casos utilizar sólo 2 GPUs y mantener el número de regiones en 2 que utilizar una tercera GPU y aumentar el número de regiones. De hecho, con esta versión, la tercera GPU no añade mucho, en general.

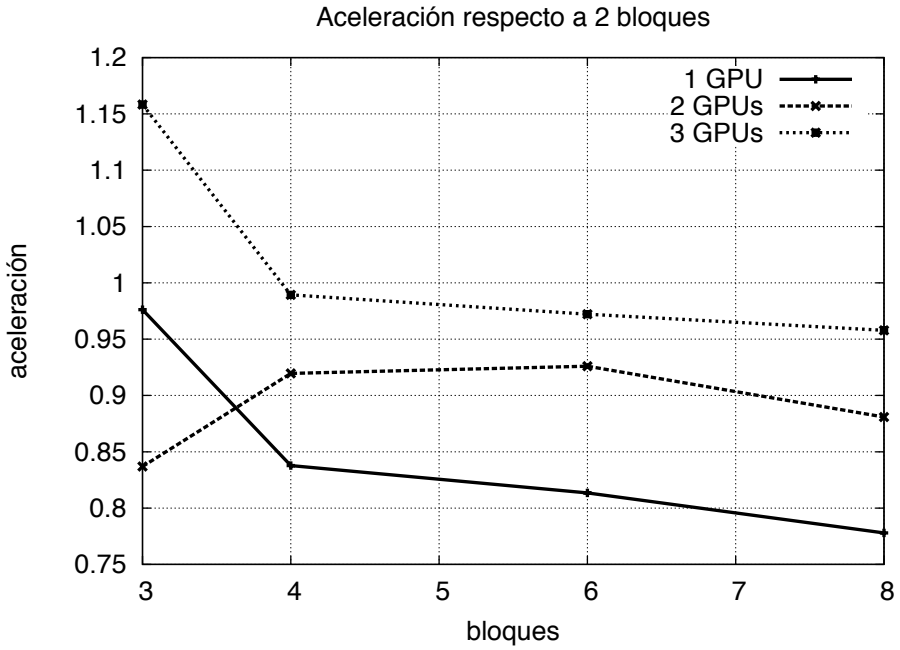


Figura 7-24. Comparación con 2 bloques

La Figura 7-24 muestra la aceleración relativa para distintos números de bloques en la geometría, tomando como referencia la división en 2 bloques. Valores menores que 1 indican que la división en 2 bloques es más rápida. Como se puede ver, en general, cuantas

más GPUs, menor es la diferencia entre dividir en 2 bloques y las demás opciones.

La Figura 7-25 muestra cómo varía la aceleración de cada forma de dividir la geometría a medida que se van aumentando las GPUs, es decir, para cada número de bloques, se muestra la aceleración con respecto a ese mismo número de bloques con una única GPU. Lo ideal es que la gráfica fuera lineal, de forma que cuando se usan 3 GPUs la aceleración fuera 3. El resultado real es algo diferente. Cuando se usan 2 GPUs, los valores de aceleración sí se acercan a 2. No alcanzan este valor, lo cual es aceptable, ya que siempre se pierde rendimiento gestionando los distintos hilos que se crean y en las partes no paralelizables de la aplicación. Sin embargo, con 3 GPUs la mejora es prácticamente inexistente. Los resultados son parecidos para cualquier división en bloques, y se muestran sólo algunos de ellos para hacer la gráfica más fácil de analizar.

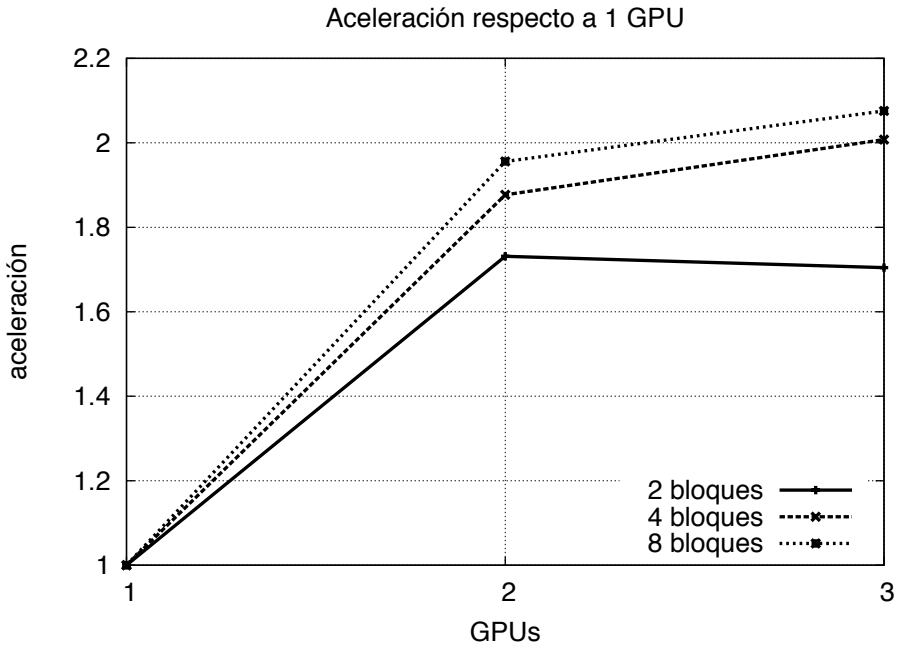


Figura 7-25. Comparación con 1 GPU

El hecho de que añadir una tercera GPU no proporcione ninguna mejora de consideración indica que es preciso mejorar la distribución del trabajo. Una posible mejora es utilizar dos hilos por cada GPU, de forma que las GPUs se aprovechen mejor. Esto es lo que muestra la Figura 7-26. Los resultados no son mucho mejores que antes. Es cierto que con 3 GPUs se consigue una aceleración algo mayor, pero esto es a costa de reducir la aceleración para 2 GPUs.

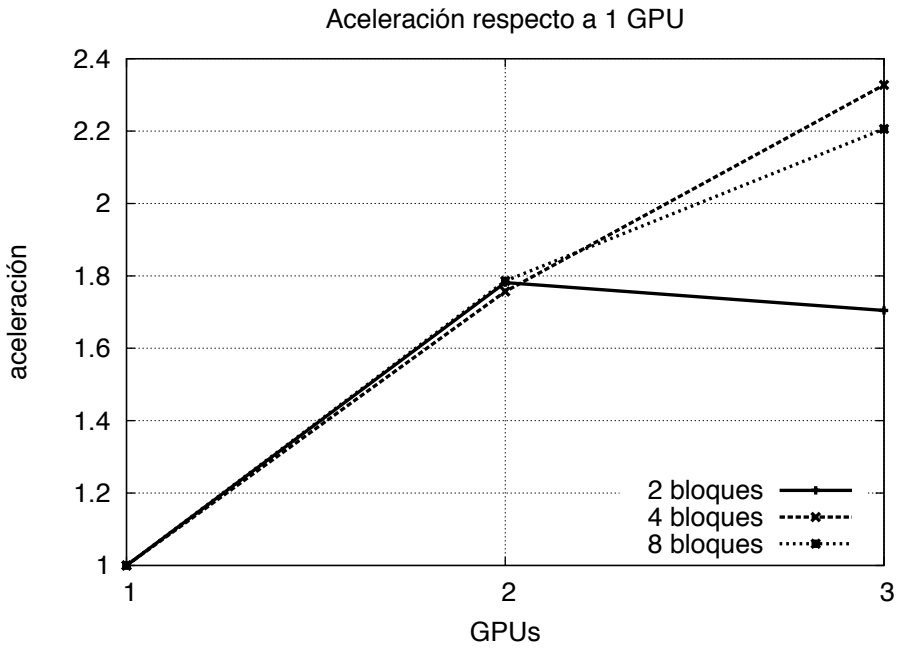


Figura 7-26. Aceleración con dos hilos por GPU.

Para entender bien lo que pasa es preciso estudiar cómo se ejecuta la aplicación. En primer lugar, y para acotar el problema, se pueden analizar las distintas fases del mismo. En concreto, en un primer paso se procesan las submatrices en la diagonal de la matriz de acoplos. Una vez hecho esto, tal y como se explica en el capítulo 5, se procede a tratar las submatrices que están fuera de la diagonal. La Tabla 7-12 muestra los tiempos de ejecución de estas dos fases, que son las que más tiempo consumen, para distintas configuraciones del problema.

Tabla 7-12. Desglose de tiempos de ejecución

		Bloques				
GPUs/Hilos		**2**	**3**	**4**	**6**	**8**
2/2	Diagonal	25.647	28.642	21.621	19.703	18.798
	No Diagonal	13.962	18.704	21.45	23.088	26.161
2/4	Diagonal	25.303	21.216	20.249	25.694	20.623
	No Diagonal	13.369	20.936	27.035	24.632	28.907
3/3	Diagonal	25.959	21.465	21.981	21.715	21.84
	No Diagonal	14.165	13.151	18.579	19.547	20.046
3/6	Diagonal	25.709	21.372	17.878	16.224	19.016
	No Diagonal	14.181	18.486	17.004	21.871	20.483

Los tiempos están expresados en segundos, y se muestran 2 y 3 GPUs, para 1 y para 2 hilos de CPU por cada GPU. Se divide la geometría en un número de bloques que varía entre 2 y 8. En general se puede decir que los tiempos para submatrices fuera de la diagonal no varían demasiado si se usa un hilo de CPU por cada GPU o si se usan dos. Se da alguna excepción, pero éste es el comportamiento general. La variación principal se da en la diagonal. Y, en el caso de las 2 GPUs, la variación es a menudo a peor, por lo que procede estudiar por qué ocurre esto.

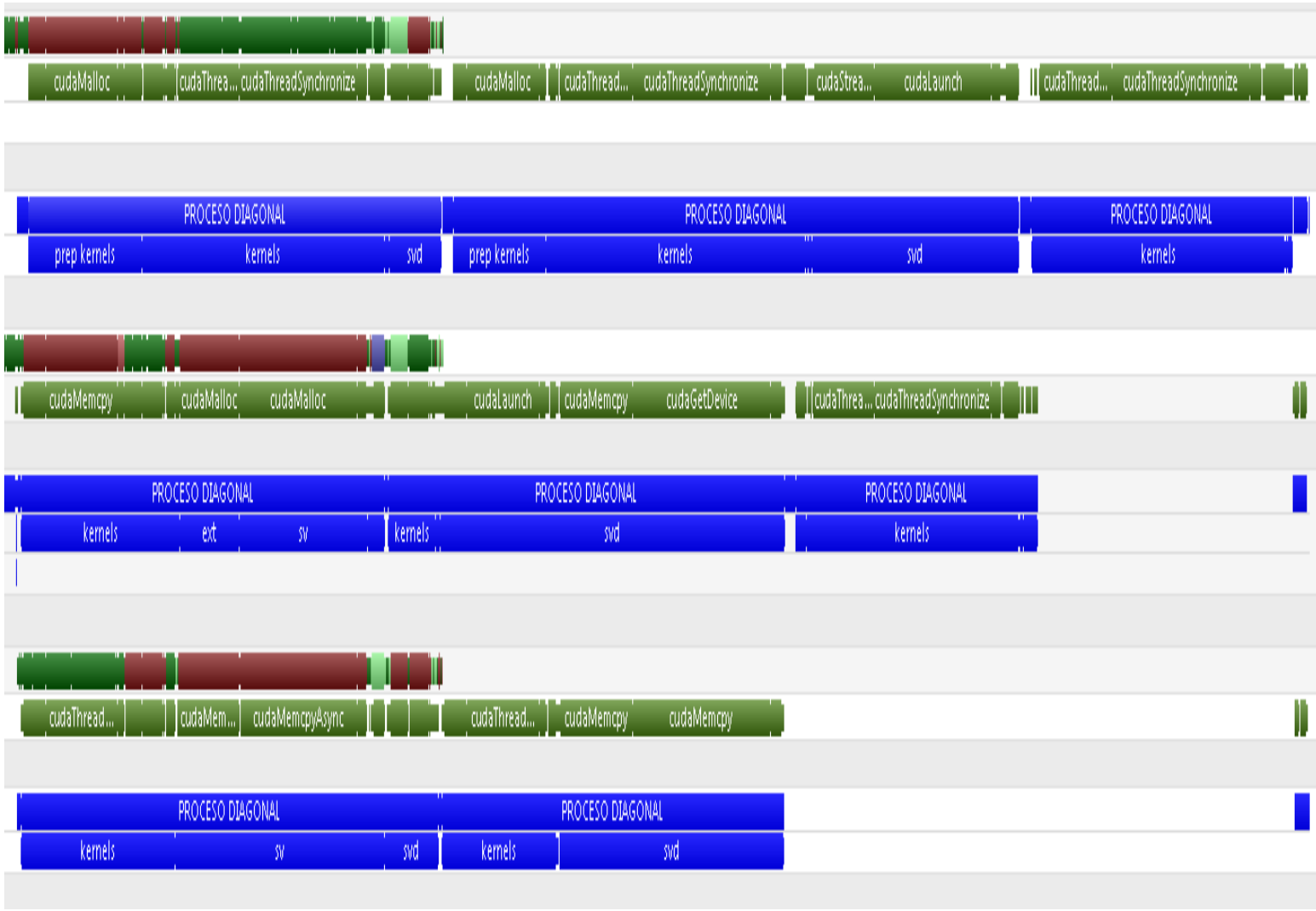


Figura 7-27. Cronograma de ejecución: 3 hilos y 3 GPUs.

Figura 7-28. Cronograma de ejecución: 6 hilos y 3 GPUs.

Una de las utilidades que incluye NVIDIA con su paquete de software es un analizador de la ejecución de programas. Aplicado a nuestra aplicación, nos proporciona de forma gráfica la información necesaria para ver cómo se ejecutan las partes de la misma. La Figura 7-28 corresponde al análisis de la ejecución de la aplicación con la geometría dividida en 8 bloques sobre 3 GPUs con 3 hilos de CPU. Se ha recortado la parte que corresponde al proceso de las submatrices en la diagonal, que es la que se muestra.

Las dobles barras horizontales azules corresponden a los *kernels* que procesan las submatrices en la diagonal, y se pueden distinguir las 8 que hay. Se puede ver que se pierde efectividad en el hecho de que la última de las barras (la inferior de las tres) está sin hacer nada, a pesar de que la primera de todas aún no ha terminado con su segunda submatriz. Esto es consecuencia de la asignación de submatrices a hilos: OpenMP asigna homogéneamente las iteraciones de los bucles que paraleliza a los hilos de que dispone, y cuando no son divisibles unas entre otros, el resto lo asigna por orden. En este caso, dado que cada submatriz es una iteración de un bucle, se asignan 2 iteraciones a cada uno de los 3 hilos disponibles. Los dos que sobran se asignan al primero y al segundo respectivamente. Como consecuencia de esto, si las iteraciones no tienen una carga de trabajo parecida, se pueden dar situaciones como esta.

La Figura 7-29 muestra la misma situación, pero con dos hilos por cada GPU. El reparto se hace igual: se asignan iteraciones a hilos, y los que sobran, por orden. Pero, al haber seis hilos, a cada uno le toca una submatriz, y a los dos primeros una adicional. La consecuencia es un reparto más equilibrado, lo que justifica que en ocasiones sea mejor utilizar más hilos de CPU. En otras ocasiones, sin embargo,

añadir hilos hace que el reparto de trabajo sea menos homogéneo, y se pierda rendimiento (como ocurre con 4 hilos y 2 GPUs en la mayoría de los casos expuestos).

Además, hay otro factor que tiene una influencia muy importante en la distribución del trabajo: la asignación de hilos de CPU a GPUs. Esta asignación es controlada por el programador, y en nuestro caso, es secuencial: el primer hilo trabaja con la primera GPU, el segundo con la segunda, etc. Cuando se acaban las GPUs, se vuelve a empezar por la primera. En otras palabras, todos los *kernels* del primer hilo se ejecutan en la primera GPU, todos los del segundo en la segunda, etc. Si hay 6 hilos, a la primera GPU le corresponden los *kernels* de los hilos 1 y 4, a la segunda los de los hilos 2 y 5 y a la tercera los de los hilos 3 y 6. El reparto final de trabajo es la consecuencia de estas dos correspondencias: iteraciones a hilos de CPU, e hilos de CPU a GPUs.

La tabla Tabla 7-13 muestra cómo se asigna trabajo a las GPUs. Se ha añadido el número de subdominios que contiene cada uno de los bloques para poder obtener una estimación del trabajo de cada GPU. Además, hilos y GPUs se han numerado desde 0.

Tabla 7-13. Asignación de bloques a hilos y GPUs

	3 hilos CPU, 3 GPUs		6 hilos CPU, 3 GPUs	
Nº de SDs	**hilo**	**GPU**	**hilo**	**GPU**
588	0	0	0	0
598	0	0	0	0
1186	0	0	1	1
1202	1	1	1	1
949	1	1	2	2
963	1	1	3	0
420	2	2	4	1
420	2	2	5	2

En el caso en el que se utilizan 3 hilos de CPU se tiene que la GPU 0 procesa un total de 2372 subdominios. En realidad este es un valor estimativo, porque habría que añadir los que forman la franja, y el número de éstos depende de cuántos subdominios cercanos tiene cada bloque, pero como valor orientativo, es útil. La GPU 1 procesa 3114 y la 2, 840. Como se ve, el reparto de trabajo no es muy equilibrado. Sin embargo, con el segundo esquema, la GPU 0 procesa 2149 subdominios, la 1, 2808 y la 2, 1369. Esto es bastante más homogéneo que el inicial, y explica por qué, en este caso, es más conveniente usar más hilos.

Sin embargo, como se ha visto, hay casos en los que es perjudicial, lo que da lugar a la necesidad de la Versión 2 de OpenMP. En esta, la asignación de iteraciones a hilos no es estática, sino que se va haciendo a medida que los hilos terminan el trabajo que se les asigna.

7.6.2 Versión 2.

La Versión 2 de OpenMP utiliza la planificación 'guiada' (*guided*) que proporciona OpenMP. De esta manera, el trabajo se reparte de forma homogénea entre los hilos, y, sobre todo, el reparto se hace de forma dinámica. La programación es más sencilla, y los resultados mejores.

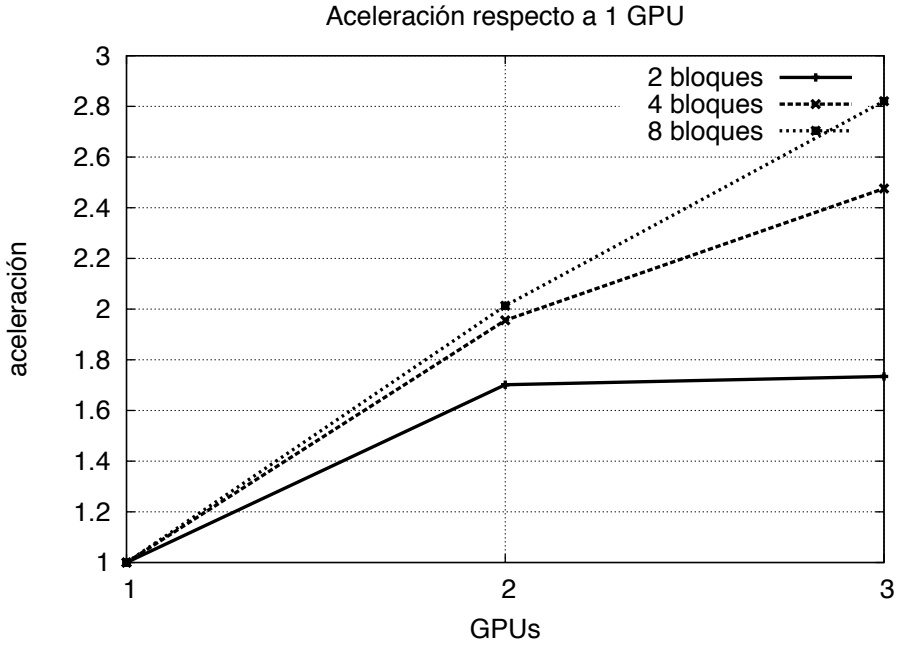


Figura 7-29. Comparación con 1 GPU

La Figura 7-29 muestra el resultado obtenido con la Versión 2. Las gráficas corresponden a la aceleración al procesar las geometrías divididas en 2, 4 y 8 bloques con 2 y 3 GPUs. Como se puede ver, con 2 GPUs se obtienen valores muy cercanos a 2, lo que sería el óptimo. Pero la mejora fundamental de esta versión frente a la Versión 1 se produce para el caso en el que se usan 3 GPUs. Salvo para 2 bloques, donde no es posible explotar la tercera GPU, para el resto de casos la aceleración es mucho mayor, e incluso se acerca

mucho a 3 en el caso de los 8 bloques. Esto es razonable, ya que, cuantos más bloques, más fácil es repartir bien el trabajo entre las CPUs.

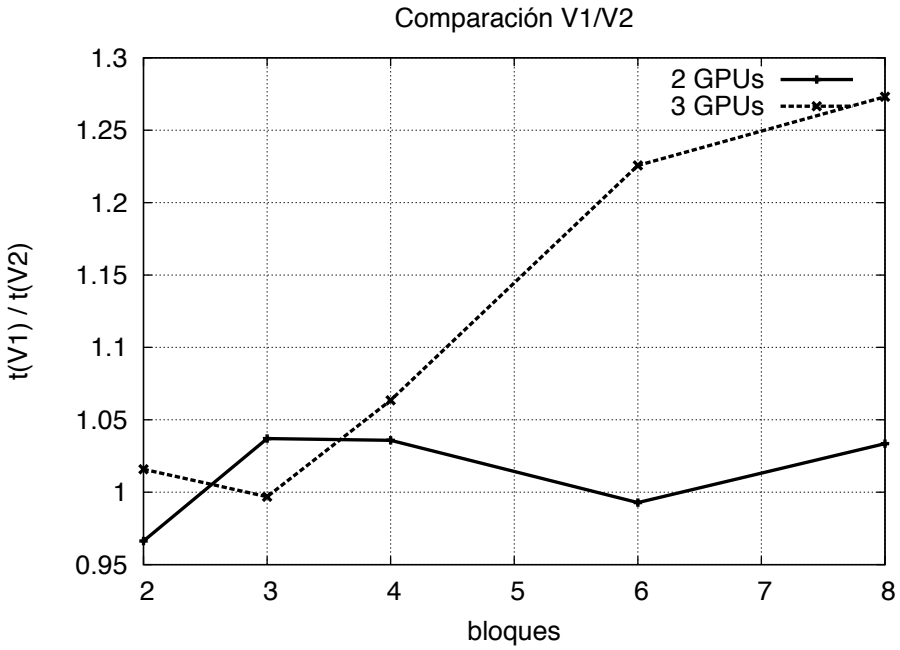


Figura 7-30. Comparación entre las versiones 1 y 2 de OpenMP

La Figura 7-30 muestra el resultado de comparar las versiones 1 y 2 de OpenMP. Para cada combinación de número de bloques y número de GPUs se calcula el cociente entre el mejor tiempo obtenido por la Versión 1 para ese número de GPUs (es decir, el mejor entre utilizar 1 hilo por GPU y utilizar 2 hilos) y el tiempo obtenido por la Versión 2. Como se ve en la gráfica, para 2 GPUs ambas versiones dan valores parecidos (valores cercanos a 1), aunque la Versión 2 es superior en general. La mayor diferencia se da para 3 GPUs, que era una situación en la que la Versión 1 daba un rendimiento malo. La Versión 2 es netamente superior en este caso, merced a su flexibilidad en el reparto de la carga. Para comparar, se muestran en la Tabla 7-14 los tiempos de ejecución de la versión 2 para 2 y 3 GPUs. Dado que los hilos se reparten el trabajo dinámicamente, no es interesante crear 2 por cada GPU. Se incluyen en la tabla los tiempos de proceso

de las submatrices en la diagonal, del resto de submatrices, y del total. Si se compara con la Tabla 7-12 se puede observar cómo, especialmente para el uso de 3 GPUs, la mejora es notable.

Tabla 7-14. Tiempos de ejecución de la Versión 2

		Bloques				
GPUs/Hilos		**2**	**3**	**4**	**6**	**8**
2/2	Diagonal	26.07	22.04	21.75	20.22	19.77
	No Diagonal	13.90	18.58	19.84	22.87	23.73
	Total	40.22	40.83	41.76	43.27	43.70
3/3	Diagonal	25.40	21.73	18.44	15.65	15.05
	No Diagonal	13.85	13.00	14.37	15.41	15.93
	Total	39.45	34.91	32.99	31.25	31.19

Como conclusión de este apartado se puede sacar que, al usar varias GPUs es muy importante el reparto equilibrado de trabajo entre las mismas. Si las herramientas de programación permiten hacer un reparto adecuado, es muy conveniente hacerlo, y si no, merece la pena la inversión en tiempo y trabajo en programarlo manualmente, ya que un resultado satisfactorio puede depender de ello. En el caso de este trabajo, se ha conseguido pasar de un rendimiento discreto con 3 GPUs a una utilización casi completa.

8 Conclusiones, aportaciones y trabajo futuro.

8.1 Conclusiones.

En la presente tesis doctoral se ha llevado a cabo la transformación del código que aplica el Método de los Momentos y el Método de las Funciones Base Características desde su versión original secuencial en FORTRAN a una versión mixta FORTRAN-C adaptada para ejecutarse en GPUs de NVIDIA por medio de la plataforma de computación paralela CUDA. El resultado general ha sido más que satisfactorio, con mejoras del rendimiento por encima de 40 para el MoM y de 30 para el CBFM en los casos mejores utilizando una única GPU. Cuando se utilizan varias, también en el caso mejor, estas mejoras se multiplican por un factor que llega a 2.8 para el caso de 3 GPUs. Todas estas mejoras se han obtenido sin menoscabo de la precisión de los resultados. Como resultado del trabajo realizado se han podido obtener una serie de conclusiones que se detallan en los siguientes subapartados.

8.1.1 Corrección de los resultados.

Todas las tarjetas GPU utilizadas durante el desarrollo de este trabajo permiten trabajar con datos de precisión simple. Algunas de ellas, las más potentes, dado que están diseñadas para el cálculo numérico, también admiten trabajar con datos de doble precisión. Sin embargo, los métodos que se han trabajado en esta tesis no requieren tanta precisión, por lo que no ha sido necesario utilizarla. Todos los

cálculos se han hecho con datos en precisión simple, que es la que utiliza la aplicación original. Como era de esperar, dado que los métodos utilizados son los mismos, los resultados obtenidos de la versión CUDA coinciden de forma completa con los proporcionados por la aplicación original. De esta manera, las tarjetas GPU pueden utilizarse con la garantía de proporcionar la misma precisión que las CPUs para el cálculo científico.

8.1.2 Rendimiento.

Las GPUs son capaces de proporcionar una enorme mejora en el rendimiento de las aplicaciones de cálculo científico. Para que esta mejora sea la mayor posible es preciso cuidar una serie de puntos muy importantes, que se detallan a continuación:

a) *Paralelismo de la aplicación*: dada la estructura de una GPU, con una gran cantidad de unidades de ejecución que pueden trabajar en paralelo, la aplicación debe tener un gran paralelismo de datos. Las tareas paralelas no tienen por qué ser muy largas, pero sí deben aplicarse a muchos datos. En caso contrario, las GPUs no aportarán una gran mejora en el rendimiento.

   En el caso de esta tesis, el Método de los Momentos sí muestra un gran contenido en paralelismo de datos, de ahí la espectacular mejora que muestra. El Método de las Funciones Base Características, con un menor grado de paralelismo, no sale tan beneficiada del uso de la GPU.

b) *Expresión del paralelismo*: En cualquier caso, que la aplicación tenga paralelismo no es suficiente: la forma de expresar los algoritmos y de organizar el trabajo de la aplicación en la GPU

debe permitir su aprovechamiento. En nuestro caso, ha sido necesario disponer las tareas paralelas (fundamentalmente, las integrales de superficie) en unas construcciones lógicas llamadas superbloques, que no tienen ningún significado en el algoritmo, pero que permiten a la GPU ejecutarlas en paralelo en mayor medida. Siguiendo en esta línea, se puede concluir que los bloques de hilos de la GPU deben ser aprovechados completamente (o lo máximo posible), es decir, deben construirse con el mayor número de hilos posible, dentro de la lógica de la aplicación. No hacerlo impide a la GPU enmascarar los tiempos de espera de operaciones largas (típicamente, accesos a memoria) con otras operaciones de otros hilos disponibles.

c) *Homogeneidad del trabajo de los hilos*: Un factor muy importante en el rendimiento de la aplicación es lo parecido que sea el trabajo que realizan los distintos hilos que forman un *kernel*. Cuanto más parecido, mejor rendimiento. Divergencias en la secuencia de instrucciones ejecutadas lleva a la secuencialización de la ejecución, como ocurría en nuestro caso en las primeras versiones del Método de los Momentos. El rendimiento mejoró cuando se dividió el *kernel* que calcula las integrales de superficie en tantos diferentes como números de puntos de integración, que es lo que determinaba la secuencia de instrucciones de los hilos del *kernel*. El ideal es un *kernel* que no ejecuta saltos incondicionales, o, si los ejecuta, aquel en el que todos los hilos saltan de la misma manera.

d) *Número de kernels*: cada lanzamiento de un *kernel* conlleva un retardo: la GPU debe prepararse para lanzarlo, recibir el

código, organizar los recursos, etc. Generalmente, el número de *kernels* en los que se descompone una aplicación viene definido por la secuencia lógica de la misma, pero, si es posible, es aconsejable reducirlo. En nuestro caso, ésto no es posible hacerlo sin pérdida de rendimiento. El problema con nuestra aplicación surge como consecuencia de la necesidad de sincronización cuando se funden los *kernels*. Inicialmente, se implementaron las integrales de superficie en un *kernel* (uno por cada tipo de integral, en realidad), y las de línea en otro. Cuando la funcionalidad se combinó en un único *kernel*, la sincronización necesaria redujo el rendimiento. De no haber sido necesaria, habría sido mejor.

Esta conclusión puede parecer incompatible con la anterior, ya que en aquella se propone aumentar el número de *kernels* y aquí reducirlo. En realidad, lo que se propone aquí es combinar las tareas de dos o más *kernels consecutivos*, donde cada uno *continúa* el trabajo del anterior, evitando así incurrir en el gasto de lanzar cada nuevo *kernel*. En aquel caso, los hilos que se separan ejecutan tareas concurrentes (el mismo paso del algoritmo para diferentes datos), y son independientes entre sí. Lo ideal sería que esas tareas fueran idénticas, para usar un único *kernel*. Pero si son demasiado diferentes, la pérdida de tiempo que se evita separando los *kernels* compensa el recargo de lanzar varios de ellos.

Cuando varios *kernels* que aplican pasos sucesivos de un algoritmo se funden en un único *kernel*, las ventajas son no sólo un menor tiempo de lanzamiento, ya que se lanza un único *kernel*, sino también una reducción en la cantidad de

memoria intermedia requerida para trasladar los datos de un *kernel* al siguiente. Esta segunda ventaja puede permitir que el problema tratado sea mayor, y nuestra aplicación sí se puede beneficiar de ella.

e) *Aprovechamiento de los recursos de la GPU:* la memoria compartida, los registros de cada multiprocesador, la memoria de constantes, son recursos que están disponibles para el programador y que pueden (y deben) ser utilizados para mejorar el rendimiento de la aplicación. En nuestro caso, se utiliza la memoria constante para contener los punteros a los datos de la geometría, y la memoria compartida para que los hilos que colaboran en las distintas integrales de superficie compartan información de forma rápida y eficiente, en lugar de hacerlo por medio de la más lenta memoria global. Los registros no son accesibles al programador directamente, pero el compilador realiza una asignación de variables a los mismos de forma probablemente más eficaz que el propio programador, por lo que no hay que preocuparse demasiado en esto. En cualquier caso, y si se prefiere gestionar los registros de forma manual, las versiones más recientes de CUDA permiten utilizar un lenguaje ensamblador intermedio (PTXAS, *Parallel Thread eXecution ASsembly*) con el que se puede acceder a los registros directamente.

8.1.3 Utilización de varias GPUs.

Si una aplicación se beneficia del uso de una GPU, lo más probable es que se pueda beneficiar aún más si utiliza varias. La transformación de la aplicación para que pueda hacerlo puede

resultar compleja, si se hace de forma manual, pero existen herramientas *software* que facilitan mucho esta labor. En esta tesis se ha optado por utilizar el entorno OpenMP para hacer constar a la aplicación de hilos de CPU que se encarguen de controlar cada GPU y la alimenten con trabajo y datos. Las conclusiones obtenidas del estudio de estas posibilidades giran en torno al reparto de trabajo de la aplicación a los hilos de CPU, ya que la organización del trabajo en cada GPU es una simple réplica del caso en el que sólo hay una.

a) *Reparto de trabajo en la CPU*: En primer lugar, es importante elegir la estrategia adecuada para asignar trabajo a los hilos de CPU. En general (y en el caso de esta aplicación), si la estrategia es dinámica se puede adaptar a las condiciones que se presenten en el momento de la ejecución. Las cargas asignadas (a hilos de CPU, en nuestro caso) se pueden equilibrar para mantener a todas las GPUs ocupadas el mayor tiempo posible.

   En nuestro caso, dado que cada hilo controla una GPU, el reparto de trabajo de los hilos repercute directamente en el aprovechamiento que se hace de ellas. De esta manera, si la estrategia de reparto va dando trabajo a los hilos a medida que van terminando sus asignaciones anteriores, en lugar de hacer un reparto fijo *a priori*, se evitan los hilos y las GPUs ociosos. El resultado final es una mejora casi lineal del rendimiento con el número de GPUs.

b) *Número de hilos por cada GPU*: Una posibilidad para mantener a las GPUs ocupadas continuamente es crear más de un hilo de CPU por cada una de ellas. La teoría es que mientras la GPU está ocupada procesando el trabajo que le da

un hilo, otro u otros hilos que usan esa misma GPU están preparando más datos para transmitirle en cuanto termine. Esto es aconsejable siempre que la proporción de tiempo entre la preparación de los datos en la CPU y su posterior procesamiento en la GPU sea alta. Ese no es el caso de nuestra aplicación, por lo que esta estrategia no es necesaria. Además, si se opta por ella, es preciso tener en cuenta los requisitos, fundamentalmente de memoria, de cada hilo de CPU, porque puede ocurrir que la suma los requisitos de todos los hilos creados supere los recursos presentes en la máquina.

8.2 Aportaciones.

La presente tesis ha proporcionado resultados interesantes en varios aspectos. En primer lugar, se ha obtenido una mejora de rendimiento notable al acelerar tanto el Método de los Momentos (MoM) como el Método de las Funciones Base Características (CBFM). Es conocida la capacidad de CUDA en combinación con las GPU de NVIDIA de acelerar la ejecución de un gran número de aplicaciones, pero hasta el momento, la bibliografía no mostraba resultados sobre la aceleración del CBFM, y sólo algunos trataban el MoM. Con este trabajo se ha pretendido mostrar una forma nueva de atacar la paralelización del MoM y, sobre todo, hacer ver que el CBFM también se puede beneficiar de CUDA.

El proceso de adaptación a CUDA se ha encontrado con una serie de dificultades que han ido siendo superadas. La aplicación, como la mayoría de las de cálculo científico, está escrita en FORTRAN. En el momento del inicio de los trabajos no existía en el mercado ningún compilador consolidado que ofreciera las características sí ofrece el

compilador de C de NVIDIA. La consecuencia principal habría sido una mejora muy reducida en el rendimiento, lo que desaconsejó ese camino. La forma de sortear esta dificultad fue reescribir parte de las rutinas de la aplicación, en concreto aquellas cuya ejecución debía trasladarse a la GPU, en C, dejando el resto de la aplicación en FORTRAN. Una traducción completa de la aplicación a C suponía una tarea inabordable, dada su complejidad, por lo que fue necesario proceder de esta manera. Sin embargo, este tipo de aplicación mixta no estaba muy estudiada en el momento de abordar el trabajo. De hecho, la documentación de NVIDIA ni la contemplaba, por lo que fue necesario crear soluciones '*ad-hoc*' para desarrollarla. El objetivo era poder utilizar las herramientas de desarrollo de NVIDIA, especialmente el depurador visual (NVIDIA Nsight) y el analizador de rendimiento, por lo que fue necesario, de alguna manera, "engañar" a las herramientas. Se creó una aplicación global en C, que llamaba a la aplicación en FORTRAN, que, a su vez, utilizaba las rutinas en C que trabajaban con la GPU. De esta manera, a los ojos de las herramientas de depuración, se trataba de una aplicación C, con lo que todo funcionaba de forma correcta.

Otro de los objetivos de la tesis era estudiar con qué facilidad se pueden utilizar varias GPUs para acelerar la aplicación. En este sentido, los resultados son más que satisfactorios: lo único que se necesita es que los compiladores admitan las directivas de OpenMP. Esta tecnología hace realmente fácil dividir el trabajo de la aplicación en "paquetes" y asignárselos a las GPUs. La única cuestión que hay que cuidar es el criterio de asignación de trabajos, para que sea repartido de forma homogénea, que es lo que permite obtener ganancias de rendimiento casi lineales con el número de GPUs.

Todas estas aportaciones se han expuesto a la comunidad científica en forma de publicaciones que se incluyen a continuación:

- J.I. Pérez, E. García, J.A. de Frutos, J.R. Almagro, F. Cátedra. *Application of GPU Computing to the Characteristic Basis Function Method*. Presentado en la conferencia: European Conference on Antennas and Propagation (EUCAP 2012).
- E. García, J.I. Pérez, J.A. de Frutos, F. Cátedra, R. Mitra. *Parallelization Strategies for the Characteristic Basis Function Method*. Capítulo del libro: *Computational Electromagnetics: Recent Advances and Engineering Applications*. Pp. 41-71. Springer Science, 2014. ISBN 978-1-4614-4381-0.
- J.I. Pérez, E. García, J.A. de Frutos, F. Cátedra. *Application of the Characteristic Basis Function Method using CUDA*. Publicado en la revista: International Journals of Antennas and Propagation. Vol. 2014, ID: 721580.

8.3 Trabajo futuro.

Toda tesis doctoral es un trabajo necesariamente acotado en el tiempo, pero existen una serie de posibles líneas de trabajo que surgen por el mero hecho de haber resuelto con éxito los problemas que ocasionaron el planteamiento de ésta. Son posibles optimizaciones o mejoras que se conciben una vez terminado cualquier trabajo de optimización, como los que se han llevado acabo aquí.

Además, los objetivos alcanzados en esta tesis permiten, de forma casi automática, plantearse otros que constituyen su continuación lógica. Incluso es posible también extrapolar lo conseguido aquí a otros campos, tal vez algo más lejanos, pero unidos a los aquí trabajados por su organización lógica o su metodología común.

8.3.1 Mejoras adicionales para acelerar aún más la aplicación desarrollada.

Dentro este grupo se pueden considerar:

- *Estudio de formas de organización más eficientes de los datos en la memoria de la GPU.* El acceso a los datos en la GPU es uno de los factores que determinan el rendimiento de forma más directa. La aplicación desarrollada en esta tesis utiliza unas estructuras de datos complejas, compuestas por arrays de varias dimensiones. La organización de estas estructuras de datos en memoria para un acceso lo más rápido posible no es un problema trivial, y requiere un estudio minucioso de la secuencia de accesos y las posiciones de memoria implicadas. Una posibilidad es modificar el código de la aplicación para ordenar los accesos, o permitir, al menos, una distribución adecuada de los datos en la memoria.

  También resulta aconsejable considerar la utilización de las memorias rápidas presentes en la GPU, la memoria compartida y los registros, para los datos que se utilicen más frecuentemente. En esta tesis se ha realizado parte de este estudio, utilizando la memoria compartida para algunos datos pertenecientes al cálculo de las integrales de superficie, pero es muy probable que su uso se pueda extender a toda la aplicación. Además, las nuevas GPUs disponen de nuevas tecnologías de memorias caché, que son menos

sensibles a la ordenación estricta de los datos y más a la localidad de los accesos, lo cual abre un nuevo abanico de opciones de mejora.

- *Adaptación a problemas mayores.* La memoria disponible en la GPU determina el tamaño máximo de la geometría que se puede utilizar. Es cierto que la cantidad disponible de esta memoria va aumentando a medida que aparecen nuevas tarjetas, pero aún así sería conveniente estudiar formas de reducir la cantidad de memoria que se utiliza en los algoritmos que componen la aplicación. De esta manera, las geometrías con las que se podría trabajar serían mayores.
- *Optimizaciones "a medida".* La parte de la aplicación que implementa el Método de las Funciones Base Características no se beneficia en la misma medida del uso de GPUs que la parte que aplica el Método de los Momentos. En ella se aplican una serie de operaciones de álgebra lineal, como la Descomposición en Valores Singulares (SVD) que tal vez se podrían optimizar si se crean funciones específicas. Las empresas que crean las librerías que implementan estas operaciones ofrecen un amplio conjunto de las mismas, lo cual puede llevar a que trabajen optimizaciones globales para todas ellas. Es posible que un estudio específico de las utilizadas en esta aplicación diera como resultado optimizaciones que, aunque no fueran aplicadas al conjunto de las librerías, sí mejoraran el CBFM de forma aislada.

8.3.2 Utilización de otras tecnologías y aplicación a otros problemas científicos.

Dado el éxito logrado con la utilización de una o varias GPUs, es posible plantearse dos líneas nuevas de trabajo: por un lado utilizar máquinas que dispongan de grandes cantidades de GPUs. Por otro lado, aplicar el método a otros problemas científicos diferentes pero parecidos en cuanto a los algoritmos que se utilizan en ellos.

- *Máquinas con grandes cantidades de GPUs.* Dentro de la primera línea, existen ya en el mercado computadores escalables compuestos por varias placas base, cada una de ellas con una o varias CPUs y una o varias GPUs. Las configuraciones son variadas, y un primer paso sería estudiar cuál de ellas se adapta mejor al problema. Estas máquinas son altamente escalables, y es necesaria una investigación minuciosa para averiguar la forma más rentable (en el aspecto económico y en el de rendimiento) de repartir el trabajo entre ellas. Las distintas placas base se comunican por medio conexiones de red muy rápidas, pero el volúmen de datos que utilizan estas aplicaciones pronto pueden saturarlas, por lo que se plantea el problema del reparto de datos. Para gestionar máquinas de este tipo puede ser más interesante utilizar otro tipo de mecanismos de comunicación entre procesos, ya que OpenMP está muy íntimamente ligado a los entornos multiprocesador de memoria compartida. Probablemente sería más indicado utilizar el paradigma MPI, más estudiado en dispositivos de este tipo.

  Otra posibilidad, dentro de la misma línea de trabajo, son los los llamados '*clusters*' de computadores. En este caso los sistemas están formados por computadores personales sin características

especiales, conectados entre sí para resolver el problema. El rendimiento obtenido será seguramente menor, pero también lo será el coste. De nuevo sería necesario un estudio de las posibilidades que ofrece el sistema, y cómo adaptar la aplicación a él. Por último, una tercera vía serían todas las posibles soluciones intermedias, entre los sistemas basados en múltiples placas base con GPUs incorporadas y los *clusters* de computadores sencillos.

- *Adaptación a CUDA de otros métodos de análisis electromagnético.* Como segunda vía de trabajo, dentro del análisis electromagnético existen multitud de métodos que se podrían beneficiar de máquinas dotadas de GPUs. Algunos, relacionados con los utilizados en esta tesis, como el Método Rápido de los Multipolos (FMM, *Fast Multipole Method*), que se puede combinar con técnicas multinivel (MFMM, *Multilevel Fast Multipole Method*), tienen muchas cosas en común con el utilizado en esta tesis, por lo que se presta de forma natural a aplicar las mismas técnicas.

Bibliografía

[1] G. A. Deschamps, "Ray Techniques in Electromagnetics," *Proceedings of the IEEE*, vol. 62, no. 9, pp. 1022-1035, 1962.

[2] J. S. Asvestas, "The Physical Optics Method in Electromagnetic Scattering," *Journal of Mathematical Physics*, vol. 21, no. 2, pp. 290-299, 1980.

[3] J. B. Keller, "Geometrical Theory of Diffraction," *Journal of the Optical Society of America*, vol. 52, no. 2, pp. 116-130, 1962.

[4] E. F. Knott, "A Progression of High-Frequency RCS Prediction Techniques," vol. 73, no. 2, pp. 252-264, 1985.

[5] R. F. Harrington, "The Method of Moments in Electromagnetics," *Journal of Electromagnetic Waves and Applications*, vol. 1, no. 3, pp. 181-200, 1987.

[6] M. F. Cátedra, "Solution to some electromagnetic problems using Fast Fourier Transform with Conjugate Gradient Method," *Electronics Letters*, vol. 22, pp. 1049-1051, 1986.

[7] M. A. Morgan, "Principles of Finite Methods in Electromagnetics Scattering," in *Finite Element and Finite Difference Methods in Electromagnetic Scattering.*: Elsevier, 1990.

[8] A. Taflove and K. R. Umashankar, "The Finite-Difference Time-Domain Method for Numerical Modelling of Electromagnetic Wave Interactions with Arbitrary Structures," in *Finite Element and Finite Difference Methods in Electromagnetic Scattering.*: Elsevier, 1990.

[9] N. Engheta, W. D. Murphy, V. Rokhlin, and M. S. Vassilou, "The Fast Multipole Method (FMM) for electromagnetic

scattering problems," *IEEE Transactions on Antennas and Propagation*, vol. 40, no. 6, 1992.

[10] Weng Cho Chew and Cai-Cheng Lu, "A multilevel N log N algorithm for solving boundary integral equation," in *International Symposium of the Antennas and Propagation Society*, vol. 1, 1994, pp. 431-434.

[11] V.V.S. Prakash and R. Mittra, "Characteristic basis function method: A new technique for efficient solution of method of moments matrix equation," *Microwave and Optical Technology Letters*, vol. 36, no. 2, pp. 95-100, enero 2003.

[12] I. González Diego, A. Tayebi Tayebi, J. Gómez Pérez, E. García García, and M. F. Cátedra Pérez, "New Moment Method Tool Combining FMLMP, CBF and MPI," in *Antennas and Propagation Society International Symposium*, 2009.

[13] T. Van Luong, N. Melab, and E. G. Talbi, "GPU Computing for Parallel Local Search Metaheuristic Algorithms," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 173-185, 2013.

[14] K. H. Lee and S. W. Heo, "GPU based Software DVB-T receiver design," in *IEEE International Conference on Consumer Electronics*, 2013, pp. 582-585.

[15] R. DeLeon, D. Jacobsen, and I. Senocak, "Large-Eddy Simulations of Turbulent Incompressible Flows on GPU Clusters," *Computing in Science & Engineering*, vol. 15, no. 1, pp. 26-33, 2013.

[16] D.J. Ludick and D.B. Davidson, "Investigating Efficient Parallelization Techniques for the Characteristic Basis Function

Method (CBFM)," in *International Conference on Electromagnetics in Advanced Applications (ICEAA 09)*, 2009.

[17] E. García, L. Lozano, M.J. Algar, and F Cátedra, "A study of the efficiency of the parallelization of a high frequency approach for the computation of radiation and scattering considering multiple bounces," *Computer Physics Communications*, vol. 184, no. 1, pp. 45-50, 2013.

[18] E. Lezar and D.B. Davidson, "GPU Acceleration of Method of Moments Matrix Assembly using Rao-Wilton-Glisson Basis Functions," in *International Conference on Electronics and Information Engineering (ICEIE)*, 2010.

[19] S. P. Walker and C. Y. Leung, "Parallel Computation of Time-Domain Integral Equation Analyses of Electromagnetic Scattering and RCS," *IEEE Transactions on Antennas and Propagation*, vol. 45, no. 4, pp. 614-619, Abril 1997.

[20] Pascal Havé, "A parallel implementation of the fast multipole method for Maxwell's equations," *International Journal for Numerical Methods in Fluids*, no. 43, pp. 839-864, 2003.

[21] Fang Wu, Yaojiang Zhang, Zaw Zaw Oo, and Erping Li, "Parallel multilevel fast multipole method for solving large-scale problems," *IEEE Antennas and Propagation Magazine*, vol. 47, no. 4, pp. 110-118, agosto 2005.

[22] D. Jiang et al., "Parallel Implementation of the Steepest Descent Fast Multipole Method (SDFMM) on a Beowulf Cluster for Subsurface Sensing Applications," *IEEE Microwave and Wireless Components Letters*, vol. 12, no. 1, pp. 24-26, enero 2002.

[23] Tom Cwik, Jonathan Partee, and Jean Patterson, "Method of Moment Solutions to Scattering Problems in a Parallel Processing Environment," *IEEE Transactions on Magnetics*, vol. 27, no. 5, pp. 3837-3840, septiembre 1991.

[24] B. A. Zimmermann and G. A. Crichton, "A programming model for the Mark III hypercube with multiple processor nodes," in *Proceedings on the third Conference on Hypercube concurrent computers and applications: Architecture, software, computer systems and general issues - Volume 1*, 1988, pp. 528-535.

[25] Dimitra I. Kaklamani, Konstantina S. Nikita, and Andy Marsh, "Extension of Method of Moments for Electrically Large Structures Based on Parallel Computations," *IEEE Transactions on Antennas and Propagation*, vol. 45, no. 3, pp. 566-568, marzo 1997.

[26] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, and Mathieu Faverge, "QR Factorization on a Multicore Node Enhanced with Multile GPU Accelerators," in *IEEE International Symposium on Parallel & Distributed Processing*, 2011, pp. 932-943.

[27] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R. Gao, "Dynamic Load Balancing on Single- and Multi-GPU Systems," in *IEEE International Symposium on Parallel & Distributed Processing*, 2010.

[28] Dana Schaa and David Kaeli, "Exploring the Multiple-GPU Design Space," in *IEEE International Symposium on Parallel & Distributed Processing*, 2009.

[29] Jiri Jaros, Bradley E. Treeby, and Alistair P. Rendell, "Use of

Multiple GPUs on Shared Memory Multiprocessors for Ultrasound Propagation Simulations," in *Proceedings of the Tenth Australasian Symposium on Parallel and Distributed Computing*, 2012, pp. 43-52.

[30] Joao V. F. Lima, Thierry Gaultier, Nicolas Maillard, and Vincent Danjean, "Exploiting Concurrent GPU Operations for Efficient Work Stealing on Multi-GPUs," in *International Symposium on Computer Architecture and High Performance Computing*, 2012.

[31] Thierry Gaultier, Joao V.F. Lima, Nicolas Maillard, and Bruno Raffin, "XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures," in *IEEE International Symposium on Parallel & Distributed Processing*, 2013.

[32] S. Potluri et al., "Optimizing MPI Communication on Multi-GPU Systems using CUDA Inter-Process Communication," in *IEEE International Symposium on Parallel & Distributed Processing*, 2012, p. 18481857.

[33] Eric Darve, Cris Cecka, and Toru Takahashi, "The fast multipole method on parallel clusters, multicore processors, and graphics processing units," *Comptes Rendus Mecanique*, no. 339, pp. 185-193, 2011.

[34] Aparna Chandramowlishwaran et al., "Optimizing and Tuning the Fast Multipole Method for State-of-the-Art Multicore Architectures," in *IEEE International Symposium on Parallel and Distributed Processing*, 2010.

[35] Qi Hu, Nail A. Gumerov, and Ramani Duraiswami, "Scalable Fast Multipole Methods on Distributed Heterogeneous

Architectures," in *International Conference for High Performance Computing, Networking and Analysis*, 2011.

[36] T. Hamada et al., "42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence," in *Conference on High Performance Computing, Networking, Storage and Analysis*, 2009.

[37] E. Lezar and D.B. Davidson, "GPU-Accelerated Method of Moments by Example: Monostatic Scattering," *IEEE Transactions on Antennas and Propagation*, vol. 52, no. 6, pp. 120-135, December 2010.

[38] S. Peng and Z. Nie, "Acceleration of the Method of Moments Calculations by Using Graphics Processing Units," *IEEE Transactions on Antennas and Propagation*, vol. 56, no. 7, pp. 2130-2133, July 2008.

[39] E. Lezar and D.B. Davidson, "GPU Acceleration of Electromagnetic Scattering Analysis using the Method of Moments," in *International Conference on Electromagnetics in Advanced Applications*, 2011.

[40] I. Kiss, P.T. Benkö, and S. Gyimóthy, "Fast Analysis of Metallic Antennas by Parallel Moment Method Implemented on CUDA," *International Journal of Applied Electromagnetics and Mechanics*, vol. 39, no. 1-4, pp. 677-683, 2012.

[41] T. Topa, A. Karwowski, and A. Noga, "Using GPU with CUDA to Accelerate MoM-Based Electromagnetic Simulation of Wire-Grid Models," *IEEE Antennas and Wireless Propagation Letters*, vol. 10, pp. 342-345, 2011.

[42] Top500. Top 500 Computer Sites. [Online]. www.top500.org

[43] Top 500. Lista Noviembre 2012. [Online]. http://www.top500.org/lists/2012/11/#.U3siayhXtFs

[44] M. J. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901-1909, 1966.

[45] NVidia Corporation, *NVIDIA CUDA C Programming Guide, v. 4.2.*, 2012.

[46] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of Control Dependence to Data Dependence," in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming*, 1983, pp. 177-189.

[47] NVIDIA Corporation, *CUDA API Reference Manual, v. 4.0.*, 2011.

[48] NVIDIA Corporation, *The CUDA Compiler Driver NVCC.*, 2008.

[49] C. L. Lawson, R. J. Hanson, D Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Transactions on Mathematical Software*, vol. 5, no. 3, pp. 308-323, Septiembre 1979.

[50] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis, "CULA: Hybrid GPU Acceleratd Linear Algebra Routines," in *SPIE Defense and Security Symposium*, 2010.

[51] E Anderson et al., *LAPACK Users' Guide*. Philadelphia: Society for Industrial and Applied Mathematics, 1999.

[52] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips, "A Comprehensie Performance Comparison of CUDA and OpenCL," in *International Conference on Parallel Processing*, 2011.

[53] Kamran Karimi, Neil G Dickson, and Firas Hamze, "A Performance Comparison of CUDA and OpenCL,".

[54] R. F. Harrington, *Field Computation by Moment Methods*. New York: McMillan, 1968.

[55] Fernando Rivas Peña, "Aplicación del método de los momentos para el análisis electromagnético de cuerpos de geometría arbitraria modelados por parches NURBS," Departamento de Ingeniería de Comunicaciones, Universidad de Cantabria, Santander, Tesis doctoral 1994.

[56] Iván González Diego, "Contribución a la Mejora de la GTD en la Predicción de los Sistemas Radiantes sobre Estructuras Complejas," Ciencias de la Computación, Universidad de Alcalá, Alcalá de Henares, Tesis doctoral 2003.

[57] Eliseo García García, "Contribución al análisis de problemas electromagnéticos mediante el Método de los Momentos con bajo coste computacional," Dpto. Ciencias de la Computación, Universidad de Alcalá, Alcalá de Henares, Tesis doctoral 2005.

[58] P. Bezier, *Essay de Définition Numérique des Courbes et des Surfaces Expérimentales.*: Universidad de París IV, 1974.

[59] G. Farin, *Curves and Surfaces for Computer Aided Geometric Design: a Practical Guide.*: Academic Press, 1988.

[60] R. T. Farouki and V. T. Rajan, "On the noumerical condition of polynomials in Bernstein form," *Computer Aided Geometric Design*, vol. 4, no. 3, pp. 191-216, 1987.

[61] L. A. Piegl and W. Tiller, *The NURBS Book.*: Springer, vol. 1997.

[62] M. Cox, "The numerical evaluation of B-Splines," *Journal of the Institute for Mathematics and its Applications*, vol. 10, pp. 134-149, 1972.

[63] W. Boehm, "Rational geometric splines," *Computer Aided Geometric Design*, vol. 4, no. 1-2, pp. 67-77, 1987.

[64] A. W. Glisson and D. R. Wilton, "Simple and efficient numerical methods for problems of electromagnetic radiation and scattering from surfaces," *IEEE Transactions on Antennas and Propagation*, vol. AP-28, no. 5, pp. 593-603, septiembre 1980.

[65] F. Rivas Peña, *Aplicación del método de los momentos para el análisis electromagnético de cuerpos de geometría arbitraria modelados por parches NURBS.*: Universidad de Cantabria, 1994, Tesis Doctoral.

[66] L. Valle, F. Rivas, and M. F. Cátedra, "Combining the Moment Method with Geometrical Modelling by NURBS Surfaces and Bézier Patches," *IEEE Transactions on Antennas and Propagation*, vol. 42, no. 3, pp. 373-381, 1994.

[67] G. Tiberi, M. Degiorgi, A. Monorchio, G. Manara, and R. Mittra, "A Class of Physical Optics-SVD Derived Basis Functions for Solving Electromagnetic Scattering Problems," in *Antennas and Propagation Society International Symposium*, Washington, D.C., 2005.

[68] Eliseo García, Carlos Delgado, Iván González Diego, and Manuel Felipe Cátedra, "An Iterative Solution for Electrically Large Problems Combining the Characteristic Basis Function Method and the Multilevel Fast Multipole Algorithm," *IEEE Transactions on Antennas and Propagation*, vol. 56, no. 8, pp.

2363-2371, agosto 2008.

[69] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in FORTRAN: The Art of Scientific Computing*. Cambridge, Reino Unido: Cambridge University Press, 1992.

[70] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: principles, techniques and tools (2nd. ed.)*.: Addison-Wesley, 2006.

[71] NVIDIA Corporation. (2013) Nvidia Developer Zone. [Online]. http://docs.nvidia.com/cuda/cublas/index.html

[72] B. Chapman, G. Jost, and R. Van der Pas, *Using OpenMP. Portable Shared Memory Parallel Programming.*: The MIT Press, 2007.

[73] S. M. Rao, "Electromagnetic scattering and radiation of arbitrarily-shaped surfaces by triangular patch modelling," Universidad de Mississippi, Tesis doctoral 1980.

[74] J. Yeo, S. Köksoy, V.V.S. Prakash, and R. Mittra, "Efficient generation of method of moments matrices using the characteristic function method," *IEEE Transactions on Antennas and Propagation*, vol. 52, no. 12, pp. 3405-3410, diciembre 2004.

[75] Carlos Delgado Hita, "Desarrollo de técnicas basadas en funciones base características para el análisis de radiación, propagación y dispersión en entornos complejos," Departamento de Ciencias de la Computación, Universidad de Alcalá, Alcalá de Henares, Tesis doctoral 2006.

[76] H. Lin, W. Cai, H. Guo, and X. Liu, "Analysis of Large-Scale EM

Scattering Using the Parallel Characteristic Basis Function Method," in *International Symposium on Antennas, Propagation & EM Theory*, Guilin, 2006, pp. 1-4.

[77] P. Havé, "A parallel implementation of the fast multipole method for Maxwell's equations," *International Journal for Numerical Methods in Fluids*, vol. 43, no. 8, pp. 839-864, 2003.

[78] S. P. Walker and C. Y. Leung, "Parallel Computation of Time-Domain Integral Equation Analyses of Electromagnetic Scattering and RCS," *IEEE Transactions on Antennas and Propagation*, vol. 45, no. 4, pp. 614-619, April 1997.

[79] T. Cwik, J. Partee, and J. Patterson, "Method of Moment Solutions to Scattering Problems in a Parallel Processing Environment," *IEEE Transactions on Magnetics*, vol. 27, no. 5, pp. 3837-3840, September 1991.

[80] Y. Zhang, R. A. van de Geijn, M. C. Taylor, and T. K. Sarkar, "Parallel MoM Using Higher-Order Basis Functions and PLAPACK In-Core and Out-of-Core Solvers for Challenging EM Simulations," *IEEE Antennas and Propagation Magazine*, vol. 51, no. 5, pp. 42-60, October 2009.

[81] D. Jiang et al., "Parallel Implementation of the Steepest Descent Fast Multipole Method (SDFMM) on a Beowulf Cluster for Subsurface Sensing Applications," *IEEE Microwave and Wireless Components Letters*, vol. 12, no. 1, pp. 24-26, January 2002.

[82] M. Swaminathan and T. K. Sarkar, "Parallel Computation of Electromagnetic Scattering from Arbitrary Structures," *IEEE Transactions on Magnetics*, vol. 25, no. 4, pp. 2890-2891, July 1989.

[83] Y. Zhu and X. Lu, "An Optimization of FMM under CPU+GPU Heterogeneous Architecture," in *IEEE International Conference on Commerce and Enterprise Computing*, 2012, pp. 147-150.