Departamento de Informática y Sistemas
Facultad de informática
Universidad de Murcia

Proyecto fin de carrera

Análisis, desarrollo y optimización de rutinas paralelas de cálculo de funciones de Green

Carlos Pérez Alcaraz
2010-2011

Dirigido por:

Domingo Giménez Cánovas
Grupo de computación científica y programación paralela
Departamento de Informática y Sistemas

Fernando Daniel Quesada Pereira
Grupo de electromagnetismo computacional
Departamento de Tecnologías de la Información y las Comunicaciones

Índice general

Abstract

En los últimos años los sistemas de computación han evolucionado hacia arquitecturas basadas en multiprocesadores, que a menudo disponen además de unidades de proceso gráfico que permiten su aprovechamiento para la computación de propósito general.

Esta evolución en el hardware ha traído consigo la aparición de diversos entornos de programación que nos proporcionan modelos de ejecución sobre dichos sistemas a un nivel superior de abstracción, como es el caso de OpenMP, MPI y CUDA.

Este proyecto tiene como objetivo el análisis y aceleración de un conjunto de rutinas utilizadas en diversos problemas de ingeniería mediante la aplicación de técnicas de paralelismo y los modelos de programación anteriores. Además, se da un primer paso en el uso conjunto de más de uno de dichos modelos, permitiendo explotar las capacidades de los sistemas de computación heterogénea disponibles en la actualidad.

En particular, las rutinas sobre las que se ha trabajado se corresponden con un conjunto de las denominadas funciones de Green, herramientas matemáticas que permiten resolver ecuaciones diferenciales no homogéneas bajo ciertas condiciones de contorno. Más concretamente se trata de funciones de Green utilizadas en el campo del electromagnetismo, por ejemplo para el análisis y diseño de filtros.

1. Introducción.

1.1. Introducción.

Desde su aparición en el siglo XVII las ecuaciones diferenciales han sido una herramienta matemática indispensable para el modelado de sistemas, y sus aplicaciones en todos los campos científicos son innumerables. Para la resolución de las mismas existen diversos métodos en función del tipo de ecuación diferencial: separación de variables, método de las series de Fourier, etc [1].

De entre dichos métodos de resolución nos interesan las denominadas funciones de Green [2], las cuales constituyen una herramienta para abordar problemas con ecuaciones diferenciales no homogéneas bajo ciertas condiciones de contorno. Estas funciones reciben su nombre del matemático George Green, que fue el primero que desarrolló la teoría de las mismas alrededor de 1830, y desde entonces han sido ampliamente aplicadas en campos de la física y la ingeniería, como por ejemplo:

- Electromagnetismo: cálculo de potenciales asociados a situaciones gobernadas por la ecuación de Laplace.
- Mecánica: movimiento del oscilador armónico forzado.
- Otros problemas de ingeniería: obtener la respuesta impulso ante una entrada de tipo delta, para conocer la respuesta ante cualquier entrada mediante la operación de convolución.

Este proyecto se centra en las funciones de Green de las guías de ondas [3], las cuales juegan un papel fundamental en el diseño y análisis de circuitos integrados MMIC (Monolithic Microwave Integrated Circuits), cuyo rango de operación se encuentra en las frecuencias de microondas.

Estas funciones tienen expresiones bien conocidas que consisten en series infinitas, bien en el dominio espacial o espectral. Sin embargo, a pesar de su sencillez analítica, presentan algunos inconvenientes desde el punto de vista computacional debido a su lenta convergencia. En el análisis práctico de dispositivos a través de ecuaciones integrales, donde se requiere calcular varios cientos o miles de funciones de Green, esto supone un alto coste computacional y, por tanto, se hacen necesarios métodos que permitan acelerar la convergencia de estas series.

Entre los métodos y técnicas para la aceleración de las funciones de Green se encuentra el método de Ewald, originalmente desarrollado por Paul Peter Ewald, y mediante el cual es posible alcanzar buenos ratios de convergencia empleando un número pequeño de términos de la serie [4]. En concreto, este método consiste en aprovechar el hecho de que las interacciones entre energías electrostáticas consisten en interacciones tanto de rango corto como largo, de forma que se puede reescribir el sumatorio en el espacio real como la suma de dos términos:

$$\varphi(r) \overset{def}{\Longrightarrow} \varphi_{sr}(r) + \varphi_{lr}(r)$$

En la ecuación anterior $\varphi_{sr}(r)$ representa el término de rango corto, que se puede calcular directa y rápidamente en el espacio real, y $\varphi_{lr}(r)$ es el término de rango largo, que se calcula rápidamente en el espacio de Fourier bajo la asunción de que el sistema es infinitamente periódico.

Así, disponemos de métodos numéricos para acelerar el cálculo de estas funciones, pero no de implementaciones eficientes que hagan uso de las capacidades de ejecución paralela que nos brindan los recientes avances en hardware, tanto en los microprocesadores (CPU) como en las unidades de proceso gráfico (GPU). El estudio y análisis de la viabilidad de dichas implementaciones constituyen, por tanto, el punto de partida de este proyecto.

1.2. Estado del arte.

Durante los últimos años los avances en computadores y programación paralela han permitido un gran crecimiento en la investigación científica. Mientras que hasta hace poco la necesidad de realizar cálculos cada vez más grandes y complejos se satisfacía mediante el uso de procesadores con una frecuencia de reloj más elevada, diferentes obstáculos como el *power wall* han impedido continuar con esta tendencia. En su lugar la computación paralela se ha perfilado como un nuevo enfoque para satisfacer esta demanda de cómputo científico, donde la idea es tan simple como la replicación de unidades de proceso para la ejecución simultánea de más de una operación al mismo tiempo.

Las librerías BLAS y LAPACK [5], por ejemplo, utilizadas ampliamente en computación científica de altas prestaciones, proporcionan un conjunto de rutinas de álgebra lineal optimizadas para diferentes arquitecturas. Debido a su interés en este campo, desde hace tiempo se vienen desarrollando versiones paralelas de las mismas. En [6] se presentan algunas técnicas generales para el desarrollo de versiones escalables de BLAS, sin centrarse en ninguna tecnología concreta; en [7] se analizan las posibilidades de aplicar OpenMP exclusivamente para la construcción de rutinas paralelas de BLAS y LAPACK, estudiando los problemas que surgen y posibles soluciones a los mismos. Otros trabajos [8] se centran en evaluar diferentes implementaciones de estas librerías para las principales arquitecturas hardware.

Por otra parte, durante los últimos seis años se ha instaurado un nuevo paradigma de programación paralela basado en las unidades de procesamiento gráfico (GPU). Las mejoras en este tipo de dispositivos han hecho que dejen de ser utilizadas exclusivamente para el procesamiento gráfico, convirtiéndose en sistemas paralelos para computación de uso general (General Purpose GPU) y dando lugar a lo que se conoce como computación GPU [9].

Este nuevo paradigma de programación ha tenido un gran impacto en el campo de la computación científica, que ha visto como un ordenador personal equipado con una GPU puede convertirse en un pequeño supercomputador con varios cientos de unidades de procesamiento. Así, numerosos trabajos han surgido en estos últimos años para proponer algoritmos que aprovechen al máximo estas capacidades, como nos encontramos en [10] o [11].

En el campo del electromagnetismo computacional, dentro del cual se enmarca este proyecto, diferentes problemas han sido objeto de análisis para tratar de proponer algoritmos paralelos eficientes que los resuelvan. En [12] se presenta un método para resolver problemas de dispersión o radiación, diseñado para diferentes arquitecturas hardware y software, utilizando a su vez librerías paralelas de álgebra lineal. También nos encontramos con trabajos para arquitecturas concretas, como el famoso CRAY T3E, donde en [13] se ofrece una implementación paralela eficiente y escalable de las ecuaciones de Maxwell dependientes del tiempo.

Más concretamente, dentro de este campo nos interesan las funciones de Green para la resolución de ecuaciones diferenciales no homogéneas aplicadas a guías de ondas. Aunque existen múltiples trabajos donde se presentan diferentes métodos de cálculo y de aceleración para este tipo de funciones [3, 4, 14, 15, 16], no tenemos referencias de otros artículos donde se diseñen y evalúen técnicas de resolución paralela de las mismas. No obstante sí que nos encontramos diversos trabajos de aplicación de paralelismo a funciones de Green utilizadas en otros problemas diferentes [17, 18].

1.3. Objetivos y metodología.

La motivación de este proyecto es el estudio de diferentes técnicas y herramientas de programación paralela que nos permitan llevar a cabo la aceleración de una aplicación científica basada en un problema real. En concreto, y tal como se comentaba anteriormente, trabajaremos sobre diferentes rutinas de cálculo de funciones de Green. Dentro de las mismas existen funciones en una, dos o tres dimensiones, y a su vez nos encontramos con distintos métodos de cálculo para ellas.

Este proyecto pretende, por tanto, dar un primer paso mediante el estudio del método de cálculo de Ewald comentado anteriormente, comenzando por el análisis de las funciones de una dimensión y tratando de avanzar en el número de dimensiones a considerar, dejando las puertas abiertas a la continuación del mismo mediante el estudio de distintos métodos o ampliando las técnicas estudiadas para el método de Ewald.

La metodología a seguir en el proyecto queda reflejada a continuación:

- Desarrollar una versión en lenguaje C++ de los códigos secuenciales utilizados para la resolución del problema, que nos sirva como punto de partida para la aplicación de las técnicas y herramientas a utilizar durante el proyecto, tratando a su vez de optimizar las mismas.
- Identificar las partes del código susceptibles de ser paralelizadas, en las que la aplicación de las herramientas comentadas en el capítulo 2 suponga una mejora considerable en el tiempo de ejecución del programa.
- Analizar las diferentes posibilidades que nos ofrecen las herramientas utilizadas para alcanzar implementaciones eficientes de las rutinas paralelas desarrolladas, comparando diferentes alternativas en caso de que existan.

- Evaluar las nuevas versiones obtenidas con cada herramienta, comparándolas tanto con la versión secuencial original como con las versiones desarrolladas con el resto de herramientas, a fin de determinar cuál de ellas presenta mejores resultados dependiendo de los parámetros de entrada del problema analizado.
- Combinar el uso de las diferentes técnicas y herramientas haciendo uso de programación híbrida, en entornos heterogéneos donde dispongamos de diferentes unidades de procesamiento de información que disponen de diferentes capacidades de cómputo.
- Presentar conclusiones sobre el trabajo realizado y los resultados obtenidos, además de ofrecer vías futuras de trabajo para la continuación del mismo.

2. Herramientas computacionales.

En los últimos años el mundo de la computación ha experimentado grandes cambios, tanto a nivel software como hardware. En la rama del hardware, los microprocesadores continúan mejorando sus prestaciones año tras año, pero a diferencia de la tendencia anterior basada en el desarrollo de monoprocesadores cada vez más complejos se ha evolucionado hacia diseños basados en multiprocesadores, integrando varios procesadores en un mismo chip en lo que se denominan CMP's (Chip-level multiprocessing).

Esta evolución viene motivada por diversos factores, como la incapacidad para encontrar suficiente paralelismo en un flujo de instrucciones que permita mantener ocupado un monoprocesador de alto rendimiento (el problema del *ILP Wall*) o la dificultad para disipar el calor producido al trabajar con frecuencias de reloj cada vez mayores (el *Power Wall*).

Además, de forma paralela al desarrollo de las CPU's se ha producido un cambio importante en las unidades de procesamiento gráfico, las GPU's. Inicialmente éstas estaban ancladas al mundo de los videojuegos, y su capacidad para ser programadas se limitaba a la inclusión de efectos que se añadían a la cadena de procesamiento gráfico de píxeles y vértices, claramente no orientadas a la computación de propósito general. Con la llegada de DirectX 9 se añadieron algunas capacidades, como el soporte de operaciones en punto flotante, que supuso una primera llegada de lo que se conoce como GPGPU (General-Purpose Computing on Graphics Processing Units), a través de una programación algo pobre a través de APIs gráficas. Pero fue con la aparición de la versión 10 de DirectX, que trajo consigo una arquitectura unificada con nuevas unidades funcionales así como la capacidad de balanceo de carga en hardware, con la que se consiguió abrir el camino definitivamente [9].

Estos avances hardware han motivado la aparición de diversos entornos y APIs de programación que proporcionan modelos de ejecución sobre los distintos tipos de arquitecturas hardware a un nivel superior de abstracción, facilitando los desarrollos de aplicaciones que permitan aprovechar las capacidades de ejecución paralela de las mismas. Entre ellos destacamos OpenMP (Open Multi-Processing) para arquitecturas de memoria compartida, MPI (Message Passing Interface) para arquitecturas de memoria distribuida y CUDA (Computer Unified Device Architecture) para las unidades de proceso gráfico de NVIDIA.

2.1. OpenMP.

OpenMP es un interfaz de programación de aplicaciones compuesto por un conjunto de funciones, directivas de compilación y variables de entorno que proporcionan un modelo de ejecución paralelo bajo arquitecturas de memoria compartida.

Este tipo de arquitecturas están formadas por varios procesadores, cada uno de los cuales tiene acceso a una zona de almacenamiento global a través de algún bus o red de interconexión y bajo un único espacio de direccionamiento, por lo que todos ellos tienen la misma visión de la memoria.

La comunicación entre procesadores se lleva a cabo mediante lecturas y escrituras en memoria, las cuales no se manejan explícitamente sino que es el propio sistema de memoria el que se encarga de ello, facilitando así la programación bajo este modelo. El principal problema de estas arquitecturas se encuentra en su escalabilidad, ya que conforme aumenta el número de procesadores el sistema de memoria se convierte en un cuello de botella si todos tratan de acceder al mismo tiempo.

Fueron varias las empresas que desarrollaron conjuntamente OpenMP y lo propusieron como un estándar para la industria de la computación, entre ellas Intel Corporation, IBM, Silicon Graphics Inc., etc. Además, otras empresas tanto de hardware como de software lo aprobaron y apoyaron, y a día de hoy puede considerarse el estándar para el desarrollo bajo el modelo de memoria compartida.

2.2. MPI.

MPI es un estándar de programación de aplicaciones paralelas que define la sintaxis y semántica de un conjunto de operaciones de paso de mensajes que permiten explotar el paralelismo en arquitecturas de múltiples procesadores, y que aunque está más orientado al desarrollo bajo sistemas de memoria distribuida también puede utilizarse en otro tipo de sistemas.

Dichas arquitecturas de memoria distribuida están formadas por un conjunto de procesadores, cada uno de los cuales tiene su propia memoria local y cuyo espacio de direcciones de memoria no está mapeado al del resto, por lo que no existe el concepto de memoria global presente en las arquitecturas de memoria compartida.

Por tanto, una característica importante de las mismas es la necesidad de una red de comunicaciones que conecte a los diferentes procesadores entre sí, de forma que cuando uno de ellos necesite acceder o enviar datos a otro pueda hacerlo a través de dicha red. Además, es tarea del programador el manejar explícitamente las comunicaciones de datos mediante operaciones definidas en el API de paso de mensajes utilizado, en este caso MPI.

MPI fue creado en 1993 como un estándar abierto por un gran número de organizaciones, principalmente de Estados Unidos y Europa, entre las que se incluyen fabricantes de sistemas, investigadores de universidades y organizaciones gubernamentales. Existen, por tanto, numerosas implementaciones de diversos proveedores optimizadas para los sistemas de cada uno de ellos.

2.3. CUDA.

CUDA (Compute Unified Device Architecture) es un framework de desarrollo de aplicaciones paralelas de propósito general mediante el uso de dispositivos de aceleración gráfica (GPU's). Se trata de la alternativa desarrollada por NVIDIA en el campo de la GPGPU y está compuesta por un nuevo modelo de programación que permite a los desarrolladores el acceso a los dispositivos con capacidades CUDA, así como un conjunto de herramientas y librerías de desarrollo para facilitar la labor de los mismos en la programación y depuración de aplicaciones bajo este modelo.

En la figura 2.1 podemos ver un esquema general del conjunto de elementos de la capa software de CUDA. Como vemos, la capa más baja comprende el propio motor de ejecución situado dentro de los dispositivos gráficos, seguido del soporte proporcionado por el kernel del sistema operativo. Sobre éste se sitúa por un lado el soporte DirectX para hacer uso directo de las llamadas del kernel, y por otro el driver CUDA que puede ser utilizado directamente o mediante una nueva capa de abstracción, como es el caso de OpenCL.

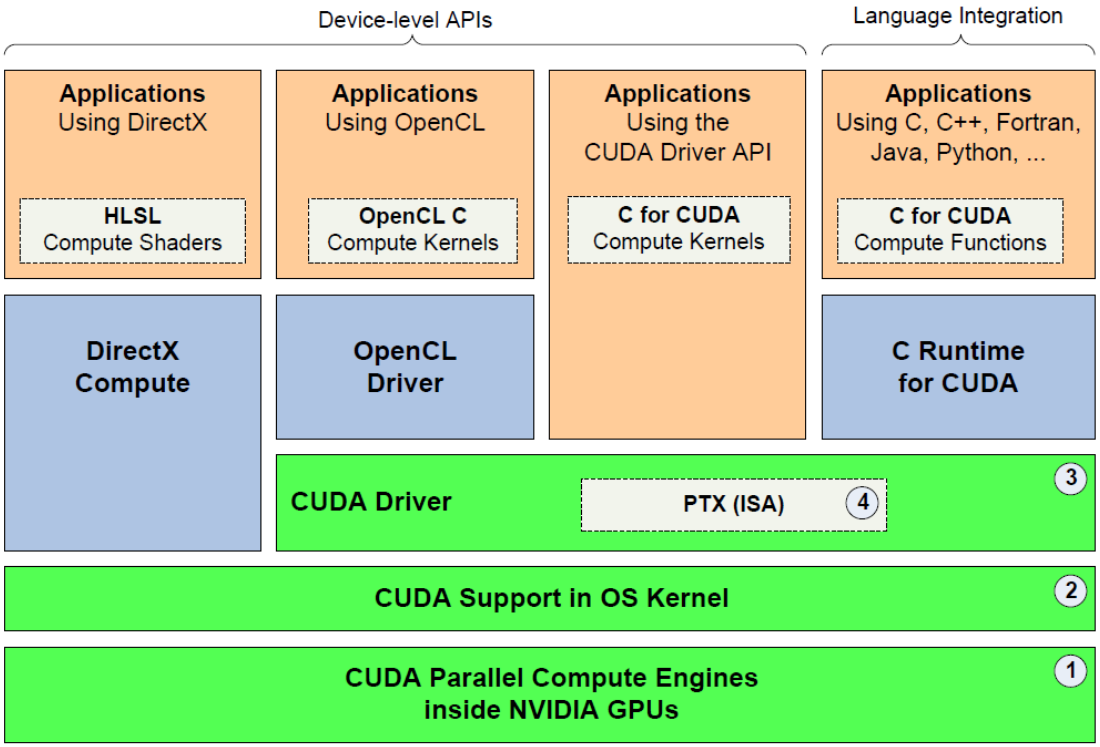

Figura 2.1 - Elementos de la capa software en el modelo CUDA.

2.3.1. Arquitectura general.

En la figura 2.2 se muestra una visión esquemática de la arquitectura general de una GPU. Como vemos en ella, los dispositivos de procesamiento gráfico están formados por un conjunto de multiprocesadores (SM) de tipo MIMD (Multiple Instruction stream, Multiple Data stream), cada uno de los cuales a su vez contiene un grupo de procesadores (SP) de tipo SIMD (Single Instruction stream, Multiple Data stream). Se trata de procesadores de tipo vectorial, los cuales permiten ejecutar una misma operación sobre un conjunto o vector de datos simultáneamente.

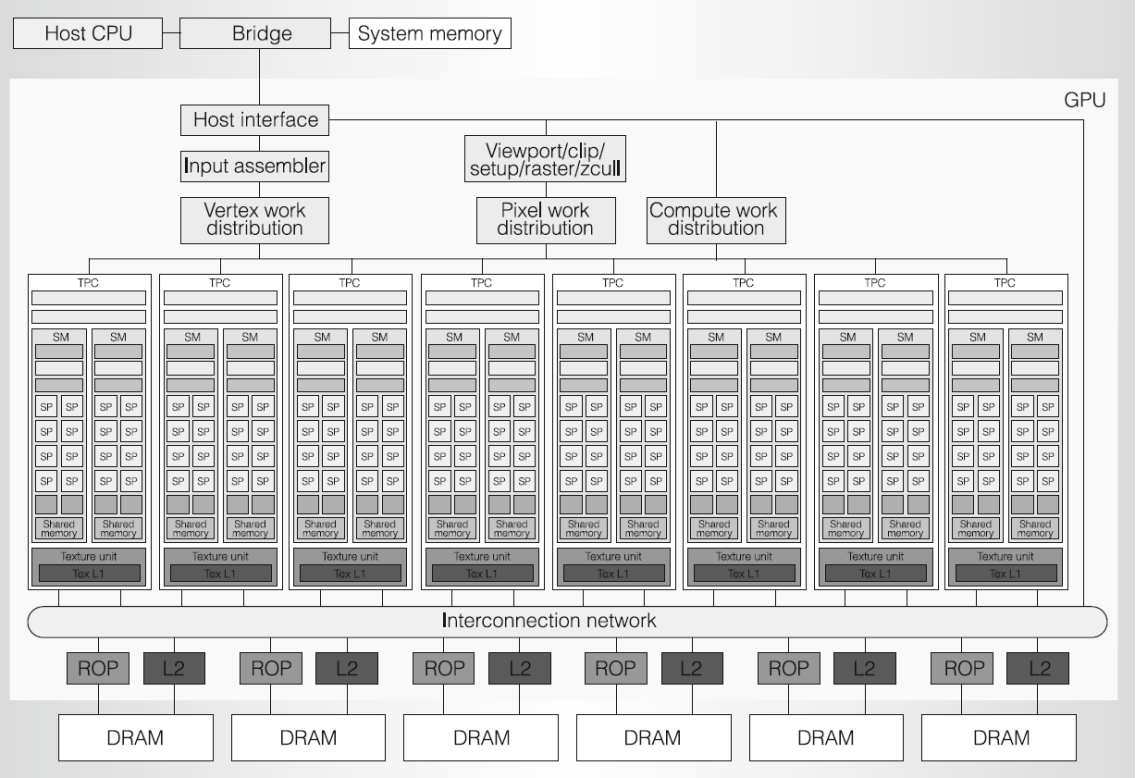

Figura 2.2 - Arquitectura general de una GPU.

Sin embargo, la potencia de cada uno de dichos procesadores es inferior a la de una CPU en términos de frecuencia de operación o de operaciones soportadas, por lo que las mejoras en el rendimiento vienen del aprovechamiento del gran número de procesadores disponibles en la GPU, lo que nos permite lanzar y ejecutar un número elevado de hilos (threads CUDA) que ejecuten un conjunto de operaciones repetidas veces sobre diferentes datos.

Otro aspecto a tener en cuenta es que cada uno de los procesadores o *cores* realiza operaciones sobre datos de tipo entero o de coma flotante en simple precisión. La capacidad de operar sobre datos reales en doble precisión fue añadida a partir de la CUDA Compute Capability 1.3, y dichas operaciones se realizan en unidades funcionales compartidas por todos los cores dentro de un mismo SM.

2.3.2. Modelo de programación.

Como se comentaba anteriormente, la GPU puede verse como un dispositivo capaz de ejecutar un gran número de threads en paralelo, que puede aprovecharse para explotar en gran medida las aplicaciones que hacen uso de paralelismo de datos SIMD. Dado que no todo el código de una aplicación hará uso de este tipo de paralelismo, y los procesadores CUDA son inferiores en potencia a una CPU, debemos utilizar ambas tecnologías para sacar el máximo partido al rendimiento de la aplicación.

Desde este punto de vista, la GPU puede verse como un coprocesador de la CPU donde se ejecutarán las partes de código de una aplicación que sean intensivas en paralelismo de datos, en lo que puede considerarse un entorno de computación heterogénea. Así, podemos diferenciar dos partes en una aplicación que haga uso de CUDA: el código ejecutado en la CPU (host) y el ejecutado en la GPU (device). A estas últimas partes de código se les denomina kernels.

Estos kernels serán ejecutados por un número de threads CUDA que se indican al invocarlos desde la CPU, y se organizan lógicamente como un conjunto (grid) de bloques de threads (thread blocks). En la figura 2.3 podemos ver reflejado este modelo de ejecución. A su vez, cada uno de estos bloques se ejecuta físicamente en un solo multiprocesador, y los threads dentro del mismo pueden cooperar compartiendo datos a través de la memoria compartida de dicho SM y sincronizar su ejecución en determinados puntos del código. Cada bloque, a su vez, se divide en warps (actualmente de 32 threads), siendo esta la unidad mínima de ejecución del modelo CUDA.

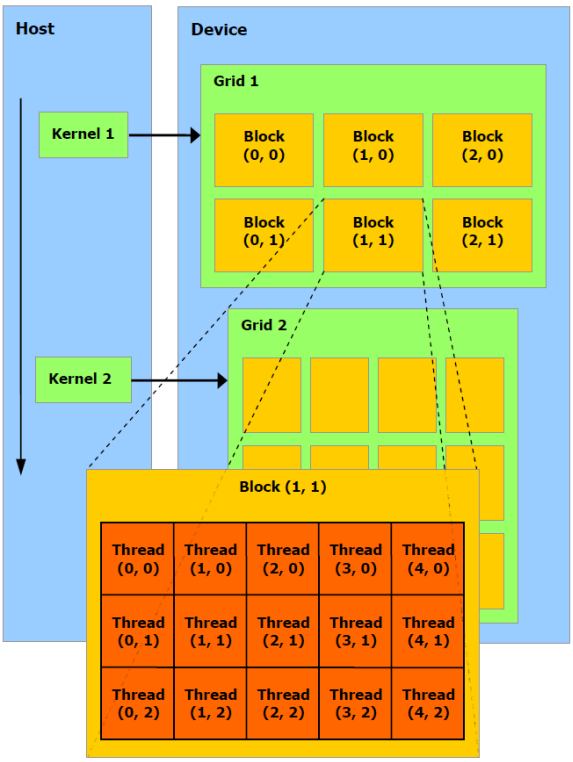

Figura 2.3 - Modelo de ejecución en CUDA.

Además, tanto la CPU como la GPU tienen su propia memoria (host memory y device memory), por lo que las comunicaciones de memoria entre ambos deben hacerse explícitas en el código de la aplicación a través del API proporcionado por CUDA.

2.3.3. Modelo de memoria.

Los threads que se ejecutan en la GPU tienen accesibles diferentes recursos de memoria del mismo, las cuales podemos ver reflejadas en la figura 2.4. Como se puede observar, las memorias global, constante y de texturas pueden ser leídas y escritas desde la CPU. Además, estas regiones de memoria son persistentes entre todas las llamadas a kernels realizadas por una misma aplicación.

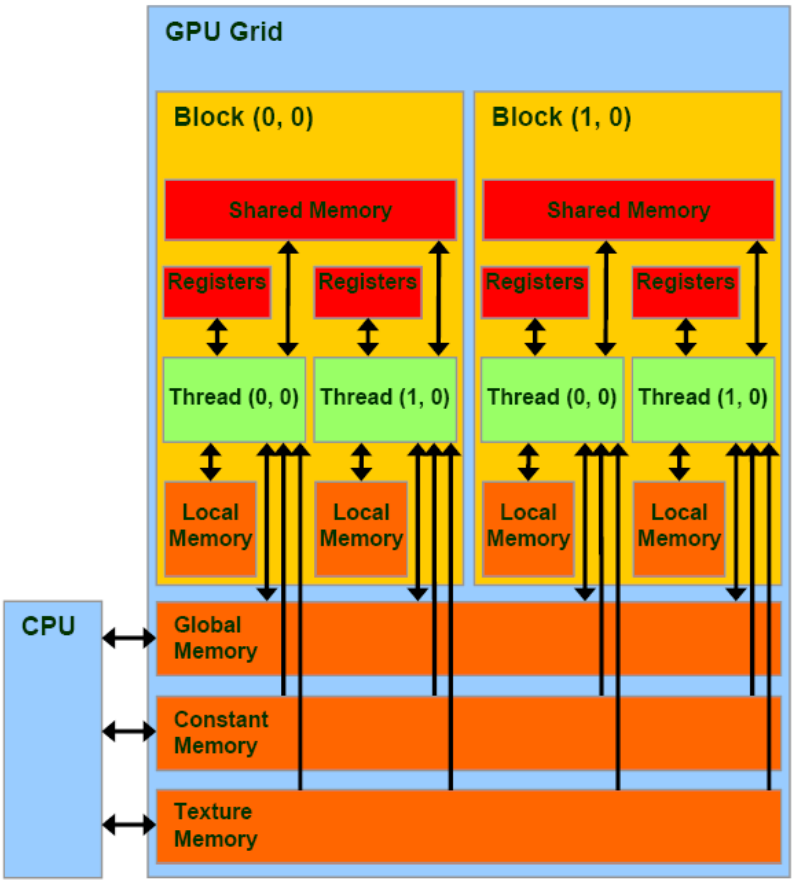

Figura 2.4 - Modelo de memoria en CUDA.

La memoria global es la de mayor capacidad, pero a la vez es la que tiene una mayor latencia de acceso así como un menor ancho de banda. Suele ser la memoria principal para el intercambio de datos entre el host y el dispositivo, por lo que un buen uso de la misma nos permitirá obtener un mejor rendimiento en la ejecución de una aplicación.

En particular, es importante seguir unos patrones de acceso a la misma determinados, los cuales se conocen como accesos coalesced. La idea es que los accesos a memoria contigua se realicen simultáneamente en cada half-warp (16 threads), para lo cual se deben cumplir las siguientes condiciones:

- En dispositivos con CUDA Compute Capability 1.0 y 1.1:
  - El tamaño de palabra al que cada thread accede debe ser o bien de 4 u 8 bytes (se realizará una única transacción de memoria) o de 16 bytes (se realizarán 2 transacciones).
  - Las 16 palabras accedidas deben encontrarse en el mismo segmento de memoria.
  - Los threads deben acceder a los datos de forma secuencial. En la figura 2.5 podemos ver un ejemplo de acceso coalescente donde se cumple esta condición, y en la figura 2.6 nos encontramos un acceso que la viola, resultando en múltiples transacciones de memoria.
- En versiones de CUDA 1.2 y superiores los requisitos se relajan, eliminando la restricción de la secuencialidad de los accesos. La figura 2.7 muestra un acceso coalesced cuando no se cumple la restricción de secuencialidad.

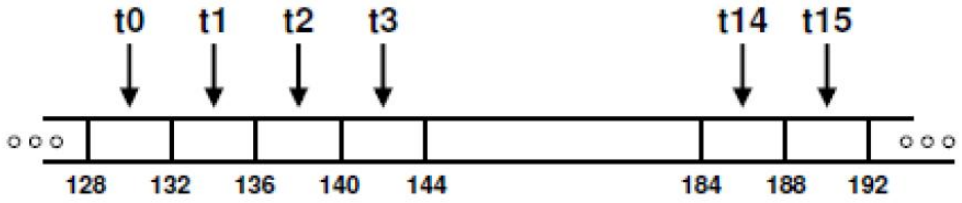

Figura 2.5 - Acceso coalesced, realizado en una única transacción de memoria. Todos los threads participan leyendo datos en posiciones de memoria consecutivas.

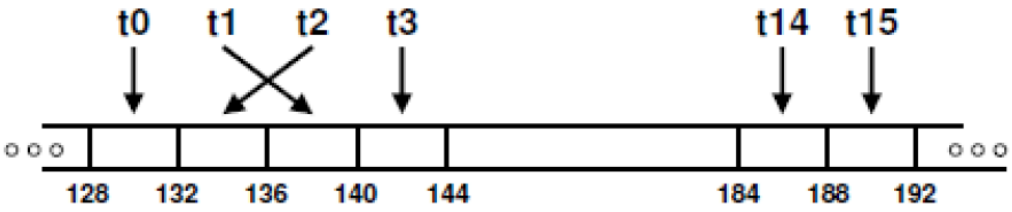

Figura 2.6 - Acceso no coalesced, lo que supone 16 accesos totales a memoria. Los threads 1 y 2 violan la condición de acceso secuencial a memoria.

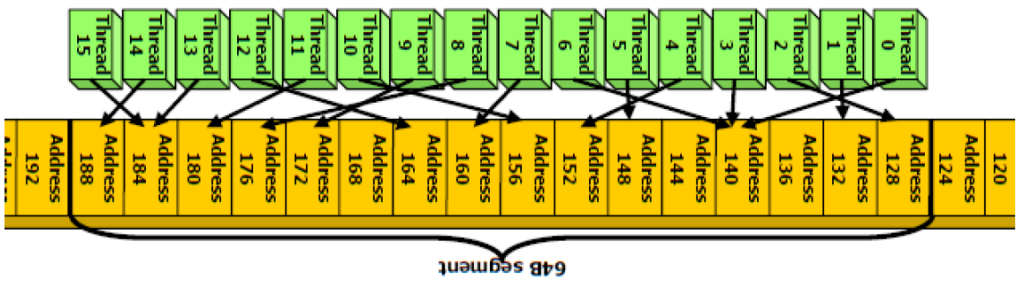

Figura 2.7 - Acceso coalesced en versiones de CUDA posteriores a la 1.2.

Por último mostramos en la figura 2.8 los distintos recursos de memoria disponibles en un dispositivo CUDA, donde se indica el ámbito y tiempo de vida de cada uno de ellos. Las memorias shared y register son las más rápidas ya que están situadas en el propio chip, y las más lentas son las memorias global, local y de texturas. De entre todas ellas las más utilizadas son la memoria global, principalmente por su gran capacidad, y las memorias register y shared.

Memory	Location	Cached	Access	Scope	Lifetime
Register	On chip	N	R/W	1 thread	Thread
Local	RAM	N	R/W	1 thread	Thread
Shared	On chip	N	R/W	Threads in a block	Block
Global	RAM	N	R/W	All thread + host	Host allocation
Constant	RAM	Y	R	All thread + host	Host allocation
Texture	RAM	Y	R	All thread + host	Host allocation

Figura 2.8 - Características de los distintos recursos de memoria disponibles.

2.4. Servidor LUNA.

La evaluación de las distintas rutinas utilizadas y desarrolladas a lo largo del proyecto se realizará en el servidor LUNA del Grupo de Computación Científica y Programación Paralela de la Universidad de Murcia. Se trata de una máquina de memoria compartida con las siguientes características:

- Procesador Intel Core 2 Quad Q6600, con 4 cores a una frecuencia de reloj de 2.4 GHz y 8 MB de memoria caché de segundo nivel (L2).
- 4 GB de memoria RAM a una frecuencia de reloj de 800 MHz, divididos en dos módulos de 2 GB.
- Dispositivo gráfico NVIDIA GeForce 9800 GT, con CUDA Capability 1.1. Dispone de 14 multiprocesadores, cada uno de los cuales contiene 8 cores, resultando en un total de 112 cores de 1.50 GHz.
- Sistema operativo Ubuntu 10.04 i686 para arquitecturas de 32 bits.

3. Guía de placas paralelas en una dimensión.

El primer paso en nuestro estudio se corresponde con el análisis de las funciones de Green de la guía de placas paralelas en una única dimensión. El sistema de referencia sobre el cual trabajamos se muestra en la figura 3.1.

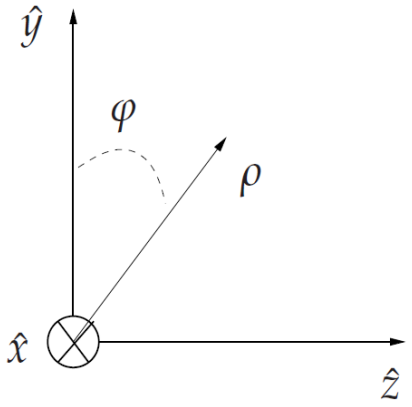

Figura 3.1 - Sistema de referencia utilizado.

Sobre este sistema de referencia situamos una guía de placas paralelas a lo largo del eje z, tal como podemos ver en la figura 3.2. Dentro de la misma se sitúan un conjunto de puntos fuente y de observación que se mueven en los ejes $\hat{y}$ y $\hat{z}$, mientras que la variación en el eje $\hat{x}$ queda recogida en función de un parámetro externo, la frecuencia espacial k_x.

De esta forma, para cada pareja de puntos tendremos que calcular la función de Green asociada a los mismos. Como se comentaba anteriormente, el cálculo de dichas funciones tiene una expresión analítica más o menos sencilla, consistente en una serie cuya convergencia en el dominio real es bastante lenta. Para acelerar la misma se hace uso del método de Ewald, descomponiendo dicha serie en dos términos, uno que se mantiene en el dominio real y otro que se realiza en el dominio de Fourier, de forma que ambos aceleran su convergencia.

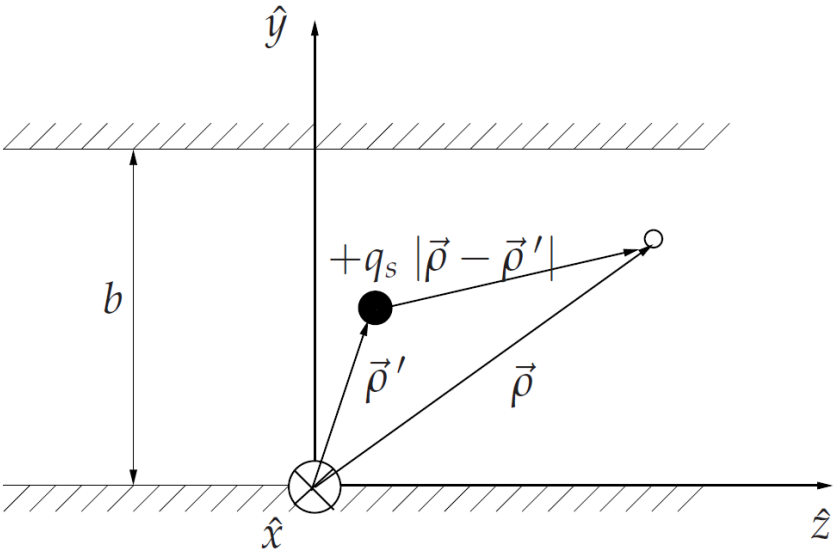

Figura 3.2 - Guía de placas paralelas en el sistema de referencia.

Así, para cada una de las funciones de Green a calcular se necesita evaluar un número determinado de términos de dichas series, el cual se puede fijar a un número determinado o, más convenientemente, variar dinámicamente en función de la distancia entre ambos puntos a lo largo del eje de la guía de placas, dada por $|z_o - z_s|$.

Por tanto dividiremos este capítulo en varias partes en función tanto del número de puntos fuente y de observación como del número de iteraciones (número de modos) a evaluar en cada una de las series. Comenzaremos por el caso más sencillo, donde únicamente tenemos un punto fuente y un punto de observación y el número de términos de la serie es constante, y finalizaremos con el caso más general con n puntos fuente, m puntos de observación y un número de modos variable en función de la separación entre cada pareja de puntos.

3.1. Análisis del algoritmo secuencial.

En primer lugar analizaremos el código fuente del algoritmo secuencial original, tratando de comprender la secuencia de llamadas dentro del mismo y el coste computacional de cada una de las partes del programa, para intentar obtener una versión secuencial optimizada antes de comenzar a aplicar el paralelismo. Además, este análisis nos permitirá estimar qué partes del algoritmo son más susceptibles de ser paralelizadas, así como determinar si obtendremos o no mejora alguna con dicho paralelismo.

Partimos de un código en el lenguaje Fortran, proporcionado por el Grupo de Electromagnetismo Computacional de la Universidad Politécnica de Cartagena, y cuyo esquema en pseudocódigo podemos observar en la figura 3.3. El código se divide principalmente en los siguientes tres ficheros:

- GF_unidimensional.f90: contiene la rutina principal del programa, donde se lee un fichero de entrada con los parámetros necesarios para el cálculo de las funciones de Green (ubicación de los puntos fuente y de observación, número de iteraciones para el cálculo del sumatorio y otros parámetros externos) y se llama a la rutina para el cálculo de las mismas.
- GF_Capa2Dt_TEx_conv.f90: se encarga de invocar a la rutina de cálculo de las funciones de Green para cada pareja de puntos fuente y observación a calcular.
- Capa_acceler.f90: contiene el conjunto de funciones que se encargan de calcular la función de Green de la guía de placas paralelas:
    - sbyparts: es la rutina principal, desde la cual se invoca a las restantes para el cálculo de las series y posteriormente se calcula la parte asintótica correspondiente al método de aceleración de Kummer [19].
    - dsum: se encarga de calcular el sumatorio de las funciones trigonométricas.
    - spectgf: se encarga de calcular la parte del sumatorio en el dominio espectral.

```
funcion GF_unidimensional::main()
       Leer parámetros de entrada
       GF_Capa2Dt_Tex_conv(m, n, nmod)
fin_funcion

funcion GF_Capa2Dt_TEx_conv::GF_Capa2Dt_TEx_conv(m, n, nmod)
       para cada punto m de observación
              para cada punto n fuente
                     sbyparts(m, n, nmod)
              fin_para
       fin_para
fin_funcion

funcion Capa_acceler::sbyparts(m, n, nmod)
       spectgf(nmod)
       dsum(nmod)
       Aplicar método de aceleración de Kummer
fin_funcion

funcion Capa_acceler::spectgf(nmod)
       repetir nmod iteraciones
              Operaciones trigonométricas
       fin_repetir
fin_funcion

función Capa_acceler::dsum(nmod)
       repetir nmod iteraciones
              Operaciones exponenciales
       fin_repetir
fin_funcion
```

Figura 3.3 - Esquema de código seguido en el cálculo de las funciones de Green unidimensionales.

Es interesante también comprobar qué porcentaje del tiempo total de ejecución se utiliza en la ejecución de cada una de las dos funciones anteriores. Para tal efecto utilizamos la herramienta callgrind del tookit de valgrind, que nos produce un perfil de ejecución mostrando el árbol de llamadas de nuestro programa junto al tiempo consumido en cada una. Los resultados los podemos observar en la figura 3.4, y como vemos muestran que en la ejecución de la función spectgf se consume cerca de un 90% del tiempo total de ejecución del programa, mientras que la ejecución de la función dsum no llega al 10%. Esto se debe principalmente a que en la función spectgf se ejecutan un mayor número de operaciones, entre las cuales se encuentran exponenciales complejas que tienen un alto coste computacional. Por el contrario, en la función dsum el número de operaciones ejecutado es menor, y todas ellas consisten en operaciones trigonométricas de números reales.

Así, además de adaptar el código del lenguaje Fortran a C++, parece interesante tratar de optimizar el mismo, en particular la función spectgf, para intentar reducir el porcentaje del tiempo de ejecución consumido en la misma. Analizando el código de dicha función, nos encontramos con que podemos mejorar los siguientes aspectos:

- En cada iteración podemos precalcular las funciones exponenciales necesarias, de forma que sólo se calculen una vez por iteración.
- En el bucle se realiza un análisis de casos que podemos eliminar si sacamos del mismo la primera iteración, eliminando así los condicionales y mejorando el flujo de ejecución dentro del mismo.

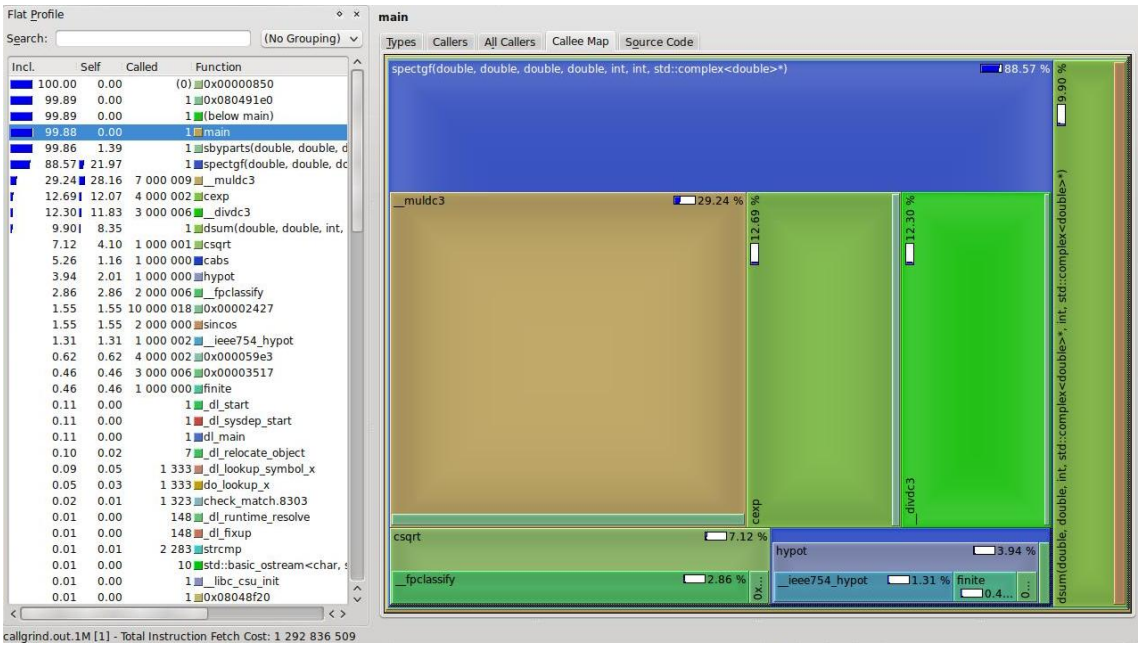

Figura 3.4 - Perfil de ejecución del algoritmo secuencial original.

Con estas sencillas optimizaciones realizamos de nuevo un perfil de ejecución del algoritmo, que queda reflejado en la figura 3.5. Como podemos ver, el porcentaje relativo del tiempo total de ejecución sigue siendo muy superior en la función spectgf con un 76% del mismo, mientras que la función dsum ahora consume el 20%. Sin embargo, el número total de instrucciones ejecutadas ha disminuido desde 1.292.836.509 a 622.835.961, suponiendo una reducción de algo más de un 50% en las mismas.

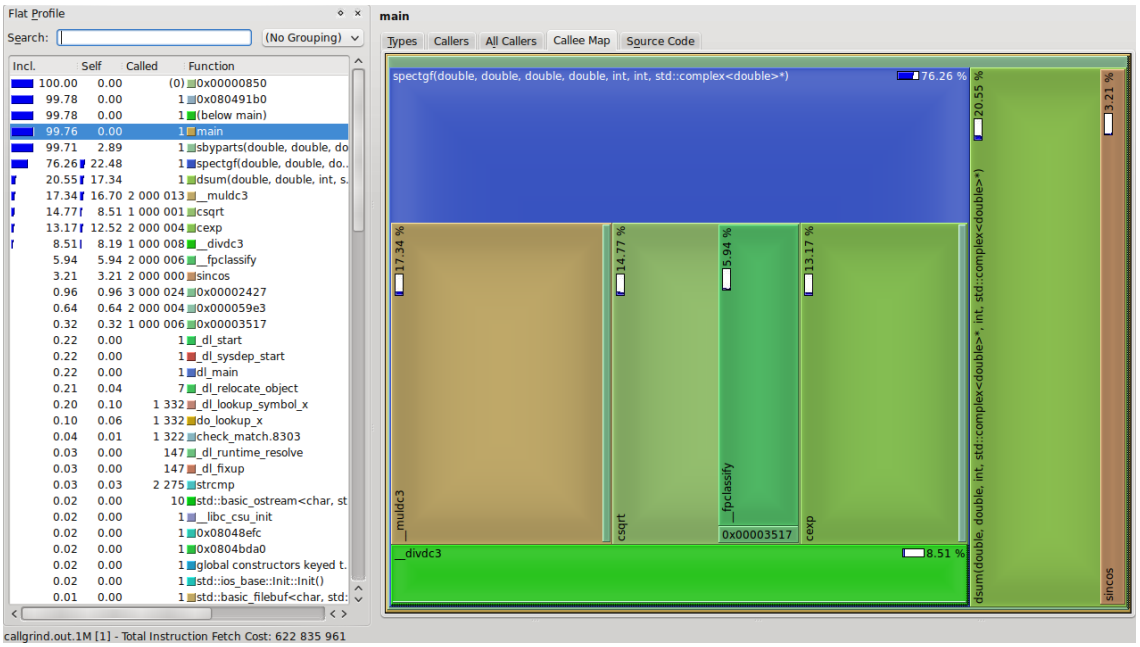

Figura 3.5 - Perfil de ejecución del algoritmo secuencial optimizado.

Por último, en la figura 3.6 mostramos una gráfica para comparar los tiempos de ejecución del algoritmo secuencial tanto en su versión original en Fortran como en las dos versiones realizadas en C++, en función del número de términos del sumatorio calculados. Podemos observar que la mejora es despreciable para un número de iteraciones inferior a 10000. Además, teniendo en cuenta que el número de términos utilizado en una ejecución típica está en torno a los 100, podemos anticipar que no obtendremos una mejora notable utilizando paralelismo.

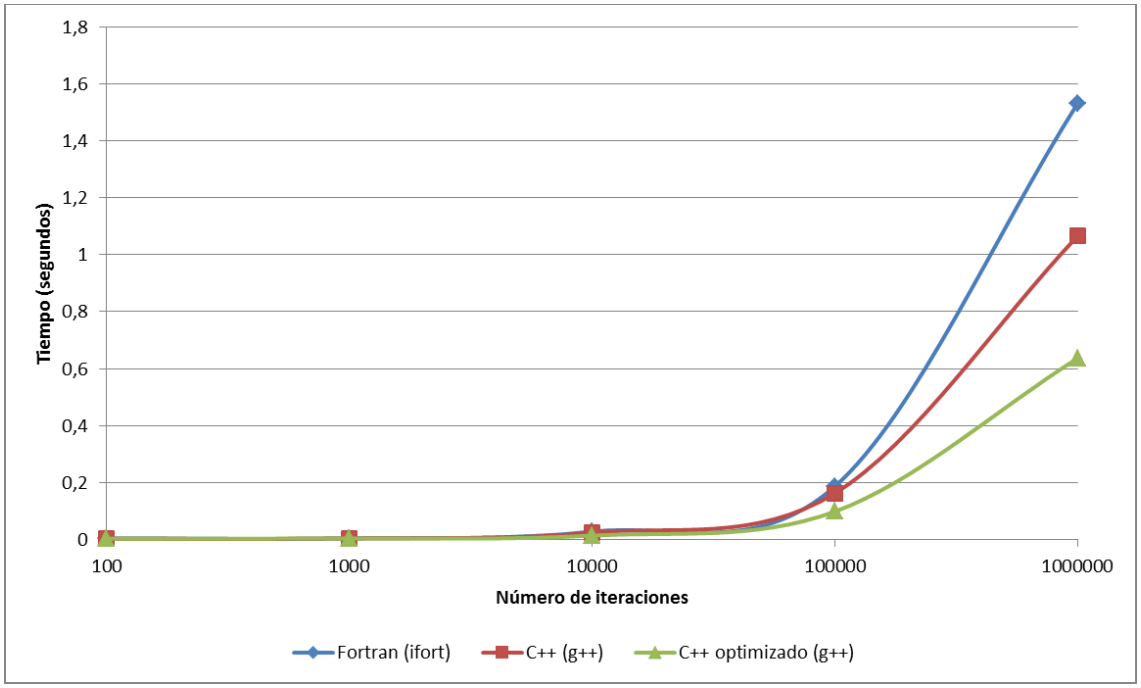

Figura 3.6 - Comparativa de los tiempos de ejecución del algoritmo secuencial.

3.2. Un punto fuente y un punto de observación.

A continuación pasaremos a analizar el caso más sencillo, donde nos encontramos un único punto fuente y un único punto de observación entre los cuales calcular la función de Green correspondiente, empleando para ello un número fijo de iteraciones para el cálculo de la serie.

El tiempo teórico de ejecución del algoritmo nos queda, para n puntos fuente, m puntos de observación y nmod iteraciones, y denotando con k el coste de las operaciones trigonométricas y exponenciales de los sumatorios más internos:

$$t(n,m,nmod) = \sum_{1}^{n}\sum_{1}^{m} 2 * \left(\sum_{1}^{nmod} k \right) = 2 * k * n * m * nmod$$

Por tanto, para este primer caso con un único punto fuente y otro de observación y un número fijo de iteraciones para el cálculo del sumatorio, el tiempo teórico nos queda:

$$t(nmod) = 2 * k * nmod$$

Como era de esperar, al tratarse del caso más sencillo del problema el tiempo de ejecución es relativamente bajo, con un orden lineal al número de iteraciones del sumatorio.

Así, las posibilidades de paralelismo aplicables en este primer caso se centran en los dos bucles más internos, correspondientes a las funciones spectgf y dsum. Si analizamos las dependencias de datos entre las iteraciones de cada uno de ellos nos encontramos lo siguiente:

- El bucle de la función spectgf se encarga de rellenar un array de nmod filas y 3 columnas, de forma que en cada iteración se calcula una fila y se hace de forma independiente del resto.
- El bucle de la función dsum se encarga de rellenar un array de 6 posiciones, de forma que en cada iteración se actualiza el valor de cada posición sumándole al valor previamente calculado un nuevo término. En la figura 3.7 mostramos la evolución del contenido de dicho array tras el cálculo de cada nueva iteración de la función.

	Array[0]	Array[1]	Array[2]	Array[3]	Array[4]	Array[5]
Iteración 0	A_0	B_0	C_0	D_0	E_0	F_0
Iteración 1	A_0+A_1	B_0+B_1	C_0+C_1	D_0+D_1	E_0+E_1	F_0+F_1
Iteración 2	$A_0+A_1+A_2$	$B_0+B_1+B_2$	$C_0+C_1+C_2$	$D_0+D_1+D_2$	$E_0+E_1+E_2$	$F_0+F_1+F_2$
...	...	...	...	...	...	...
Iteración nmod	$\sum_{i=0}^{nmod} A_i$	$\sum_{i=0}^{nmod} B_i$	$\sum_{i=0}^{nmod} C_i$	$\sum_{i=0}^{nmod} D_i$	$\sum_{i=0}^{nmod} E_i$	$\sum_{i=0}^{nmod} F_i$

Figura 3.7 - Evolución del contenido del array calculado en la función dsum.

Por tanto, en el caso del primer bucle podemos realizar una paralelización del mismo asignando a cada unidad de proceso un conjunto de iteraciones a calcular, ya que no existen dependencias de datos entre ellas. Sin embargo, en el caso del segundo bucle esto no es tan sencillo, ya que al existir dependencias debemos buscar una solución alternativa.

Si nos fijamos de nuevo en la figura 3.7 en la forma en que se calcula cada posición del array en cada iteración, vemos que depende únicamente del valor calculado en la iteración previa en la misma posición más un término que se calcula directamente. Por tanto, al no existir dependencias de los valores de una posición del array con los del resto de posiciones, una posibilidad podría ser calcular en paralelo los valores de cada una de las posiciones del array, de forma que cada unidad de proceso calcularía todas las iteraciones de una única posición. El principal problema de esta aproximación es que el factor de mejora que conseguiría sería como máximo de 6, que es el número de posiciones del array, y no dependería del número de iteraciones a calcular, por lo que no parece la mejor solución.

Otra alternativa es aplicar un algoritmo de reducción sobre los valores de cada posición del array calculados en cada iteración aplicando sobre los mismos una operación de suma. Así, utilizando un algoritmo de suma por reducción en árbol tal como se muestra en la figura 3.8 conseguiríamos una mejora proporcional al número de unidades de proceso utilizadas.

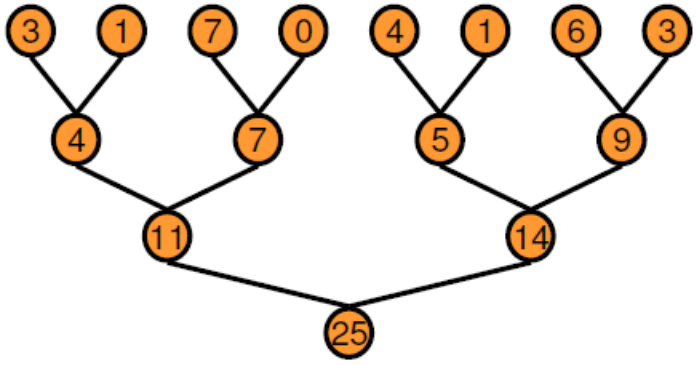

Figura 3.8 - Esquema de reducción en árbol.

Partiendo del código secuencial optimizado obtenido anteriormente, y teniendo en cuenta que la función donde se consume el mayor tiempo de ejecución es la que incluye la ejecución del primer bucle, comenzaremos por paralelizar dicha función y analizar la mejora en el rendimiento. Para ello haremos uso tanto de OpenMP como de CUDA para su implementación en un sistema de memoria compartida y en una unidad de proceso gráfico, respectivamente. Despreciamos a priori, pues, la opción de utilizar MPI debido a que al tratarse de un problema de bajo coste computacional, la sobrecarga introducida por el paso de mensajes en arquitecturas de memoria distribuida supondría un empeoramiento del algoritmo secuencial.

Así pues, para el caso de OpenMP utilizaremos un pragma para repartir el número de iteraciones entre el número de unidades de proceso disponibles. La figura 3.9 recoge el esquema en pseudocódigo resultante del bucle de la función a paralelizar.

```
#pragma omp parallel for
       for(int i = 0; i < num_iteraciones; i++)
              Compute spectral domain Green's functions
              gf[i][0] = Direct function
              gf[i][1] = Derivative respect z-axis
              gf[i][2] = Derivative respect y-axis
```

Figura 3.9 - Pseudocódigo en OpenMP de la paralelización de la función spectgf.

En cuanto a la implementación mediante el paradigma CUDA, tenemos que elegir la organización del grid de threads que utilizaremos. La opción más sencilla consistiría en utilizar un thread CUDA para el cálculo de cada iteración, pero nos encontramos con la limitación impuesta por cada GPU del número máximo de threads a utilizar en un único thread block.

En general, en las tarjetas actuales este número suele ser 256 o 512, lo que bastaría para una ejecución típica donde el número de iteraciones no será superior a unas pocas centenas. Sin embargo, para el caso más general podríamos pensar en otra solución utilizando varios thread blocks, cada uno con un número fijo de threads fijado por un parámetro para el cual una buena elección suele ser 64 o 128.

De esta forma nos quedaría un código como el mostrado en la figura 3.10. Como vemos, nuestro grid estaría formado por $\left\lceil \frac{numIteraciones}{THREADS\_PER\_BLOCK} \right\rceil$ thread blocks, cada uno con THREADS_PER_BLOCK threads CUDA, de forma que en el kernel cada uno de ellos se encargaría de calcular una iteración concreta.

```
__global__ void spectgf_cuda_kernel (...)
       tn = (THREADS_PER_BLOCK * blockId.y) + threadId.y;

       if(tn < numIteraciones)
              Compute spectral domain Green's functions
              gf[tn][0] = Direct function
              gf[tn][1] = Derivative respect z-axis
              gf[tn][2] = Derivative respect y-axis
       end_if
end_function

void spectgf_cuda(...)
       dim3 grid(1, ceil(numIteraciones / THREADS_PER_BLOCK));
       dim3 block(1, THREADS_PER_BLOCK);

       spectgf_cuda_kernel<<<grid, block>>>(...);
end_function
```

Figura 3.10 - Pseudocódigo en CUDA de la paralelización de la función spectgf, incluyendo el código del kernel y el de la invocación al mismo.

Los resultados de ambas implementaciones se muestran en la figura 3.11. Como era de esperar, debido al bajo coste computacional del cálculo de las funciones para un número bajo de iteraciones, los tiempos de ejecución no mejoran al del código secuencial.

Sin embargo, de nuevo obtenemos una mejora para un número de iteraciones superior a 100000. En particular podemos observar que, para un número suficientemente alto de las mismas, la implementación que mejor se comporta es la desarrollada en CUDA, con una reducción de algo más de un 75% en el tiempo de ejecución respecto a la rutina secuencial original en Fortran. El principal problema de dicha implementación reside en que añade una sobrecarga al coste computacional de aproximadamente 0,1 segundos, por lo que para problemas cuyo coste sea menor a dicha sobrecarga no obtendremos mejora alguna.

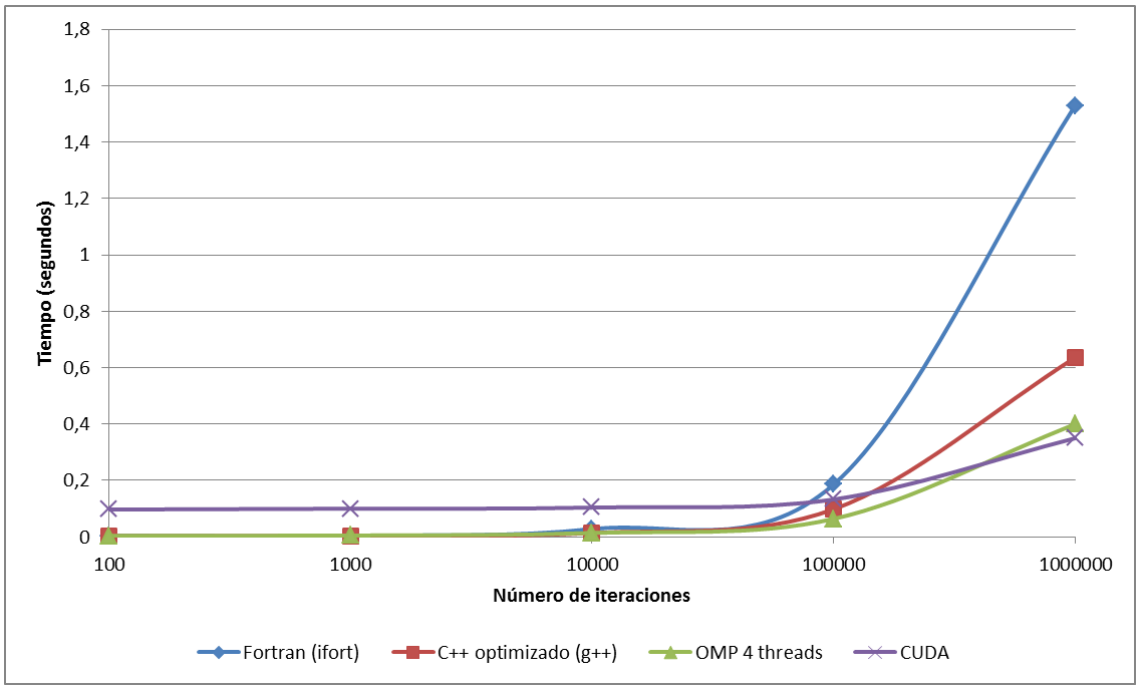

Figura 3.11 - Comparativa de los tiempos de ejecución obtenidos para las distintas rutinas desarrolladas para el cálculo de una única función de Green.

3.3. Varios puntos fuente y varios puntos de observación.

Una vez analizadas las posibilidades de paralelismo que podemos aplicar al problema más elemental del cálculo de una función de Green para un único punto fuente y un único punto de observación, el siguiente paso es estudiar el caso más genérico con n puntos fuente y m puntos de observación.

Tal y como comentábamos anteriormente, el tiempo teórico de ejecución del algoritmo secuencial era el siguiente:

$$t(n,m,nmod) = \sum_{1}^{n}\sum_{1}^{m} 2 * \left(\sum_{1}^{nmod} k \right) = 2 * k * n * m * nmod$$

Como vemos, en este caso el orden de complejidad del mismo aumenta linealmente respecto a 3 factores: el número de puntos fuente (n), el número de puntos de observación (m), y el número de modos o iteraciones a calcular de la serie (nmod). Esto, además, supone la aparición de dos niveles de paralelismo aplicables al problema:

- Por un lado tenemos un nivel más interno, limitado al cálculo de una única función de Green, y para el cual podemos utilizar las rutinas desarrolladas en el capítulo anterior.
- Por otro lado tenemos un nivel más externo que nos permite utilizar un paralelismo de grano grueso, donde en vez de paralelizar el cálculo de una función de Green, lo que hacemos es calcular dichas funciones para varias parejas de puntos fuente y de observación al mismo tiempo.

En el primer caso el beneficio que podemos esperar del paralelismo será proporcional al número de iteraciones a calcular en la función de Green, mientras que en el segundo caso lo será al número de puntos fuente y de observación. Dado que generalmente n * m > nmod, podemos esperar mejores resultados utilizando dicho paralelismo de grano grueso.

Como comentábamos anteriormente, para el paralelismo más interno se han hecho uso de las rutinas desarrolladas en el capítulo anterior. Sin embargo, se ha analizado también una nueva mejora en las subrutinas spectgf y dsum necesarias para el cálculo de una función de Green.

Hasta ahora ambas subrutinas, invocadas desde sbyparts, se ejecutaban separadamente de forma que en la primera de ellas se calculaban unos determinados coeficientes que se almacenaban en una matriz temporal de 3 * num_iteraciones elementos llamada tf, y que posteriormente se utilizaban en la segunda subrutina para calcular los valores definitivos de la función de Green, almacenados en la matriz de 6 posiciones fp.

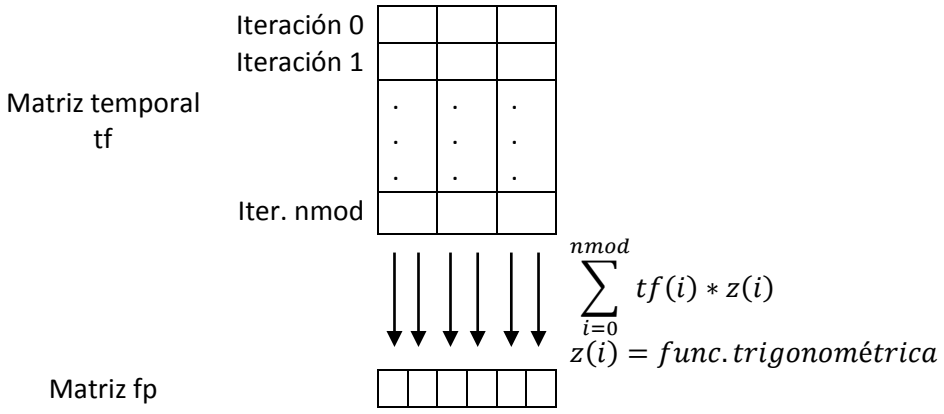

Figura 3.12 - Los valores de la matriz tf, calculados por la subrutina spectgf, son posteriormente utilizados por la subrutina dsum para calcular los valores de las funciones de Green en la matriz fp.

Cuando analizamos en el capítulo anterior el porcentaje del tiempo de ejecución consumido por la función dsum nos encontrábamos con que era bastante menor que el de la función spectgf, y ya que estábamos con el problema más básico convenimos que en dicho momento la paralelización de la función dsum no supondría una mejora relevante.

Sin embargo, ahora que no vamos a tratar con una única invocación a dicha rutina, sino con un número elevado de ellas, parece interesante que la ejecución se la misma se realice también en paralelo. En la figura 3.13 podemos ver una primera aproximación en pseudocódigo a esta propuesta. Como vemos, se trata de utilizar un nuevo pragma de OpenMP para paralelizar la función dsum al igual que hicimos anteriormente con la función spectgf.

```
funcion sbyparts()
	tf = spectgf(tf)
	fp = dsum(tf)
end_funcion

funcion spectgf(tf)
	#pragma omp parallel for
		for(int i = 0; i < num_iteraciones; i++)
			Compute spectral domain Green's functions in tf
	return tf;
end_funcion

funcion dsum(tf)
	#pragma omp parallel for
		for(int i = 0; i < num_iteraciones; i++)
			Compute green functions using tf in fp
	return fp
end_funcion
```

Figura 3.13 - Pseudocódigo en OpenMP de la paralelización de las funciones spectgf y dsum.

Si evaluamos el rendimiento de este nuevo código comparado con el anterior, efectivamente notaremos una mejora en el tiempo de ejecución. Sin embargo, aún podemos mejorar el mismo si observamos que los coeficientes temporales tf calculados en la iteración i de la función spectgf únicamente se utilizan en la misma iteración de la función dsum. Por tanto, podemos unir ambas funciones y eliminar la matriz temporal de coeficientes, tal como se ilustra en la figura 3.14.

```
function sbyparts()
	fp = spectgf_and_dsum()
	Apply Kummer method to fp values
end_function

function spectgf_and_dsum()
	#pragma omp parallel for
		for(int i = 0; i < num_iteraciones; i++)
			Compute spectral domain Green's functions
			Use them to compute Green functions and store in fp
	return fp;
end_function
```

Figura 3.14 - Pseudocódigo en OpenMP de la unión de las funciones spectgf y dsum y su posterior paralelización.

En la figura 3.15 se muestran los tiempos de ejecución obtenidos para la versión inicial desarrollada en OpenMP y la versión propuesta en la figura 3.14. Como podemos ver, independientemente del número de iteraciones el tiempo de ejecución de la rutina mejorada es de un 50-60% del tiempo de la rutina desarrollada inicialmente, consiguiendo mejores resultados cuantas más parejas de puntos e iteraciones se calculen.

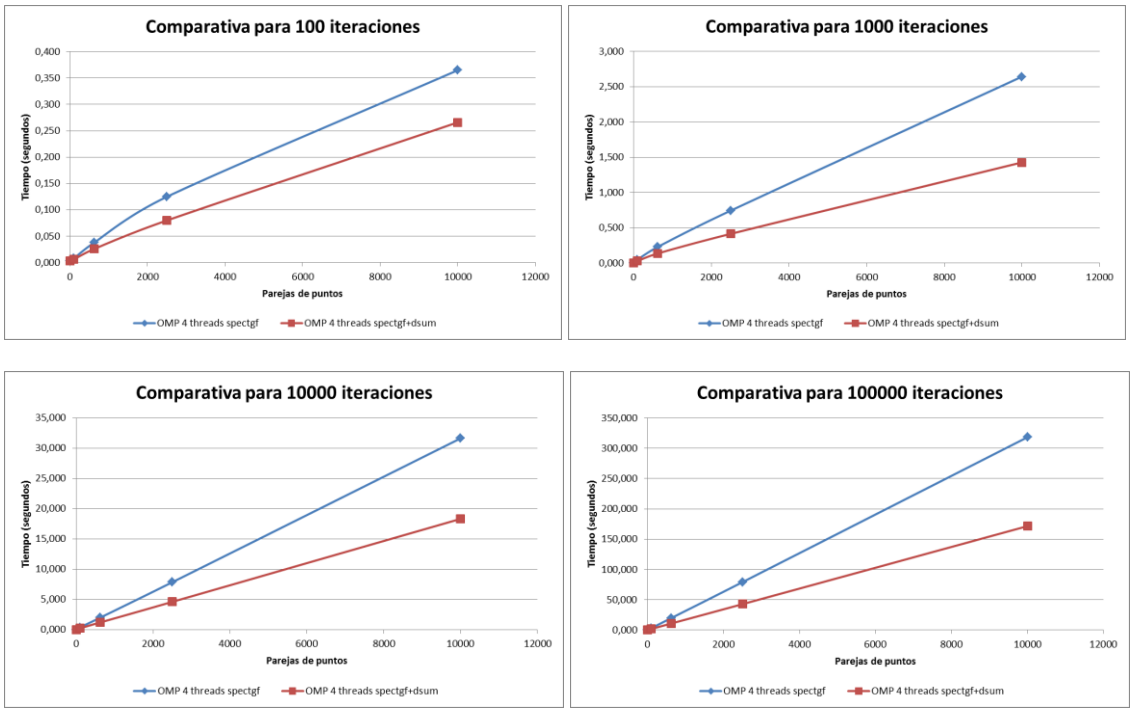

Figura 3.15 – Comparativa de los tiempos de ejecución obtenidos para las dos versiones desarrolladas en OpenMP.

Visto esto, podríamos pensar en aplicar el mismo planteamiento a la versión desarrollada en CUDA. Sin embargo, ya que la filosofía de los threads CUDA se basa en que el paralelismo debe conseguirse a través de la creación de múltiples threads ligeros, esto es, con poca carga de trabajo, al estar incrementando dicha carga posiblemente se vería perjudicado el rendimiento global de la aplicación.

Así pues, presentamos en las figuras 3.16 y 3.17 los resultados de evaluar las distintas rutinas para diferentes tamaños de problema según el número total de parejas de puntos: 10x10, 25x25, 50x50 y 100x100. Como podemos ver en las gráficas, la implementación que mejor se comporta independientemente tanto del número de iteraciones a calcular de las series como de la cantidad de parejas de puntos es la versión de OpenMP mejorada. En cuanto a la implementación de CUDA, su comportamiento depende de dichos parámetros:

- Para un número bajo de iteraciones a calcular (figura 3.16), apenas mejora el tiempo de ejecución de la versión secuencial optimizada en C, llegando incluso a superar dicho tiempo si el número de iteraciones es muy bajo, lo cual es más factible en un problema real que un número alto de las mismas. Esto se debe a que, como comentábamos anteriormente, el paralelismo en CUDA se explota a través de la creación de muchos hilos ligeros; y dado que cada hilo se encarga del cálculo de una única iteración de la serie, si hay pocas iteraciones a calcular no podemos aprovechar este paralelismo.
- El incremento en el número de parejas de puntos a calcular supone una mejora en el speed-up obtenido, llegando a acercarse al obtenido por la rutina en OpenMP cuando el número de iteraciones a calcular es de 10000. Sin embargo, cuando dicho número de iteraciones es suficientemente alto, como ocurre en el último caso, el incremento en el número de puntos ya no supone ninguna mejora en el speed-up.

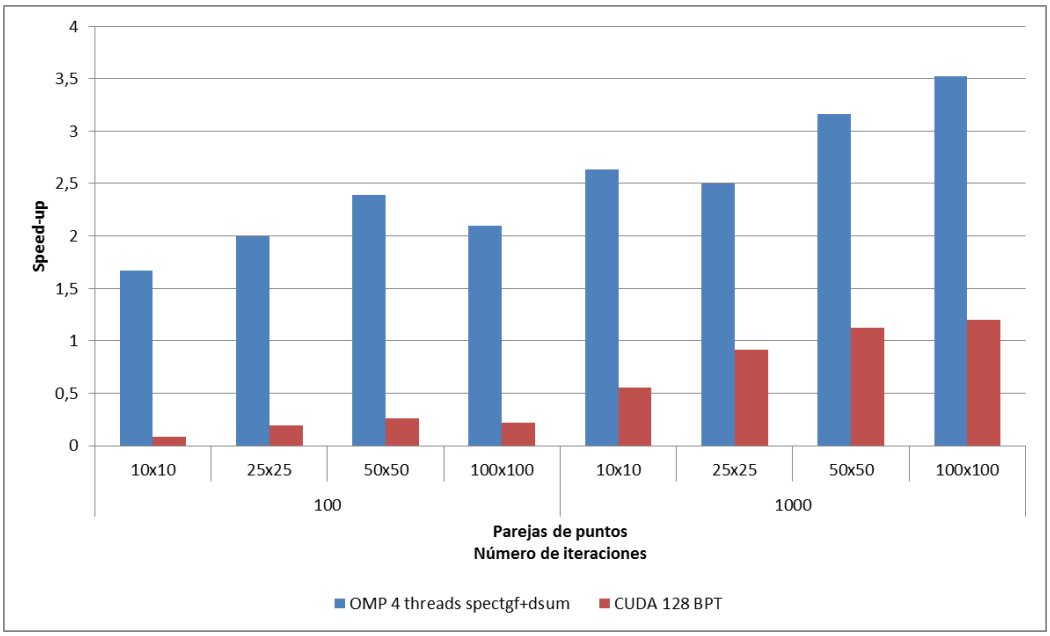

Figura 3.16 - Comparativa de los speed-up de las rutinas en OMP y CUDA con respecto a la versión secuencial optimizada en C+, para distinto número de parejas de puntos y un número bajo de iteraciones.

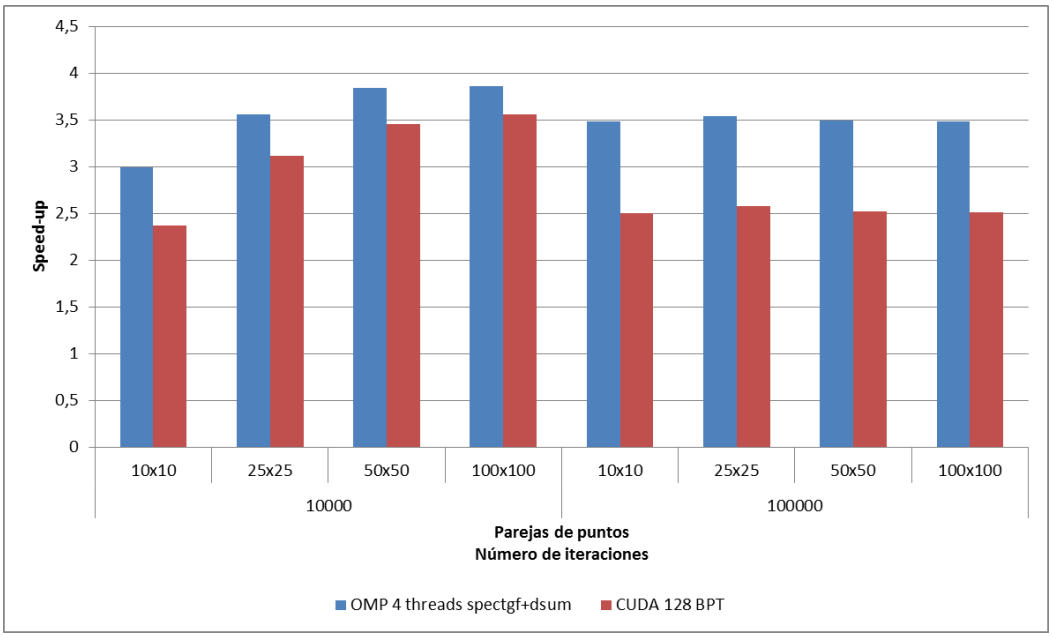

Figura 3.17 - Comparativa de los speed-up de las rutinas en OMP y CUDA con respecto a la versión secuencial optimizada en C++, para distinto número de parejas de puntos y un número alto de iteraciones.

Una vez concluido el estudio de las diferentes implementaciones para aprovechar el paralelismo a nivel de una única función de Green, trataremos de analizar el paralelismo de grano grueso que se nos presenta debido a la inclusión de varias parejas de puntos, para cada una de las cuales se puede calcular su función de Green en paralelo.

En las figuras 3.18 y 3.19 se muestran los fragmentos de pseudocódigo donde podemos ver la diferencia entre el esquema básico seguido en el caso anterior y el que nos encontramos en este nuevo nivel de paralelismo. En el primer caso nos encontramos un paralelismo a nivel de una única función de Green, donde cada unidad de proceso se encarga del cálculo de una o varias iteraciones del cálculo de una serie. Por otro lado, en el segundo caso tenemos un paralelismo a nivel de parejas de puntos, de forma que la carga de trabajo aumenta hasta el cálculo de una o varias funciones de Green.

```
foreach source and destination points i and j
       Invoke parallel routine for computing Green function between i and j
```

Figura 3.18 - Pseudocódigo en OpenMP del paralelismo a nivel de una única función de Green.

```
foreach source and destination points i and j in parallel
       Invoke sequential routine for computing Green function between i and j
```

Figura 3.19 - Pseudocódigo en OpenMP del paralelismo a nivel de varias funciones de Green.

Este aumento en la carga de trabajo supone un impedimento para el desarrollo de una versión que utilice únicamente CUDA para la paralelización, ya que incluso asignando a cada uno de los threads el cálculo de una única pareja de puntos supone una carga de trabajo mayor de la que se espera para un thread ligero de los que nos proporciona CUDA. Además, si recordamos que en la rutina desarrollada en CUDA seguíamos necesitando reservar una cantidad de memoria de 3 * num_iteraciones elementos para cada función de Green, y que la memoria de la unidad de proceso gráfico es limitada y compartida por todos los threads, tendríamos un nuevo problema adicional al intentar reservar varias veces dicha cantidad de memoria para cada pareja de puntos que quisiéramos calcular en paralelo.

Por tanto, en principio descartamos utilizar CUDA de forma única en este nivel de paralelismo, quedando únicamente la opción de OpenMP. Sin embargo, y como veremos más adelante, podemos aprovechar ambas tecnologías para desarrollar una versión que aproveche las capacidades de un entorno heterogéneo de computación CPU + GPU.

En cuanto a la versión desarrollada haciendo uso únicamente de OpenMP, hacemos uso de la rutina secuencial optimizada en C++ para el cálculo de una función de Green individual, y repartimos las parejas de puntos entre los procesadores disponibles utilizando una planificación dinámica. De esta forma evitamos el problema del balanceo de carga de trabajo que surge debido a que diferentes parejas de puntos pueden necesitar calcular un número diferente de iteraciones de la serie.

Para evaluar esta rutina haremos uso de diferentes tamaños de problema, variando el número de parejas de puntos así como el número de iteraciones a calcular. En la figura 3.20 mostramos el speed-up de esta nueva rutina con respecto a la versión secuencial optimizada en C++. Como podemos ver, para tamaños de problema suficientemente grandes el speed-up tiende al número de threads puestos en marcha, con una eficiencia cercana al 95%. Éstos speed-up se comparan en la figura 3.21 con los obtenidos con la versión anterior que explotaba el

paralelismo de grano fino. En este caso vemos que, excepto para un número pequeño de puntos a calcular, el comportamiento de esta nueva rutina es mejor que el de la anterior.

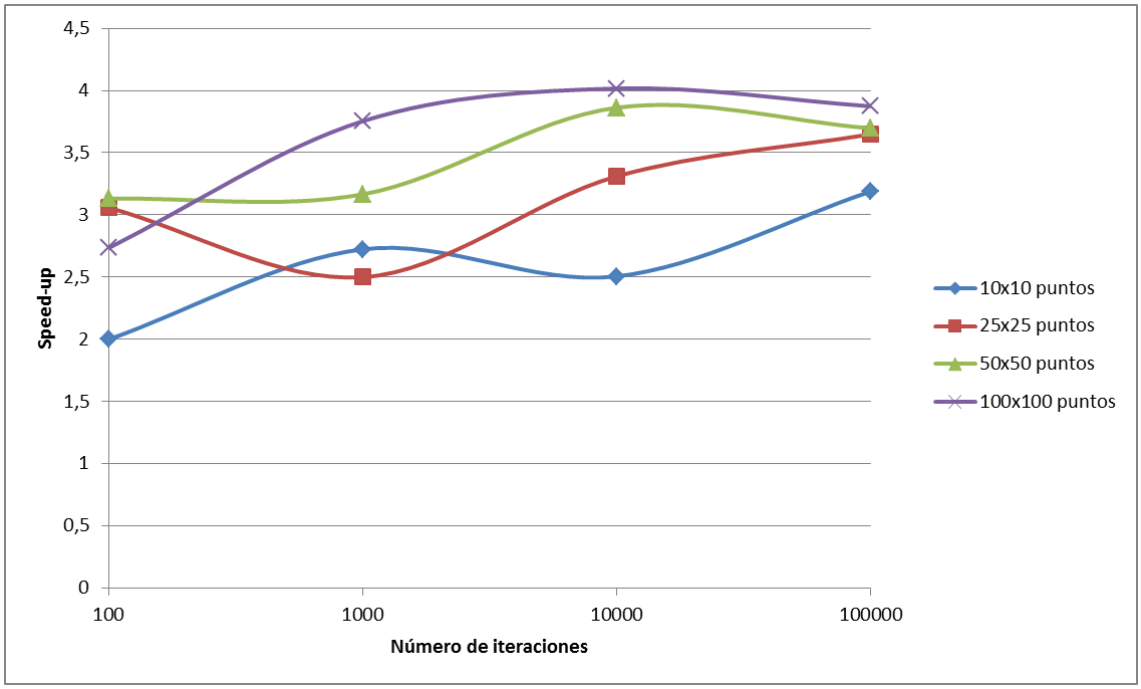

Figura 3.20 – Speed-up de la rutina en OpenMP que explota el paralelismo de grano grueso con respecto a la versión secuencial optimizada en C++.

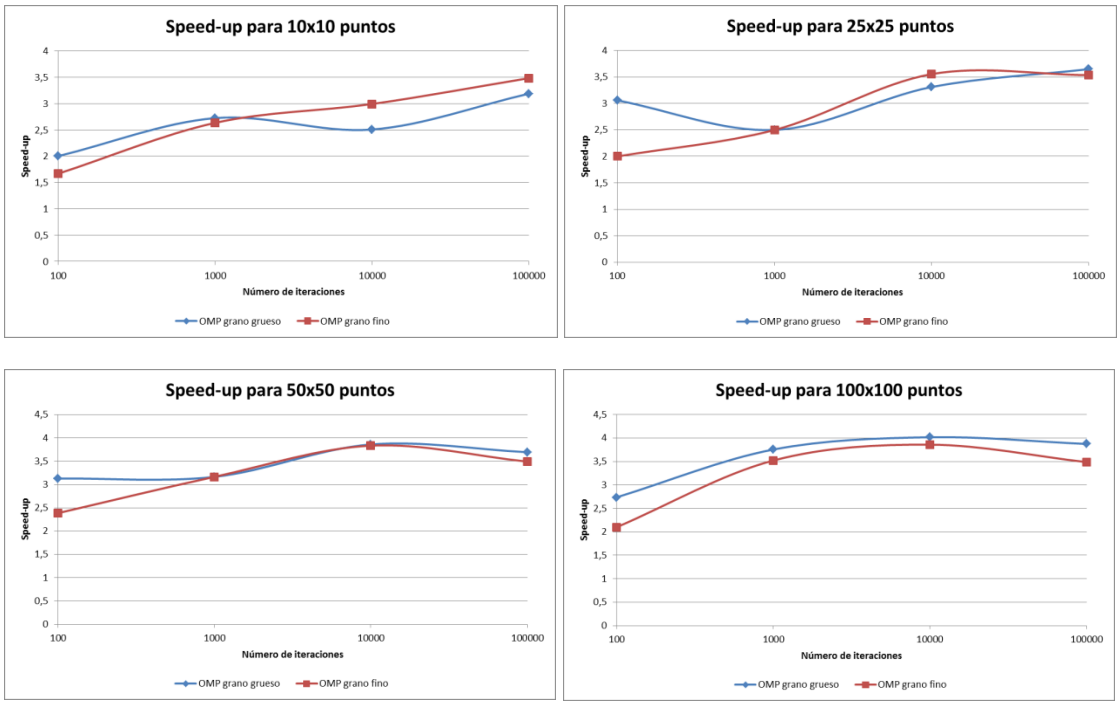

Figura 3.21 - Comparativa de los speed-up obtenidos por las rutinas en OpenMP que hacen uso del paralelismo de grano grueso (línea azul) y de grano fino (línea roja).

Una vez evaluada esta rutina y comprobado que ofrece el mejor comportamiento de entre todas las analizadas, podemos pensar en tratar de mejorarla combinándola con el uso de la GPU en un entorno heterogéneo de computación, tal como se comentaba anteriormente.

La idea es ampliar la rutina desarrollada en OpenMP de forma que parte de las funciones de Green a calcular se lleven a cabo en la GPU mientras el resto se calculan en la CPU. Para combinar ambas tecnologías, aprovecharemos las siguientes características de las mismas:

- En el modelo de ejecución de CUDA, una vez que desde el código de la CPU se invoca al kernel ejecutado en la GPU, se interrumpe la ejecución del mismo hasta que se retorna de dicho kernel.
- El modelo de OpenMP permite la distribución de la carga de trabajo de una región paralela de forma equitativa entre los threads de ejecución mediante un esquema de scheduling dinámico.

Así, la idea será crear una región paralela con el número máximo de threads disponibles, más uno. Este thread adicional se encargará únicamente de realizar la invocación al kernel de CUDA, de forma que aunque virtualmente hay un thread inactivo, seguimos teniendo el número máximo de threads disponibles trabajando. Por otra parte, dado que el tiempo de cálculo de una función de Green es distinto según se realice en la CPU o en la GPU, haremos uso de un esquema de planificación dinámica para balancear dicha carga.

En las figuras 3.22 y 3.23 mostramos los resultados obtenidos con esta rutina híbrida en comparación con los obtenidos con la mejor versión desarrollada hasta el momento. Como podemos observar seguimos teniendo un problema que nos encontrábamos con la rutina desarrollada únicamente mediante CUDA, que es una sobrecarga adicional de tiempo que hace empeorar el rendimiento cuando tenemos que calcular un número bajo de parejas de puntos o hacemos uso de pocas iteraciones.

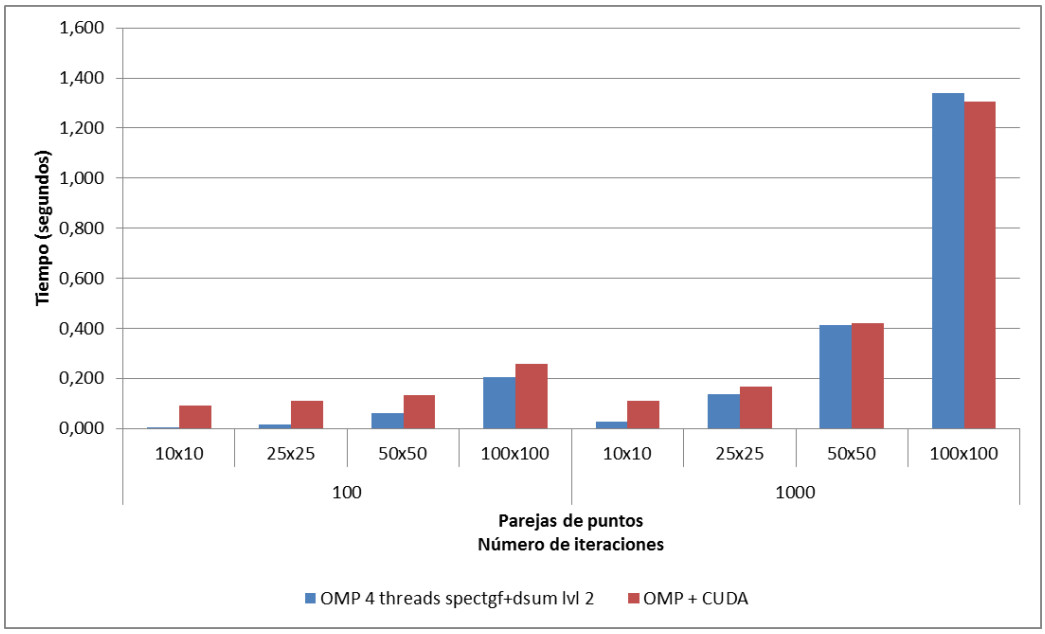

Figura 3.22 - Tiempo de ejecución de la rutina híbrida desarrollada en OpenMP y CUDA con respecto a la versión de OpenMP que explota el paralelismo de grano grueso. Comparativa para un número bajo de iteraciones a calcular.

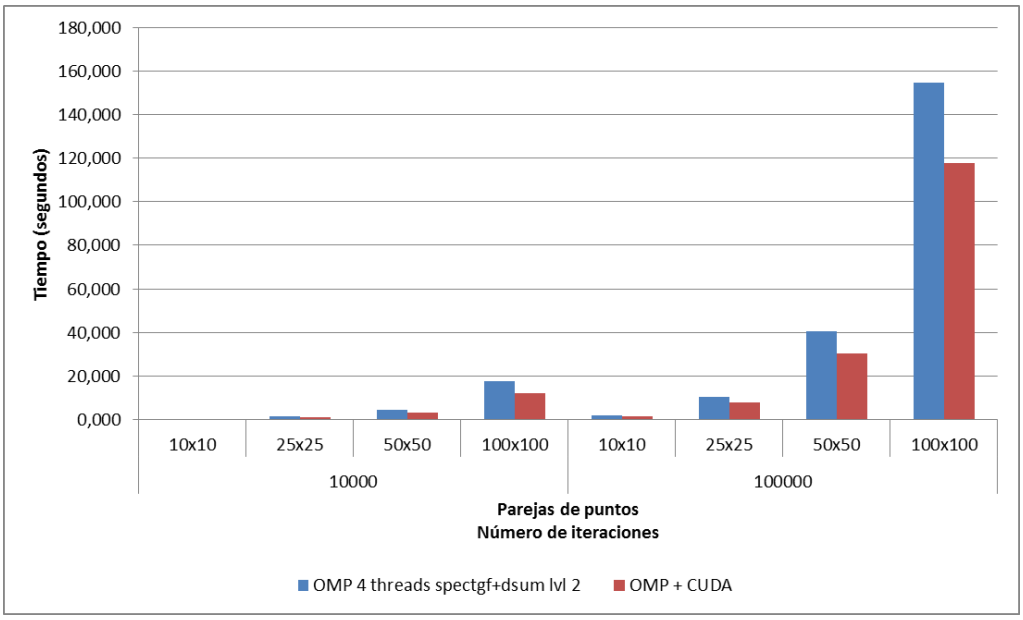

Figura 3.23 - Tiempo de ejecución de la rutina híbrida desarrollada en OpenMP y CUDA con respecto a la versión de OpenMP que explota el paralelismo de grano grueso. Comparativa para un número alto de iteraciones a calcular.

Sin embargo, para un número lo suficientemente alto de alguno de dichos parámetros, esta nueva versión mejora en torno a un 25% a la anterior. Por tanto, podríamos determinar la mejor versión a utilizar en base a los valores de los mismos.

Por último la tabla 3.1 muestra, a modo de resumen, una comparativa de los tiempos obtenidos para las mejores rutinas analizadas, desde la rutina secuencial original en Fortran hasta esta última basada en OMP + CUDA. Los resultados se han resaltado con una escala de colores desde el verde, que representa los tiempos de ejecución más bajos, hasta el rojo para los más elevados.

modos	puntos	Fortran	C++ optimizado (g++)	OMP grano fino	OMP grano grueso	OMP + CUDA
100	10x10	0,015	0,010	0,006	0,005	0,093
	25x25	0,074	0,052	0,026	0,017	0,109
	50x50	0,261	0,191	0,080	0,061	0,135
	100x100	0,833	0,558	0,266	0,204	0,257
1000	10x10	0,139	0,079	0,030	0,029	0,109
	25x25	0,635	0,345	0,138	0,138	0,168
	50x50	2,323	1,314	0,415	0,415	0,419
	100x100	9,091	5,023	1,427	1,338	1,306
10000	10x10	1,615	0,772	0,258	0,308	0,304
	25x25	9,867	4,340	1,221	1,311	0,966
	50x50	39,207	17,725	4,619	4,589	3,217
	100x100	156,87	70,798	18,349	17,627	12,003
100000	10x10	14,445	6,260	1,797	1,963	1,581
	25x25	89,786	38,688	10,944	10,604	7,758
	50x50	360,826	149,817	42,868	40,528	30,13
	100x100	1435,542	598,326	171,615	154,463	117,535

Tabla 3.1 - Comparativa de los tiempos de ejecución de las distintas rutinas desarrolladas.

4. Funciones de Green bidimensionales de la guía rectangular.

Una vez concluido el estudio de las funciones de Green presentes en una guía de placas paralelas en una dimensión, pasaremos a analizar en este capítulo las funciones de Green bidimensionales de una guía rectangular.

Al igual que en el caso anterior, se dispone de un sistema de referencia sobre el que se sitúa una guía rectangular, dentro de la cual se sitúa por un lado un punto fuente, y por otro lado un plano de observación. Y sobre este plano se fijan un número de puntos sobre los cuales calcular las funciones de Green con respecto al punto fuente.

Para el cálculo de estas funciones de Green existen diversos métodos, de entre los cuales nos interesa el método de Ewald que es el mismo utilizado anteriormente para el caso unidimensional. En este caso dicho método descompone la función de Green como suma de dos términos, uno calculado en el dominio espectral y el otro en el dominio espacial. Para la parte espectral del mismo se necesitan evaluar un número de términos o modos a lo largo de un solo eje, mientras que para el cálculo de la parte espacial se utiliza el método de imágenes, habiendo de evaluar las funciones a lo largo de imágenes en dos ejes diferentes.

Aunque nos centraremos en un método concreto de resolución de funciones de Green para el estudio de diferentes técnicas de paralelismo, la metodología empleada se puede aplicar a otros métodos utilizados en diferentes problemas de electromagnetismo computacional, para los que los resultados obtenidos serían similares a los expuestos a lo largo de este capítulo.

4.1. Análisis del algoritmo secuencial.

Comenzaremos, al igual que en el capítulo anterior, analizando el código fuente del algoritmo secuencial proporcionado por el Grupo de Electromagnetismo Computacional de la Universidad Politécnica de Cartagena, a fin de dar una visión de las distintas partes del mismo junto con el coste computacional asociado a las mismas, y tratar de establecer la relación entre el código fuente y lo dicho anteriormente sobre el método de Ewald.

El código de partida se encuentra en lenguaje Fortran, y está dividido en los siguientes ficheros:

- GF_wg_rectangular.f90: se corresponde con la rutina principal del programa, desde la cual se lee el fichero de configuración con los parámetros de cálculo de las funciones de Green (principalmente el número de puntos del plano de observación sobre el que calcular las mismas, así como el número de modos e imágenes para los sumatorios de las series de las partes espectral y espacial de dichas funciones). Una vez leídos dichos parámetros, se invoca a las rutinas de inicialización y por último a la función principal encargada del cálculo de las funciones de Green.

- Funciones de inicialización:
    - Modos_grecta.f90: se encarga de calcular una cantidad de modos dada como parámetro, los cuales se corresponden con los primeros modos que se excitarían en la guía rectangular, ordenando los números de onda obtenidos de forma creciente. El orden de complejidad de esta rutina está determinado por el número de modos a calcular especificado como parámetro.
    - Reduce_modos_grecta.f90: se llama inmediatamente después de la rutina anterior, y se encarga de quedarse solo con los modos transversales electromagnéticos (modos TE) obtenidos por la misma. Al igual que esta otra, tiene un orden de complejidad proporcional al número de modos especificados.
    - Ini_Ewald.f90: determina el valor del parámetro de división (splitting parameter [20]) utilizado posteriormente en el cálculo de las funciones de Green, y su coste computacional puede considerarse nulo.
- GF_wg_ewald_xy_ord.f90: se trata de la rutina principal de cálculo de las funciones de Green de la guía rectangular, de forma que para cada punto del plano de observación se calcula la función de Green asociada como suma de dos partes:
    - Una parte espectral, para la cual se utilizan los modos calculados anteriormente en las rutinas de inicialización, y que por tanto tiene un orden de complejidad proporcional a dicho número de modos.
    - Una parte espacial, evaluada en las imágenes a lo largo de los ejes x e y, con un coste computacional proporcional al producto del número de imágenes en cada eje.

En resumen, en la figura 4.1 presentamos un esquema en pseudocódigo que muestra cómo queda organizado el código así como la secuencia de invocación entre las distintas funciones. Los parámetros npunx y npuny se corresponden al número de puntos calculados en el plano de observación, en los ejes x e y respectivamente. En cuanto a los parámetros nmodos, nimag y mimag se corresponden al número de modos a calcular en la parte espectral, y el número de imágenes en los ejes x e y, respectivamente, donde se evalúa la parte espacial de la serie.

Las funciones spectral_gf y spatial_gf, además de operaciones con números complejos y en coma flotante, contienen cada una un bucle más interno para el cálculo de las funciones complementarias de error utilizadas. El número de iteraciones a desarrollar en dichos bucles está determinado por un parámetro externo adicional, newald, que se corresponde con el número de dígitos de precisión a considerar en el cálculo de dichas funciones de error. Sin embargo, al tratarse de un parámetro con una variación menor que el resto (posiblemente en el rango [5, 15]) se ha considerado a lo largo de este estudio como el valor constante 10, por lo que a efectos prácticos consideraremos que las funciones spectral_gf y spatial_gf tienen un orden de complejidad constante k, siendo k el número de operaciones en coma flotante realizadas dentro de las mismas.

```
function main()
       modos_grecta(nmodos)
       reduce_modos_grecta(nmodos)
       ini_Ewald()
       GF_wg_ewald_xy_ord(npunx, npuny, nmodos, nimag, mimag)
              para cada punto m de 1..npunx
                     para cada punto n de 1..npuny
                            /* Parte espectral */
                            para cada modo nmodoloop de 1..nmodos
                                   GF[m][n] += spectral_gf(nmodoloop)
                            fin_para
                            /* Parte espacial */
                            para cada imagen mima de -mimag..mimag
                                   para cada imagen nima de –nimag..nimag
                                          GF[m][n] += spatial_gf(mima, nima)
                                   fin_para
                            fin_para
                     fin_para
              fin_para
       fin_para
end_function
```

Figura 4.1 - Esquema en pseudocódigo del algoritmo secuencial de cálculo de las funciones de Green bidimensionales.

Por otra parte, el número de modos utilizados en el cálculo de la parte espectral de las funciones de Green también se ha considerado con un valor constante de 100, principalmente debido a la dificultad para modificar el mismo al estar relacionado con otros parámetros. Además, ya que el rango de valores de nmodos, nimag y mimag es similar, podríamos considerar que el orden de complejidad del cálculo de una función de Green en un determinado punto viene dado por la siguiente ecuación:

$$O(nmodos, nimag, mimag) = O_{spectral}(nmodos) + O_{spatial}(nimag * mimag)$$
$$= O(nimag * mimag)$$

Y por tanto podemos despreciar el tiempo que supone el cálculo de la parte espectral. Es interesante destacar el hecho de que ésto se debe al uso de los modos ordenados proporcionados por la rutina gmodos_recta. En caso contrario, y tal como se ha observado en versiones antiguas de los códigos proporcionados, para el cálculo de la parte espacial tendríamos un código similar al mostrado en la figura 4.2. Como vemos en ella se trata del mismo esquema seguido en el cálculo de la parte espacial, por lo que las técnicas que analizaremos durante el estudio serían aplicables también en caso de querer utilizar dicha versión antigua para calcular la parte espectral.

```
/* Parte espectral */
Para cada mmodo mmodoloop de 1..nmodos
       Para cada nmodo nmodoloop de 1..mmodos
              GF[m][n] += spectral_gf(nmodoloop, mmodoloop)
       Fin_para
Fin_para
```

Figura 4.2 – Pseudocódigo del cálculo de la parte espectral de las funciones de Green bidimensionales mediante el uso de modos en los ejes x e y, dados por los parámetros nmodos y mmodos.

Una vez analizadas las distintas partes del código así como el flujo de ejecución entre las mismas, pasamos a desarrollar el código en lenguaje C++ que utilizaremos durante el desarrollo de este capítulo. Es interesante destacar el hecho de que, a diferencia de lo ocurrido en el capítulo anterior, el código desarrollado en C++ y compilado con el compilador de GNU g++ optimizado con la opción –O3 es varias veces más lento que el correspondiente en Fortran. Y no solo hay diferencia en cuanto al tiempo de ejecución, sino que los resultados obtenidos difieren incluso para órdenes de magnitud que podemos considerar bajos, como 1E-6. Sin embargo, esto último se debe a la activación de uno o varios de los flags de optimización del compilador, ya que al compilar sin optimizaciones los resultados obtenidos son los correctos.

En la tabla 4.1 podemos comparar los tiempos de ejecución obtenidos para la versión secuencial proporcionada en Fortran, y la versión en C++ desarrollada a partir de la misma y compilada mediante el compilador de GNU. Como podemos ver, en general los resultados obtenidos con el compilador de GNU son casi 4,5 veces más lentos que los de Fortran.

imágenes	puntos	Fortran (ifort)	C++ (g++)	speed down
100	10x10	0,148	0,487	3,290
	25x25	0,689	2,680	3,889
	50x50	2,584	10,616	4,108
	100x100	9,986	42,372	4,243
1000	10x10	0,859	3,603	4,194
	25x25	5,018	22,272	4,438
	50x50	19,659	88,853	4,519
	100x100	78,710	355,210	4,512
10000	10x10	7,432	33,351	4,487
	25x25	46,318	208,091	4,492
	50x50	185,205	832,248	4,493
	100x100	760,954	3420,284	4,494
100000	10x10	74,676	330,927	4,431
	25x25	459,856	2067,221	4,495
	50x50	1822,058	8195,091	4,497

Tabla 4.1 - Comparativa del tiempo de ejecución de la versión secuencial original en Fortran y la equivalente en C++ compilada mediante g++.

Por tanto, antes de pasar a estudiar las posibilidades de paralelismo del problema se ha hecho uso del compilador homónimo de C++ al utilizado con Fortran, el Intel C++ Compiler (icpc). En este caso tanto los tiempos de ejecución como los resultados obtenidos son los esperados, con una pequeña mejora en el rendimiento de la versión de C++ respecto a la de Fortran que podemos observar en la tabla 4.2.

imágenes	puntos	Fortran (ifort)	C++ (icpc)	speed up
100	10x10	0,148	0,162	0,913
	25x25	0,689	0,729	0,945
	50x50	2,584	2,767	0,933
	100x100	9,986	10,945	0,912
1000	10x10	0,859	0,674	1,274
	25x25	5,018	4,072	1,232
	50x50	19,659	16,209	1,212
	100x100	78,710	65,489	1,201
10000	10x10	7,432	5,794	1,282
	25x25	46,318	36,568	1,266
	50x50	185,205	144,560	1,281
	100x100	760,954	576,456	1,320
100000	10x10	74,676	56,974	1,310
	25x25	459,856	354,847	1,295
	50x50	1822,058	1491,650	1,221

Tabla 4.2 - Comparativa del tiempo de ejecución de la versión secuencial original en Fortran y la equivalente en C++ compilada mediante icpc.

Por último, y antes de comenzar a estudiar el paralelismo, utilizaremos la herramienta callgrind para realizar un perfil de ejecución del código secuencial que nos permita identificar las partes del código que consumen un mayor tiempo de ejecución y que, por tanto, son más susceptibles de ser paralelizadas. Además, también nos aportará información sobre las instrucciones más costosas de cada una de dichas partes de código, de forma que podamos tenerlo en cuenta para tratar de optimizar las mismas.

En la figura 4.3 podemos ver el perfil de ejecución obtenido para el caso con 1 único punto en el plano de observación y 100000 de imágenes espaciales a calcular, donde podemos observar cómo el 99.5% del tiempo total se consume en el cálculo de la parte espacial de las funciones de Green.

También se muestra en la figura 4.4 el perfil obtenido para una ejecución con 10x10 puntos en el plano de observación y un número menor de imágenes, concretamente 1000. En este caso prácticamente el 100% del tiempo de ejecución se corresponde con el cálculo de las funciones de Green mediante el método de Ewald, pero casi un 7% del mismo se utiliza en la parte espectral de dicho método.

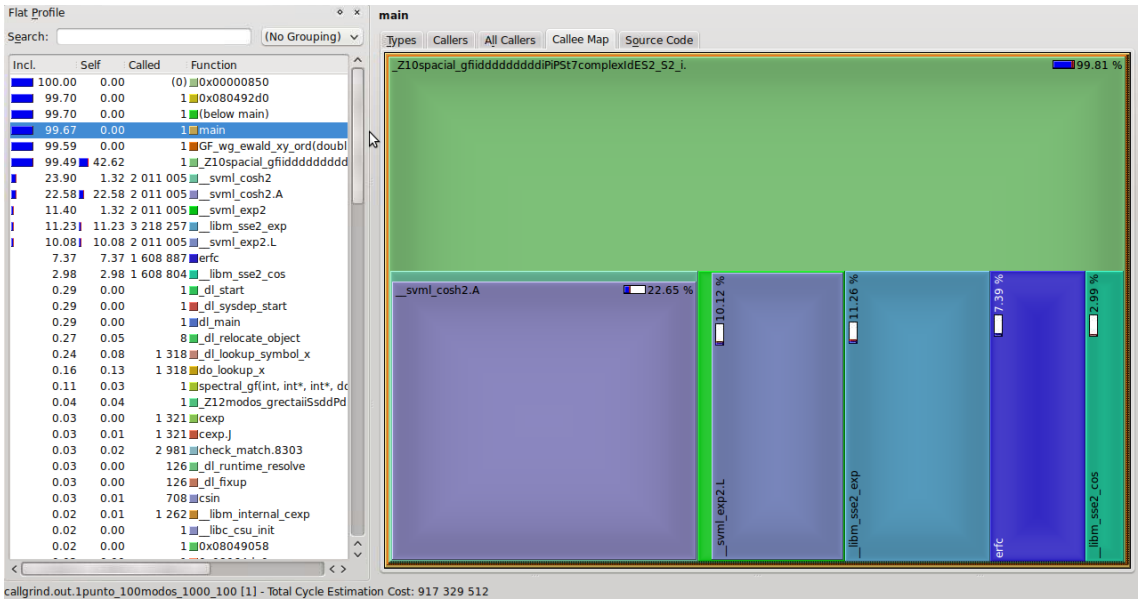

Figura 4.3 - Perfil de ejecución para un único punto en el plano de observación y 100000 de imágenes espaciales a calcular.

Así pues, vemos que tal como se esperaba la parte espacial es la parte más costosa del programa, y por tanto aquella en la que centraremos la aplicación de las técnicas de paralelismo. Sin embargo, también estudiaremos el efecto de aplicar paralelismo a la parte espectral mediante técnicas que no añadan un alto overhead, como OpenMP, para ver si los casos como este donde esa parte consume algo menos del 10% del tiempo total se ven afectados positivamente.

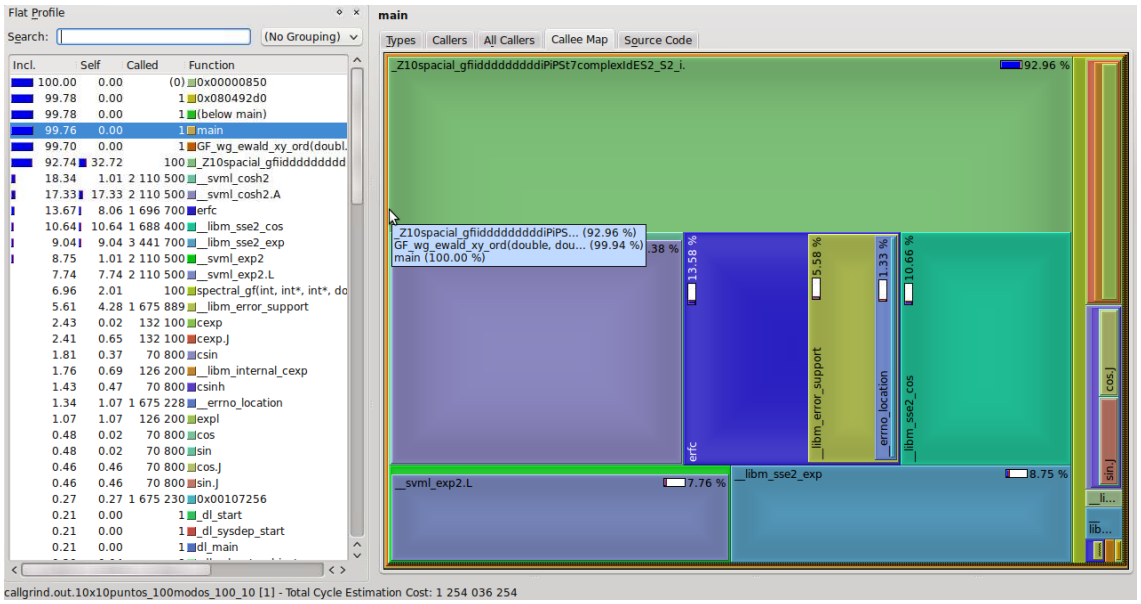

Figura 4.4 - Perfil de ejecución para 10x10 puntos en el plano de observación y 1000 imágenes espaciales a calcular.

4.2. Paralelización de una única función de Green.

Una vez analizado el funcionamiento del algoritmo secuencial y determinada la parte del código con más carga computacional trataremos de reducir el tiempo de ejecución de la misma utilizando distintas tecnologías para su paralelización.

Sean n y m el número de puntos en los ejes x e y del plano de observación, y nmod, nimag y mimag el número de modos e imágenes en los ejes x e y de dicho plano, respectivamente, el tiempo de ejecución del algoritmo secuencial nos queda:

$$t(n,m,nmod,nimag,mimag) = \sum_{1}^{n}\sum_{1}^{m}\left(\sum_{1}^{nmod} k + \sum_{1}^{nimag}\sum_{1}^{mimag} k\right)$$
$$= n * m * k * (nmod + nimag * mimag) \approx n * m * k * nimag * mimag$$

donde k denota el coste de las operaciones con números reales y complejos del cálculo de las partes espectral y espacial del método de Ewald. Como era de esperar, el coste computacional del problema bidimensional es de un orden de magnitud superior al del problema unidimensional. Así, además de utilizar OpenMP y CUDA para el estudio del problema bidimensional, también haremos uso de MPI, ya que posiblemente el overhead introducido por las comunicaciones sea inferior al beneficio obtenido por el paralelismo.

4.2.1. OpenMP.

En primer lugar analizaremos la solución más sencilla, que es la basada en OpenMP. La idea es repartir el cálculo de la serie de una única función de Green en un único punto entre el número de unidades de proceso disponibles, de forma que el resultado final será la suma de las partes calculadas por cada una de ellas. Así, puesto que cada hilo tendrá que calcular parte de una misma función de Green, es necesario disponer de un mecanismo que evite la modificación del valor de la misma por dos hilos al mismo tiempo. Dos posibles opciones son las siguientes:

- Utilizar una sección crítica mediante el pragma que nos ofrece el API de OpenMP. Sin embargo, esto supone una sincronización por cada iteración entre todos los hilos puestos en marcha, con un coste de hasta $t_s = k_s * nimag * imag$.
- Utilizar una variable auxiliar privada a cada hilo para acumular la suma parcial de su parte de la serie, de forma que el resultado final es la suma de las sumas parciales de cada hilo. A su vez, esto podemos hacerlo bien de forma automática, mediante un pragma reduction de OpenMP, o realizando la suma de forma manual una vez que todos han calculado su parte.

En la tabla 4.3 se comparan los tiempos de ejecución utilizando cada una de las estrategias anteriores. También se muestran los speed-up conseguidos por las mismas en la figura 4.5.

imágenes	puntos	C++ (icpc)	OMP pragma critical	OMP pragma reduction	OMP reduccion manual
100	100	0,162	0,069	0,054	0,072
	625	0,729	0,340	0,228	0,243
	2500	2,767	1,118	0,754	0,785
	10000	10,945	4,133	2,847	2,889
1000	100	0,674	0,327	0,235	0,255
	625	4,072	1,701	1,128	1,139
	2500	16,209	6,680	4,267	4,377
	10000	65,489	26,849	16,944	17,447
10000	100	5,794	2,298	1,406	1,467
	625	36,568	13,818	8,517	8,536
	2500	144,56	55,333	33,816	35,116
	10000	576,456	221,498	135,392	139,091
100000	100	56,974	21,876	13,406	13,823
	625	354,847	136,283	83,526	84,129
	2500	1491,65	546,453	335,502	343,848

Tabla 4.3 - Tiempos de ejecución obtenidos para la versión desarrollada en OpenMP haciendo uso de las tres estrategias para llevar a cabo la modificación de datos compartidos.

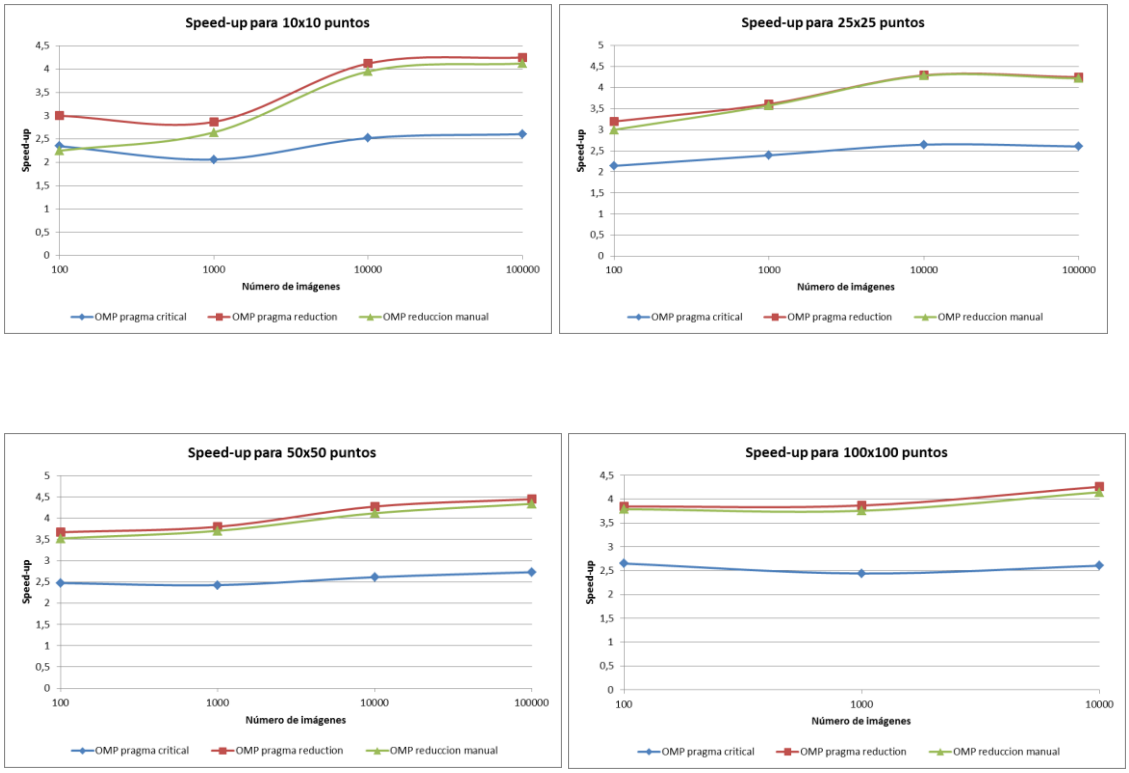

Figura 4.5 – Speed up conseguido por las rutinas desarrolladas en OpenMP, cuyos tiempos de ejecución podemos ver en la tabla 4.3, respecto a la versión secuencial en C++.

Como vemos, ambas estrategias de reducción producen un speed-up cercano al número de threads utilizados, siendo en algunos casos incluso ligeramente superior (speed-up superlineal), lo cual probablemente sea debido a una mejor distribución de los datos en la memoria caché. Además, la diferencia entre hacer uso del pragma proporcionado por OpenMP para la reducción o llevar a cabo la misma de forma manual no supone una diferencia de tiempo considerable, si bien es cierto que la primera es ligeramente más eficiente. Por el contrario, y tal como era de esperar, la versión que hace uso de secciones críticas para el acceso a variables compartidas es notablemente más lenta que las anteriores, con un speed-up medio de 2.5 que supone una eficiencia en torno al 62.5%

No obstante, además de los tiempos de ejecución también es interesante observar la precisión de los resultados obtenidos, y es que aunque el uso del pragma reduction es la solución más eficiente, obtiene pequeñas pérdidas de precisión del orden de magnitud de 1E-24, que podrían ser significativas dependiendo del problema y la precisión deseada.

4.2.2. CUDA.

Una vez concluido el estudio con OpenMP pasamos a analizar las posibilidades de paralelismo aplicables mediante el paradigma CUDA. Comenzaremos por aplicar el paralelismo únicamente a la parte espacial del cálculo de las funciones de Green, al tratarse de la parte más costosa, y finalmente trataremos de extender el mismo a la parte espectral para ver si se obtiene alguna mejora más.

Si recordamos, el código para el cálculo de la parte espacial tenía la estructura mostrada en la figura 4.6. Puesto que la filosofía en que se basan los threads CUDA para obtener un buen rendimiento consiste en la ejecución simultánea de múltiples threads ligeros, una buena opción podría ser poner en marcha tantos como número de imágenes a calcular, de forma que cada hilo se encargaría de realizar el cálculo de spatial_gf(mimag, nimag).

Por tanto, utilizaremos una matriz auxiliar para almacenar la suma parcial calculada por cada thread, de forma que una vez finalizada la invocación al kernel tendremos que sumar los valores de la misma. Aunque podríamos utilizar un nuevo kernel para realizar dicha suma mediante un algoritmo de reducción en árbol, por ejemplo, en principio optaremos por realizar la misma de forma secuencial.

```
/* Parte espacial */
para cada imagen mimag de -Mimag..Mimag
       para cada imagen nimag de -Nimag..Nimag
              GF[m][n] += spatial_gf(mimag, nimag)
       fin_para
fin_para
```

Figura 4.6 - Pseudocódigo del cálculo de la parte espacial de las funciones de Green bidimensionales mediante el método de Ewald.

A diferencia de lo que ocurría anteriormente en la paralelización mediante CUDA de las funciones de Green unidimensionales donde la opción más sencilla consistía en utilizar un único thread block con tantos threads como número de iteraciones a calcular, en este caso podemos aprovechar la naturaleza bidimensional del problema para establecer una relación entre el número de imágenes en cada eje y el número de thread blocks y de threads dentro de los mismos. En concreto crearemos un bloque de threads por cada imagen en el eje x (mimag), y dentro del mismo pondremos en marcha un thread por cada imagen en el eje y (nimag), tal como se ilustra en la figura 4.7.

```
__global__ void spectral_gf_cuda_kernel(…)
       mima = blockIdx.x          // mima entre [0, 2*mimag + 1)
       nima = threadIdx.x         // nima entre [0, 2*nimag + 1)
       mima -= gridDim.x / 2      // mima entre [-mimag, mimag]
       nima -= blockDim.x / 2     // nima entre [-nimag, nimag]

       Calcular spatial_gf(mimag, nimag)
```

Figura 4.7 - Código del kernel desarrollado en CUDA para el cálculo de la parte espectral de las funciones de Green bidimensionales mediante el método de Ewald.

Los tiempos de ejecución obtenidos para esta rutina se muestran en la tabla 4.4, y sus respectivos speed up los podemos observar en la gráfica de la figura 4.8.

imágenes	puntos	C++ (icpc)	CUDA
100	10x10	0,162	0,155
	25x25	0,729	0,453
	50x50	2,767	1,476
	100x100	10,945	5,463
1000	10x10	0,674	0,167
	25x25	4,072	0,474
	50x50	16,209	1,492
	100x100	65,489	5,636
10000	10x10	5,794	0,223
	25x25	36,568	0,780
	50x50	144,560	2,737
	100x100	576,456	10,59
100000	10x10	56,974	0,940
	25x25	354,847	5,361
	50x50	1491,650	21,022

Tabla 4.4 - Tiempos de ejecución de la rutina desarrollada en CUDA para distintos tamaños de problema.

Como podemos ver en la gráfica los resultados son bastante positivos, con mejoras de hasta 70 veces en el tiempo de ejecución para los casos con mayor número de imágenes a calcular, y con tendencias al alza que nos hacen suponer que podrían ser superiores en caso de utilizar más imágenes para el cálculo de la parte espacial de las funciones de Green. Por otra parte también vemos como la mejora es mayor cuantos más puntos del plano de observación se calculen, aunque en este caso parece que sólo es hasta 50x50 puntos.

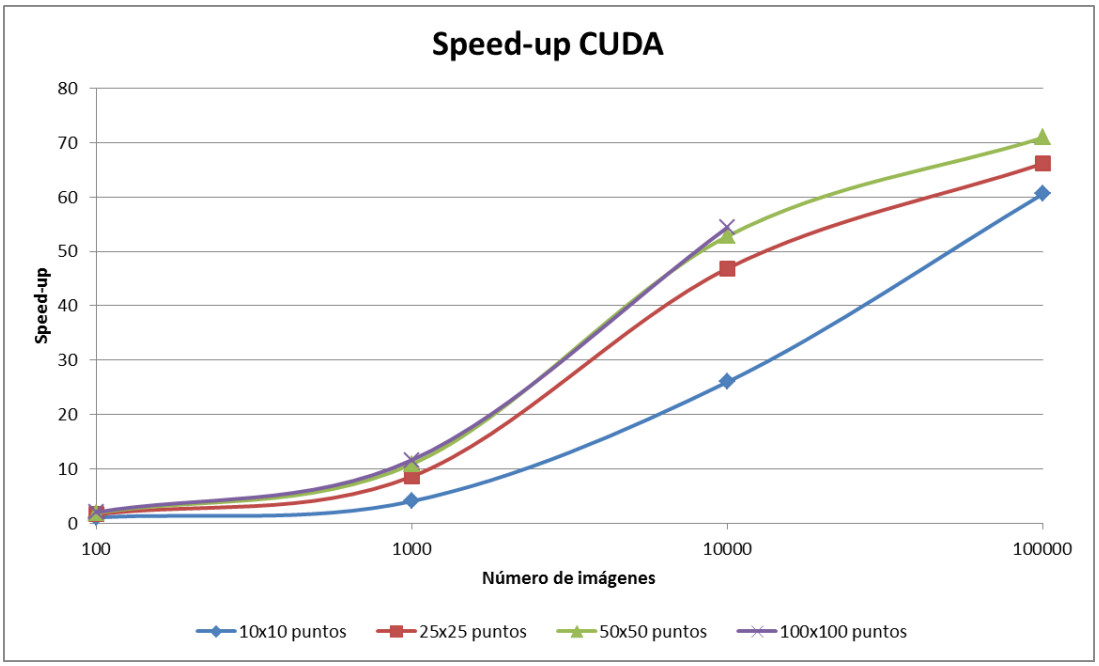

Figura 4.8 - Speed up conseguido por la rutina desarrollada en CUDA respecto a la versión secuencial en C++.

En cualquier caso debemos remarcar el hecho de que estos resultados se han obtenido con una GPU con CUDA Compute Capability 1.1, que al no disponer de capacidad para trabajar con números en coma flotante de doble precisión (incluidos a partir de la versión 1.3) nos han forzado a trabajar con datos de tipo float, con lo que se ha perdido cierta precisión que, de nuevo, puede ser relevante dependiendo del problema.

Viendo la gran mejora obtenida al paralelizar el cómputo de la parte espacial del problema, podríamos pensar en realizar la parte espectral también en la GPU. El código de la misma, si recordamos, tenía la forma mostrada en la figura 4.9.

Así pues, la idea es similar a lo explicado anteriormente, ya que de nuevo se trata de utilizar un thread CUDA para el cálculo de cada iteración, y almacenar las sumas parciales en un array auxiliar, que a diferencia del anterior es unidimensional. Además, tal y como comentábamos al principio del capítulo, el número de iteraciones o modos a calcular lo habíamos fijado en 100, por lo que utilizaremos un único thread block que contendrá a los 100 hilos a crear.

```
/* Parte espectral */
para cada modo nmodoloop de 1..Nmodos
       GF[m][n] += spectral_gf(nmodoloop)
fin_para
```

Figura 4.9 - Pseudocódigo del cálculo de la parte espectral de las funciones de Green bidimensionales mediante el método de Ewald.

En la tabla 4.5 se muestran los tiempos obtenidos para esta nueva versión en CUDA donde paralelizamos el cálculo tanto de la parte espectral como de la espacial. Como vemos prácticamente no hay diferencia entre una versión y la otra, aunque en general podemos decir que esta nueva empeora ligeramente los tiempos de ejecución. Sin embargo, en caso de no fijar el parámetro del número de modos a utilizar para el cálculo de la parte espectral sí que podríamos conseguir una mejora significativa con la paralelización de dicha parte.

imágenes	puntos	CUDA spatial	CUDA spectral+spatial	diferencia
100	10x10	0,155	0,159	0,004
	25x25	0,453	0,455	0,002
	50x50	1,476	1,499	0,023
	100x100	5,463	5,719	0,256
1000	10x10	0,167	0,163	-0,004
	25x25	0,474	0,474	0,000
	50x50	1,492	1,541	0,049
	100x100	5,636	5,879	0,243
10000	10x10	0,223	0,221	-0,002
	25x25	0,780	0,780	0,000
	50x50	2,737	2,810	0,073
	100x100	10,590	10,954	0,364
100000	10x10	0,940	0,954	0,014
	25x25	5,361	5,402	0,041
	50x50	21,022	21,287	0,265
	100x100	83,744	84,882	1,138

Tabla 4.5 - Tiempos de ejecución de las rutinas desarrolladas en CUDA.

4.2.3. MPI.

Por último nos queda por analizar el uso de MPI como mecanismo para desarrollar una versión paralela enfocada, pero no limitada, a sistemas de memoria distribuida. En este caso debemos tener en cuenta que uno de los aspectos más costosos de este tipo de sistemas son las comunicaciones entre nodos, que pueden ser de varios órdenes de magnitud superiores al coste de una operación en la CPU. Debido a esto nos centraremos en la parte espacial de las funciones de Green, ejecutando la parte espectral en secuencial a fin de evitar un incremento en el tiempo de ejecución que probablemente no compense a la mejora obtenida por la paralelización del código.

Dicho esto, para la versión desarrollada en MPI hemos seguido la misma idea que en el caso de OpenMP, con la diferencia de que en este caso el reparto del trabajo entre los distintos nodos se hace de forma estática, asignando a cada uno un número consecutivo de iteraciones a calcular. Como cada nodo realiza una suma parcial, se hace necesario nuevamente sumar los valores producidos por cada nodo para obtener la suma total. Para ello, al igual que en OpenMP, podemos bien realizar una reducción manual de los datos haciendo que todos los procesos le envíen su suma parcial a un nodo, o bien utilizar la rutina MPI_Reduce proporcionada por el API de MPI. En este caso optamos por utilizar directamente esta última, a la vista de los resultados obtenidos con OpenMP.

Para la compilación se ha hecho uso de la distribución MPICH2, y cabe destacar el uso del parámetro –CXX=icpc al compilar mediante mpic++, ya que de otro modo se utilizaría el compilador por defecto, que en nuestro caso es el de GNU, y que como ya vimos anteriormente supone un incremento sustancial del tiempo de ejecución.

En la tabla 4.6 presentamos los resultados obtenidos haciendo uso de esta versión desarrollada en MPI, y en la figura 4.10 mostramos la gráfica con los speed-up logrados por la misma respecto a la versión secuencial en C++. Como vemos el comportamiento es bastante similar al observado en el caso de OpenMP, con un speed-up asintótico de algo más del número total de threads utilizados, y que probablemente también se deba a la gestión de los datos en las memorias caché.

imágenes	puntos	C++ (icpc)	MPI (mpich2)
100	100	0,162	0,392
	625	0,729	0,745
	2500	2,767	1,882
	10000	10,945	6,386
1000	100	0,674	0,554
	625	4,072	1,637
	2500	16,209	5,434
	10000	65,489	20,789
10000	100	5,794	1,773
	625	36,568	9,431
	2500	144,560	35,167
	10000	576,456	144,722
100000	100	56,974	13,882
	625	354,847	84,873
	2500	1491,650	344,977

Tabla 4.6 - Tiempos de ejecución de la rutina desarrollada en MPI.

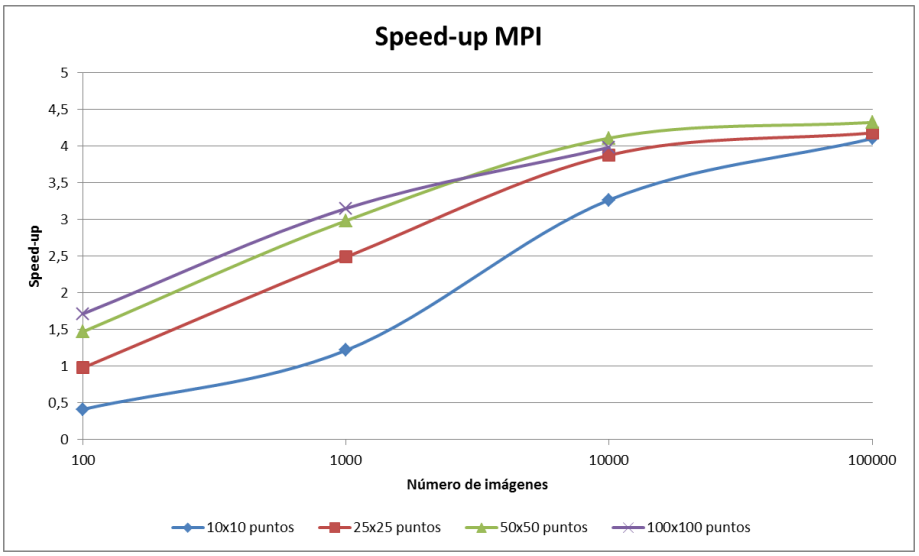

Figura 4.10 - Speed-up de la rutina desarrollada en MPI con respecto a la versión secuencial en C++.

4.2.4. Resumen.

Finalmente, a modo de resumen mostramos en la tabla 4.7 una comparativa de las distintas versiones estudiadas a lo largo del capítulo a modo de resumen. Los resultados se han resaltado con una escala de colores desde el verde, que representa los tiempos de ejecución más bajos, a rojo para los tiempos más elevados. Como podemos observar, para problemas con una baja carga computacional los mejores resultados vienen del uso de OpenMP, al introducir un overhead relativamente bajo comparado con el de CUDA o MPI. Sin embargo, en cuanto la carga de trabajo es mayor (más de 1000 imágenes espaciales a evaluar en el método de Ewald) la rutina que mejor se comporta es la desarrollada en CUDA.

imágenes	puntos	Fortran (ifort)	C++ (icpc)	OMP pragma reduction	CUDA	MPI (mpich2)
100	10x10	0,148	0,156	0,054	0,155	0,392
	25x25	0,689	0,727	0,228	0,453	0,745
	50x50	2,584	2,823	0,754	1,476	1,882
	100x100	9,986	10,972	2,847	5,463	6,386
1000	10x10	0,859	0,726	0,235	0,167	0,554
	25x25	5,018	4,202	1,128	0,474	1,637
	50x50	19,659	16,688	4,267	1,492	5,434
	100x100	78,71	67,411	16,944	5,636	20,789
10000	10x10	7,432	6,081	1,406	0,223	1,773
	25x25	46,318	37,605	8,517	0,78	9,431
	50x50	185,205	151,638	33,816	2,737	35,167
	100x100	760,954	607,812	135,392	10,59	144,722
100000	10x10	74,676	59,391	13,406	0,94	13,882
	25x25	459,856	370,951	83,526	5,361	84,873
	50x50	1822,058	1495,109	335,502	21,022	344,977

Tabla 4.7 - Comparativa de los tiempos de ejecución de las distintas rutinas desarrolladas a lo largo del capítulo.

5. Conclusiones y trabajos futuros.

5.1. Conclusiones.

A lo largo de este proyecto hemos analizado y desarrollado diferentes rutinas paralelas para el cálculo de funciones de Green, tanto para las funciones unidimensionales como las bidimensionales. Para ello hemos empleado distintas tecnologías que nos facilitan la programación sobre los diferentes entornos de computación que nos encontramos actualmente, como son los sistemas de memoria compartida, los de memoria distribuida y los dispositivos de aceleración gráfica.

Para llevar a cabo el desarrollo de estas rutinas paralelas hemos realizado una adaptación y optimización previa de los códigos secuenciales en lenguaje Fortran al lenguaje C++, no solo por la comodidad para trabajar en el mismo sino por ser un lenguaje que destaca por la eficiencia del código generado. Recordemos que el objetivo último de aplicar paralelismo a la resolución de un problema es la disminución en el tiempo de ejecución del mismo, por lo que es importante partir de una versión secuencial lo más rápida posible.

En este aspecto los resultados obtenidos han sido interesantes, consiguiendo versiones secuenciales hasta 2,4 y 1,2 veces más rápidas para las funciones de Green undimensionales y bidimensionales, respectivamente. Otro aspecto a destacar es el gran impacto que el compilador tiene en el código generado, ya que si bien es cierto que hemos obtenido dicha mejora para las funciones de Green bidimensionales haciendo uso del compilador de C++ de Intel, también sorprende el empeoramiento de casi 4,5 veces que se produce al emplear el compilador proporcionado por GNU.

En el caso de las funciones de Green unidimensionales hemos identificado dos niveles diferentes a los que aplicar el paralelismo, tanto aplicado al cálculo de una única función calculando el sumatorio de las series del método de Ewald en paralelo (paralelismo de grano fino) como aplicado al cálculo de distintas funciones al mismo tiempo (paralelismo de grano grueso). Además, debido a la relativamente baja carga computacional de las mismas solo hemos considerado su paralelización mediante OpenMP y CUDA, dado que el coste de las comunicaciones asociadas al uso de MPI reduciría notablemente la mejora obtenida por el mismo.

Los resultados obtenidos con la versión de OpenMP con 4 threads de ejecución suponían mejoras de 3,4 a 3,8 veces en el tiempo de ejecución para el paralelismo de grano fino frente a mejoras de 3,6 a 4 veces para el de grano grueso, tal y como cabía esperar. En cuanto a la versión desarrollada en CUDA apenas hemos llegado a una mejora de 2 veces en el mismo, con empeoramientos en los casos de entrada de tamaño más pequeño, principalmente debido a la baja carga computacional y al overhead introducido por la inicialización del dispositivo tras la primera invocación a un kernel CUDA.

También se ha desarrollado una versión híbrida que combina las dos tecnologías anteriores para aprovechar las capacidades de cómputo de la CPU y la GPU conjuntamente. En este caso seguimos teniendo el empeoramiento que nos producía el uso de la GPU para tamaños de problema muy reducidos, pero conseguimos mejoras superiores, de hasta 4 a 6 veces, para el resto de casos de entrada.

Por otra parte también hemos analizado las funciones de Green bidimensionales, donde al igual que en las anteriores nos encontramos con dos diferentes niveles a los que aplicar paralelismo. De ellos nos hemos centrado en estudiar el nivel más interno o de grano fino, que puede usarse como base para el estudio del nivel más exterior o de grano grueso. Al tratarse de funciones con un orden de complejidad computacional de un orden de magnitud mayor al de las anteriores, hemos desarrollado rutinas tanto en OpenMP y CUDA como haciendo uso del paradigma de paso de mensajes basado en MPI.

En el caso de la paralelización mediante OpenMP hemos planteado diferentes alternativas para solucionar el conflicto de la modificación de zonas de memoria compartida por diferentes threads al mismo tiempo. Los resultados obtenidos para una ejecución con 4 threads nos muestran como una implementación directa a partir del código secuencial y mediante el uso de secciones críticas para el manejo de variables compartidas no permite explotar correctamente el paralelismo, a diferencia de una versión modificada para minimizar los tiempos de sincronización entre threads. En concreto, el speed-up conseguido por la primera de dichas versiones difícilmente alcanza el 2,8, mientras que en la versión modificada hemos conseguido speed-up superlineales de 4 a casi 4,5.

Por otra parte hemos analizado el comportamiento de una versión desarrollada en MPI siguiendo la misma idea que la anterior basada en OpenMP. Los resultados, como era de esperar al haber sido ejecutados sobre el mismo sistema, han sido muy similares a los de estos anteriores, obteniendo speed-up ligeramente inferiores a los mismos para el mismo número de threads de ejecución.

Por último se ha presentado una versión en CUDA que aprovecha el gran paralelismo de datos existente en el problema para acelerar el mismo mediante la GPU. Es aquí donde los resultados han sido más sorprendentes, con mejoras de hasta 70 veces en el tiempo de ejecución y con una tendencia alcista en dichas mejoras, lo que supone que podría ser aún mayor para casos de entrada de tamaños superiores.

Así, hemos evaluado los resultados de aplicar diferentes paradigmas de programación paralela a un problema, valorando diferentes alternativas que se nos presentan a la hora de particionar el problema para su ejecución en distintas unidades de proceso, así como para la sincronización y comunicación de datos entre las mismas. También se ha ofrecido una primera visión del uso de computación híbrida en CPU y GPU para aprovechar el uso de más de uno de dichos paradigmas al mismo tiempo. Además se ha evaluado la escalabilidad de las rutinas paralelas propuestas en función de diferentes parámetros de entrada al problema, consiguiendo resultados más que aceptables para los sistemas evaluados.

5.2. Trabajos futuros.

Hemos presentado a lo largo de este proyecto diferentes técnicas de paralelización para un subconjunto de funciones de Green, habiendo obtenido buenos resultados en la mayoría de las rutinas implementadas. Esto permite tener un punto de partida para la continuación del mismo, que por un lado permita estudiar la aplicación de las mismas u otras técnicas a otras funciones de Green diferentes y por otro lado nos proporcione la capacidad de evaluar el rendimiento de dichas rutinas en problemas de aplicación real de diferente ámbito.

Así, podríamos proponer algunas líneas de trabajo futuro que podrían seguirse como una continuación natural de este proyecto, entre las que se encuentran:

- Finalizar el estudio de las funciones de Green bidimensionales comenzado en el capítulo 4, explotando el paralelismo de grano grueso presente en el mismo y contrastando los resultados obtenidos con los del paralelismo de grano fino expuestos en dicho capítulo, así como el estudio de las funciones de Green tridimensionales.
- Evaluar el comportamiento de las rutinas paralelas desarrolladas en sistemas de computación de diferentes características, desde sistemas de memoria compartida con diferente número de procesadores o jerarquía de memoria, hasta sistemas de memoria distribuida con distintas redes de comunicación.
- Diseñar y evaluar rutinas que aprovechen las capacidades de computación híbridas presentes en la mayoría de sistemas actuales, donde habitualmente tenemos uno o varios procesadores, cada uno con uno o más cores, y una o más tarjetas gráficas que podemos aprovechar para computación general mediante paradigmas como CUDA de NVIDIA o Stream de ATI. Además, es posible que tengamos más de uno de dichos nodos conectados entre sí por una red de comunicación, dando lugar a la posibilidad de explotar los 3 paradigmas utilizados de forma conjunta.
- Continuar explorando las posibilidades que ofrecen los dispositivos de aceleración gráfica para aprovechar el gran paralelismo de datos de este tipo de funciones. Además, también sería interesante comprobar las mejoras que se presentan tanto en el modelo de programación como en el rendimiento ofrecido por los nuevos dispositivos NVIDIA con CUDA Compute Capability 2.0.
- Desarrollar toda una gama de rutinas para el cálculo de funciones de Green optimizadas para diversas arquitecturas y sistemas, que implementen mecanismos de autooptimización transparentes al usuario de forma que, dependiendo de los parámetros de entrada del problema y del sistema donde se va a resolver, se utilice la rutina apropiada que lo resuelva más rápidamente.

6. Referencias.

[1] J. Heinbockel. *"Mathematical Methods for Partial Differential Equations"*. December 2006.

[2] D. G. Duffy. *"Green's Functions with Applications"*. Studies in Advanced Mathematics, Chapman & Hall/CRC, 1st edition, May 2001.

[3] F. J. Pérez, F. D. Quesada, D. Cañete, A. Álvarez, J. R. Mosig. *"A Novel Efficient Technique for the Calculation of the Green's Functions in Rectangular Waveguides Based on Accelerated Series Decomposition"*. IEEE transactions on antennas and propagation, vol. 56, nº 10, October 2008.

[4] F. Capolino, D. R. Wilton, and W. A. Johnson. *"Efficient computation of the 2-D Green's function for 1-D periodic layered structures using the Ewald method"*. IEEE AP-S Symp., San Antonio, TX, Jun. 16–21, 2002.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. "*LAPACK Users' Guide*". Philadelphia, PA: Society for Industrial and Applied Mathematics. 1999.

[6] A. Chtchelkanova, J. Gunnels, G. Morrow, J. Overfelt, and R. van de Geijn. *"Parallel Implementation of BLAS: General Techniques for Level 3 BLAS"*. Concurrency: Practice and Experience, 9(9):837–857, Sept. 1997.

[7] C. Addison, Y. Ren, and M. van Waveren. "*OpenMP issues arising in the development of parallel BLAS and LAPACK libraries*". Scientific Programming, 11(2), 2003.

[8] T. Wittwer. *"Choosing the optimal BLAS and LAPACK library"*.

[9] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. *"GPU computing"*. Proceedings of the IEEE, 96(5), May 2008

[10] T. Hahn, V. Heuveline and B. Rocker. *"GPU accelerated scientific computing: Fluid and particulate flows with CUDA"*.

[11] Y. Okitsu, F. Ino, and K. Hagihara. *"Accelerating Cone Beam Reconstruction Using the CUDA-enabled GPU"*.

[12] Y. Zhang, T. K. Sarkar, H. Moon, A. De, and M. C. Taylor, *"Solution of large complex problems in computational electromagnetic using higher order basis in MoM with parallel solvers"*. Proc. IEEE Antenna and Propagation Symp., Jun. 9–15, 2007, pp. 5620–5623.

[13] Y. Pan, J. Shang and M. Guo. *"A scalable HPF implementation of a finite volume CEM application on a CRAY T3E parallel system"*. Concurrency Comput.: Pract. Exp. 15 (6) (2003) 607–621.

[14] F. D. Quesada, F. J. Pérez, B. Gimeno, V. Boria, J. Pascual, J. Gómez, D. Cañete, A. Álvarez. *"Análisis Eficiente de Componentes Pasivos Inductivos en Guía Empleando una Nueva Formulación de Ecuación Integral en el Dominio Espacial".* XXI Simposium Nacional de la Unión Científica Internacional de Radio.

[15] M.-J. Park and S. Nam. *"Rapid summation of the Green's function for the rectangular waveguide"*. IEEE Trans. Microw. Theory Tech., vol. 46, pp. 2164–2166, Dec. 1998.

[16] M.-J. Park, J. Park, and S. Nam. *"Efficient calculation of the Green's function for the rectangular cavity"*. IEEE Trans. Microw. Guided Wave Lett., vol. 8, no. 3, pp. 124–126, Mar. 1998.

[17] U. Jakobus, I. Sulzer, and F. M. Landstorfer. *"Parallel implementation of the hybrid MoM/Green's function technique on a cluster of workstations"*. ICAP'97, IEE 10th International Conference on Antennas and Propagation, Edinburgh, Conf. Publication Number 436, vol. 1, pp. 182–185, Apr. 1997.

[18] U. Jakobus. *"Parallel computation of the electromagnetic field of hand–held mobile telephones radiating close to the human head"*. Parallel Computing '97 (ParCo97), Bonn, Sept. 1997.

[19] G. Valerio, P. Baccarelli, P. Burghignoli, and A. Galli. *"Comparative analysis of acceleration techniques for 2-D and 3-D Green's functions in periodic structures along one and two directions"*. IEEE Trans. Microw. Theory Tech., vol. 55, pp. 1630–1643, Jun. 2007.

[20] A. Kustepely and A. Q. Martin. *"On the splitting parameter in the Ewald method"*. IEEE Trans. Microw. Guided Wave Lett., vol. 10, no. 5, pp. 168–170, May 2000.