UNIVERSIDAD DE MURCIA
FACULTAD DE INFORMÁTICA
INGENIERÍA EN INFORMÁTICA

Desarrollo, evaluación y autooptimización de rutinas paralelas híbridas de cálculo de funciones de Green

PROYECTO FIN DE CARRERA

Autor:

TOMÁS RAMÍREZ GARCÍA
tomas.ramirez@um.es

Directores:

DOMINGO GIMÉNEZ CÁNOVAS
Grupo de Computación Científica y Programación Paralela
Departamento de Informática y Sistemas
Universidad de Murcia

FERNANDO DANIEL QUESADA PEREIRA
ALEJANDRO ÁLVAREZ MELCÓN
Grupo de Electromagnetismo Aplicado a las Telecomunicaciones
Departamento de Tecnologías de la Información y las Comunicaciones
Universidad Politécnica de Cartagena

Murcia, Septiembre de 2012

Agradecimientos

Este proyecto es el punto culminante después de una gran variedad de exámenes, prácticas y trabajos repartidos a lo largo de todos estos duros años en los que he aprendido mucho. Pero esta época no hubiera sido soportable sin el apoyo de muchas personas y por ello les dedico este trabajo. A mis padres, por estar ahí siempre recordándome que no debo desistir; a Pris, por hacerme compañía en todo momento y darme ánimo infinito; y a mis amigos, por ayudarme a desconectar de los largos días de estudio.

Con respecto al proyecto, quisiera agradecer la ayuda que me han proporcionado Jesús y Luis Pedro siempre que he tenido algún problema con las máquinas y, cómo no, a los directores de este proyecto: Fernando, Alejandro y, especialmente, Domingo, por haberme introducido en esto de la programación paralela y servir de guía principal para que este trabajo haya llegado a su fin.

Tomás Ramírez García

Resumen

Hasta hace unos años, la forma más común para acometer la resolución de un problema por ordenador era secuencialmente. Pero con la evolución de los sistemas de computación hacia arquitecturas basadas en multiprocesadores, surge la necesidad de un nuevo modelo que nos permita coordinar dichos elementos de cálculo, con el objetivo de aprovechar los recursos disponibles y obtener una aceleración en el tiempo de resolución del problema. Aparece entonces la programación paralela.

Con ella, aparecen una serie de paradigmas a utilizar, dependiendo del tipo de arquitectura subyacente. De entre los más notables podemos destacar OpenMP, MPI y CUDA; los cuales están enfocados a, respectivamente, sistemas de memoria compartida, sistemas de memoria distribuida y unidades de procesamiento gráfico (GPUs). Aunque estas últimas están diseñadas específicamente para el procesamiento de gráficos, vienen siendo utilizadas en los últimos años para la resolución de problemas de propósito general, debido a su alta capacidad de cómputo en paralelo.

Este proyecto surge como continuación natural del trabajo realizado en un proyecto anterior, en el que se estudiaron un conjunto de las llamadas funciones de Green, unas herramientas matemáticas que nos permiten resolver ecuaciones diferenciales no homógeneas bajo ciertas condiciones de contorno. En concreto, trabajaremos con una serie de rutinas secuenciales y paralelas que calculan funciones de Green empleadas en el campo del electromagnetismo, como por ejemplo, para el análisis y diseño de filtros.

Tras una fase previa en la que se realiza un análisis de dichas rutinas, con el objetivo de comprender mejor su funcionamiento y, además, modelar el tiempo teórico de ejecución de las mismas, se evalúa el comportamiento de éstas en otros sistemas distintos a donde fueron desarrolladas, comprobando que la mejora del rendimiento al paralelizar es similar a la obtenida en los sistemas utilizados en el anterior proyecto.

A continuación, se continúa investigando sobre la utilización simultánea de los modelos de programación MPI, OpenMP y CUDA, permitiendo sacar el máximo partido de los sistemas computacionales actuales, los cuales suelen estar formados por varios elementos de procesamiento diferentes entre sí organizados jerárquicamente.

Finalmente, como consecuencia de la gran variedad de rutinas disponibles, así como de la posibilidad de ejecutar cada una de ellas con más o menos recursos computacionales y en sistemas de diferentes características, se concluye con la aplicación de un sencillo mecanismo de autooptimización que, en función de los parámetros del problema a resolver y de las características de la máquina donde se esté trabajando, decida en tiempo de ejecución la rutina más adecuada y las unidades computacionales implicadas para su resolución.

Abstract

Few years ago, the most common approach to solve a problem by computer was a sequential one. However, the evolution of computational systems towards multiprocessor-based architectures requires a new model capable to make these computing elements cooperate, in order to make the best of the available resources and reduce a problem resolution time. This brings up parallel programming.

Thereby, several paradigms come on the scene, each used within a particular underlying architecture. OpenMP, MPI and CUDA are highlighted among the most remarkable; which are designed respectively for shared memory systems, distributed memory systems and graphics processing units (GPUs). Although the latter ones are specially designed for graphics processing, for the last years they have been used to solve general purpose problems, because of its high capacity for parallel computing.

This project arises as a natural continuation of the work performed in a previous one, in which a set of the so called Green functions were studied. These are math tools that enable us to solve inhomogeneous differential equations by establishing some boundary conditions. Particularly, we will work on several sequential and parallel routines which compute Green functions used in the field of electromagnetism, for instance, for analyzing and designing filters.

After a preliminary stage in which these routines are analyzed, with the aim of better understanding their performance and also modelling their theoretical execution time, these are assessed in other different systems than where they were developed, verifying that performance improvement when parallelizing is similar to what was achieved in the system used in the previous project.

Next, research continues on simultaneous utilization of the programming models MPI, OpenMP and CUDA, allowing us to take advantage of current computational systems, which usually consist of multiple mutually different processing elements in a hierarchy.

Finally, as a consequence of the wide variety of available routines as well as the possibility of executing each one using more or less computing resources in systems with different specs, we conclude by applying a simple self-optimization mechanism that decide at runtime the most suitable routine and how many computational units will be involved for resolving a problem depending on its parameters and the features of the machine where we work.

Índice general

1. Introducción **1**
1.1. Estado del arte 1
1.2. Objetivos 3
1.3. Metodología 3

2. Herramientas **5**
2.1. Herramientas software 5
2.1.1. *OpenMP*: un estándar para programación en memoria compartida 5
2.1.2. *MPI*: un estándar de paso de mensajes 6
2.1.3. *CUDA*: computación de propósito general sobre GPUs 7
2.2. Herramientas hardware 13
2.2.1. Estación de trabajo *geatpc2* 13
2.2.2. Cluster *hipatia* 14
2.2.3. Servidores *marte* y *mercurio* 15

3. Rutinas de cálculo de funciones de Green **17**
3.1. Funciones de Green unidimensionales 17
3.1.1. Algoritmo secuencial 17
3.1.2. Paralelismo de grano fino con OpenMP 18
3.1.3. Paralelismo de grano fino con CUDA 19
3.1.4. Paralelismo de grano grueso con OpenMP 20
3.1.5. Paralelismo híbrido combinando OpenMP y CUDA 20
3.2. Funciones de Green bidimensionales 22
3.2.1. Algoritmo secuencial 22
3.2.2. Paralelismo de grano fino con OpenMP 23
3.2.3. Paralelismo de grano fino con CUDA 24
3.2.4. Paralelismo de grano fino con MPI 25

4. Evaluación de las rutinas previas **27**
4.1. Funciones de Green unidimensionales 27
4.1.1. Comparación de rendimiento entre `double` y `float` 29
4.1.2. Evolución del speedup 30
4.1.3. Resumen de resultados y contraste con resultados previos 31
4.2. Funciones de Green bidimensionales 32
4.2.1. Evolución del speedup 36
4.2.2. Resumen de resultados y contraste con resultados previos 37

5. Paralelismo híbrido **41**

5.1. Diseño e implementación 42
5.2. Evaluación 44
5.2.1. Comparación de rendimiento entre `g++` e `icpc` 44
5.2.2. Combinando MPI y OpenMP 45
5.2.3. Combinando MPI, OpenMP y CUDA 47

6. Autooptimización **53**
6.1. Metodología 54
6.2. Instalación de la rutina 54
6.3. Elección de la implementación óptima 55
6.4. Contraste de resultados 56

7. Conclusiones y vías futuras **59**
7.1. Conclusiones 59
7.2. Vías futuras 61

A. Instalación del compilador de Intel **63**

B. Instalación del entorno CUDA **65**
B.1. Developer Driver 65
B.2. CUDA Toolkit 65
B.3. GPU Computing SDK 66
B.4. Post-instalación 66
B.4.1. Variables de entorno 66
B.4.2. Librerías del SDK 66

C. Remodelado de las funciones de Green bidimensionales **67**
C.1. Paralelismo de grano fino con OpenMP 67
C.2. Paralelismo de grano fino con MPI 68
C.3. Paralelismo de grano fino con CUDA 68
C.4. Paralelismo híbrido combinando MPI, OpenMP y CUDA 69

Bibliografía **71**

Índice de figuras

1.1. Rendimiento en GFLOPS de una GPU frente a una CPU 2

2.1. Arquitectura unificada NVIDIA *Tesla* de la GPU GeForce 8800 8
2.2. Streaming multiprocessor (SM) de la GPU GeForce 8800 8
2.3. Planificación de warps SIMT en la arquitectura unificada NVIDIA *Tesla* . . 9
2.4. Ejecución heterogénea y jerarquía de hilos de CUDA 11
2.5. Jerarquía de memoria de CUDA 12
2.6. Accesos a memoria global con CUDA Compute Capability < 1.2 12
2.7. Topología de *geatpc2* obtenida con `lstopo` 14
2.8. Topología de *hipatia* (nodos *cn3* y *cn4*) obtenida con `lstopo` 15
2.9. Topología de *marte* obtenida con `lstopo` 16

4.1. Slowdown producido en las funciones de Green unidimensionales al adaptar el código Fortran a C++ (*geatpc2*) 28
4.2. Comparativa de los speedups de las diferentes implementaciones paralelas de las funciones de Green unidimensionales (*geatpc2*) 28
4.3. Speedup alcanzado en las implementaciones secuencial y OpenMP (8 threads) de las funciones de Green unidimensionales al utilizar `float` en lugar de `double` (*geatpc2*) 30
4.4. Evolución del speedup de las implementaciones OpenMP y OpenMP+CUDA de las funciones de Green unidimensionales (*geatpc2*) . . 31
4.5. Speedup alcanzado en las funciones de Green bidimensionales al adaptar el código Fortran a C++ (*geatpc2*) 33
4.6. Speedup alcanzado en las funciones de Green bidimensionales al adaptar el código Fortran a C++ (*hipatia*) 33
4.7. Comparativa de los speedups de las implementaciones paralelas OpenMP y CUDA de las funciones de Green bidimensionales (*geatpc2*) 34
4.8. Comparativa de los speedups de las implementaciones paralelas OpenMP y MPI de las funciones de Green bidimensionales (*hipatia*) 35
4.9. Evolución del speedup de las implementaciones OpenMP y MPI de las funciones de Green bidimensionales (*hipatia*) 36

5.1. Ejemplo de un sistema computacional jerárquico básico 41
5.2. Speedup alcanzado en las funciones de Green bidimensionales al utilizar `icpc` en puesto de `g++` (*marte*) 45
5.3. Speedups de la rutina híbrida MPI+OpenMP en un nodo (*hipatia*) 46
5.4. Speedups de la rutina híbrida MPI+OpenMP en un nodo (*marte*) 46
5.5. Speedups de la rutina híbrida MPI+OpenMP en dos nodos (*hipatia*) 47

5.6. Speedups de la rutina híbrida MPI+OpenMP en dos nodos (*marte/mercurio*) 48
5.7. Evolución del speedup de la rutina híbrida MPI+OpenMP+CUDA utilizando un *kernel* por nodo (*marte/mercurio*) 49
5.8. Evolución del speedup de la rutina híbrida MPI+OpenMP+CUDA utilizando tres *kernels* paralelos por nodo (*marte/mercurio*) 50
5.9. Evolución del speedup de la rutina híbrida MPI+OpenMP+CUDA utilizando tres *kernels* paralelos por nodo (*marte/mercurio*) 51

6.1. Representación gráfica de la vecindad entre problemas 56

Índice de tablas

4.1. Preferencia de rutina y número de cores para las funciones de Green unidimensionales en función del tamaño del problema (*geatpc2*) 31
4.2. Speedups obtenidos con las rutinas preferidas para el cálculo de las funciones de Green unidimensionales en función del tamaño del problema (*geatpc2*) 32
4.3. Preferencia de rutina y número de cores para las funciones de Green unidimensionales en función del tamaño del problema (*luna*) 32
4.4. Preferencia de rutina y número de cores para las funciones de Green bidimensionales en función del tamaño del problema (*geatpc2*) 37
4.5. Preferencia de rutina y número de cores para las funciones de Green bidimensionales en función del tamaño del problema (*hipatia*) 37
4.6. Speedups obtenidos con las rutinas preferidas para el cálculo de las funciones de Green bidimensionales en función del tamaño del problema (*geatpc2*) 38
4.7. Speedups obtenidos con las rutinas preferidas para el cálculo de las funciones de Green bidimensionales en función del tamaño del problema (*hipatia*) 38
4.8. Preferencia de rutina y número de cores para las funciones de Green bidimensionales en función del tamaño del problema (*luna*) 38

5.1. Rutinas disponibles con las diferentes configuraciones de la implementación híbrida 44
5.2. Configuraciones preferidas de la rutina híbrida MPI+OpenMP+CUDA (*marte/mercurio*) 51

6.1. Rutinas y configuraciones preferidas para el conjunto de instalación y su speedup asociado (*hipatia*) 54
6.2. Rutinas y configuraciones preferidas para el conjunto de instalación y su speedup asociado (*marte/mercurio*) 55
6.3. Rutinas y configuraciones preferidas para el conjunto de validación aplicando autooptimización (*hipatia*) 56
6.4. Rutinas y configuraciones preferidas para el conjunto de validación aplicando autooptimización (*marte/mercurio*) 56
6.5. Comparación entre el tiempo de ejecución óptimo y el obtenido con autooptimización para el conjunto de validación (*hipatia*) 57
6.6. Comparación entre el tiempo de ejecución óptimo y el obtenido con autooptimización para el conjunto de validación (*marte/mercurio*) 57

Listados de código

3.1. Pseudocódigo secuencial para calcular funciones de Green unidimensionales 18
3.2. Pseudocódigo OpenMP de la función `spectgf_dsum()` 19
3.3. Pseudocódigo CUDA de la función `spectgf_dsum()` 19
3.4. Pseudocódigo OpenMP de la función `GF_Capa2Dt_Tex_conv()` 20
3.5. Pseudocódigo OpenMP+CUDA para calcular funciones de Green unidimensionales 21
3.6. Pseudocódigo secuencial para calcular funciones de Green bidimensionales 22
3.7. Pseudocódigo OpenMP de la función `GF_wg_ewald_xy_ord()` 24
3.8. Pseudocódigo CUDA de la función `GF_wg_ewald_xy_ord()` 24
3.9. Pseudocódigo MPI de la función `GF_wg_ewald_xy_ord()` 26

5.1. Pseudocódigo MPI+OpenMP+CUDA de la función `GF_wg_ewald_xy_ord()` 43

6.1. Pseudocódigo de la función que elige la implementación y configuración . 55

A.1. Script de instalación del compilador de Intel 63

Capítulo 1

Introducción

El presente proyecto tiene como objetivo general continuar el trabajo realizado en un proyecto anterior [21], en el cuál se llevó a cabo la aceleración de una serie de rutinas utilizadas para el cálculo de funciones de Green en problemas de electromagnetismo, bajo la utilización de varios modelos y técnicas de programación paralela.

El trabajo consistirá en seguir investigando con las funciones de Green, desarrollando **nuevas rutinas paralelas híbridas** que las resuelvan, combinando varios paradigmas de programación paralela; evaluando las mismas en **diferentes sistemas actuales, que suelen ser híbridos, heterogéneos y jerárquicos**, y que, por lo tanto, permitan aprovechar las características híbridas de las rutinas; y valorar la posibilidad de desarrollar un **mecanismo de autooptimización (*autotuning*)** que escoja dinámicamente la mejor de las rutinas disponibles y los recursos a utilizar (número de nodos, cores, GPUs. . .) en función de los parámetros de entrada al problema y del sistema computacional donde se esté trabajando.

1.1. Estado del arte

Desde su aparición, los computadores han supuesto una revolución para la investigación científica, ya que estos permitían resolver problemas cuya resolución manual era impensable por la inmensa cantidad de cálculos implicados. Los continuos avances realizados en arquitectura de computadores han ido dotando a estas máquinas de la capacidad de resolver problemas cada vez más complejos. Hasta hace unos pocos años, la tendencia seguida para conseguir dichos avances era incrementar la frecuencia de reloj a la que trabajaba el procesador. Sin embargo, dado que esta tendencia significaba aumentar el consumo de energía (*power wall*), entre otros problemas, se ha dejado de lado para dar paso a un nuevo modelo de computador, el multiprocesador. Con él aparece la programación paralela como un nuevo enfoque de resolución de problemas que aprovecha los elementos de procesamiento replicados para la ejecución simultánea de varias operaciones.

Partiendo de este planteamiento para resolver problemas en paralelo, recientemente se ha identificado la posibilidad de utilizar dispositivos de aceleración gráfica o GPUs como si se trataran de multiprocesadores de propósito general, dando lugar al paradigma de programación GPGPU (*General Purpose computing on Graphics Processing Units*) [16].

El empleo de GPUs para realizar procesamiento de tipo no gráfico se debe a la rápida evolución del rendimiento de este tipo de dispositivos en contraposición a la que han sufrido los procesadores de propósito general a lo largo de los años, como se puede ver en la figura 1.1, además de que se trata de dispositivos diseñados específicamente para realizar un procesamiento de datos masivamente paralelo.

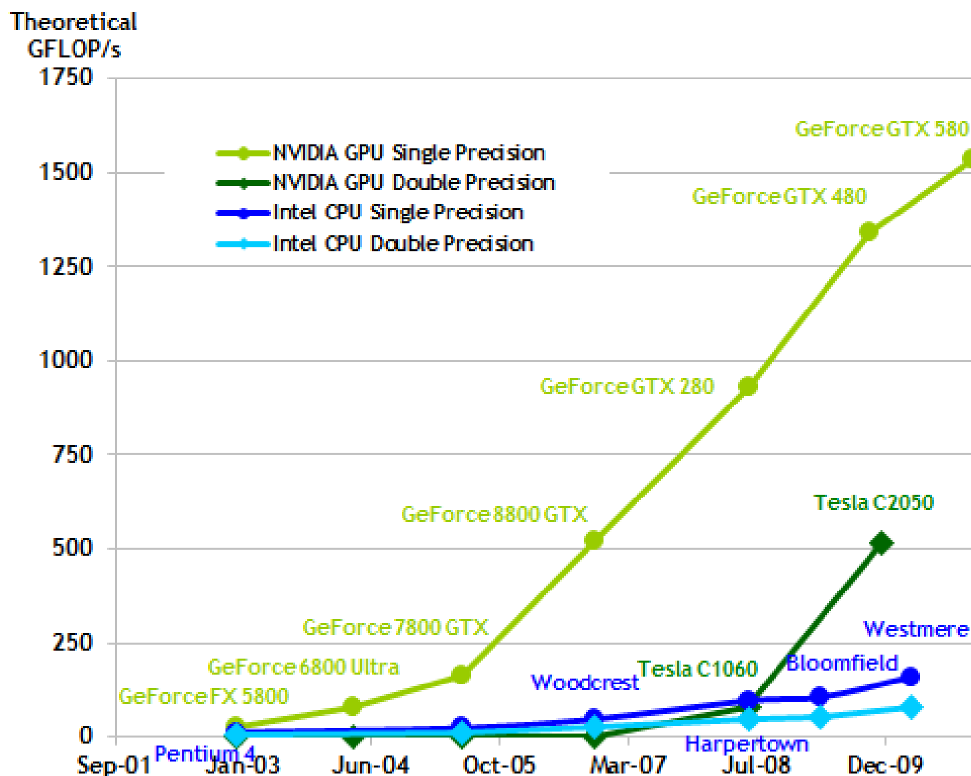

Figura 1.1: Rendimiento en GFLOPS de una GPU frente a una CPU (fuente: [8])

La aparición de este nuevo paradigma ha tenido un enorme efecto en la computación científica, ya que un simple ordenador personal equipado con una tarjeta gráfica se puede convertir en un supercomputador de bolsillo, sin tener que invertir en máquinas multiprocesador de un coste mucho mayor. Es por ello que se vienen desarrollando últimamente trabajos que involucran este tipo de dispositivos para la aceleración de algoritmos, como [15], donde podemos ver ejemplos de simulación de dinámica de fluidos y molecular, multiplicación de matrices densas. . . , y [13], donde se utiliza la transformada de wavelet para el análisis de imágenes.

Además de utilizar un sistema multiprocesador (un multicore o una GPU) por sí solo, también se puede aprovechar la organización jerárquica de los sistemas actuales [1] para resolver problemas aplicando adecuadamente diferentes niveles de paralelismo [25, 26].

También es algo muy común disponer de varias implementaciones paralelas distintas de rutinas que resuelven un mismo problema, de forma que los usuarios de éstas puedan elegir la más adecuada en función de las características de los datos de entrada al problema y del sistema utilizado. Pero normalmente estos usuarios son personas que no tienen un conocimiento profundo ni del sistema ni del comportamiento interno de las rutinas, por lo que difícilmente pueden predecir qué rutina será la más rápida. Debido a esto, se han desarrollado varios trabajos de autooptimización de código paralelo [14, 27], algunos

de ellos para sistemas multicore [9, 12], que pretenden que la implementación óptima (o una que se acerque a ella) sea escogida de forma automática evitando la intervención del usuario.

En lo que respecta al electromagnetismo computacional y la aplicación de las funciones de Green en este ámbito, aunque existen varios trabajos donde se realiza la aceleración por métodos numéricos del cálculo de dichas funciones [10, 19, 20], no disponemos de referencias sobre implementaciones paralelas de las mismas anteriores a [23], la cual servirá como punto de partida para el desarrollo del presente proyecto.

1.2. Objetivos

La principal motivación de este proyecto es realizar un estudio sobre como es posible conseguir la aceleración de una aplicación científica basada en un problema real mediante la aplicación de diferentes técnicas y herramientas de programación paralela. Como ya se mencionó anteriormente, este proyecto parte de uno previo en el que se desarrollaron un conjunto de rutinas paralelas sobre diferentes modelos que conseguían acelerar el cálculo de funciones de Green mediante el método de Ewald, para los casos de una y dos dimensiones, dejando a un lado el caso tridimensional.

Teniendo lo anterior como punto de partida, este proyecto tiene por objetivo continuar con el estudio de las funciones de Green:

- Evaluando el comportamiento de las rutinas ya desarrolladas en otros sistemas diferentes.
- Investigando sobre nuevas oportunidades de paralelismo, como nuevas implementaciones híbridas o utilización de sistemas heterogéneos frente a homogéneos.
- Valorando también la posibilidad de incluir un proceso de *autotuning* a las rutinas, de forma que se pueda utilizar la implementación óptima de entre todas las disponibles de forma automática, en función del tamaño del problema a resolver.

1.3. Metodología

La metodología a seguir a lo largo del proyecto será la siguiente:

- Comprobar el correcto funcionamiento y evaluar la mejora obtenida de las rutinas disponibles en otros sistemas distintos a donde fueron desarrolladas, comparando con los resultados obtenidos anteriormente de modo que obtengamos una valoración de la utilidad de las mismas en términos generales y no sujetos a la utilización de un sistema concreto.
- Identificar nuevas posibilidades de paralelismo combinando dos o más de los modelos de programación paralela que nos ofrecen las herramientas software presentadas en la sección 2.1, los cuales ya han sido utilizados individualmente en las rutinas disponibles, y en una de ellas ya se ha comprobado el comportamiento conjunto de dos de las tecnologías que se presentan.

- Evaluar y comparar las nuevas implementaciones híbridas entre sí y con la versión secuencial, a fin de determinar cual de ellas ofrece mejores resultados en función de los parámetros de entrada al problema. Además, cada versión híbrida deberá compararse consigo misma para diferentes configuraciones de hilos y/o procesos, con objeto de comprobar cuál es la configuración óptima para cierto tamaño del problema.

- Diseñar un mecanismo de autotuning que permita al algoritmo elegir, en tiempo de ejecución, la implementación paralela o secuencial más eficiente para unos valores de los parámetros de entrada concretos.

- Presentar las conclusiones obtenidas tras el trabajo realizado y los resultados alcanzados. Además, se ofrecerán posibles vías futuras de trabajo para continuar con el desarrollo del mismo.

Capítulo 2

Herramientas

A lo largo del desarrollo del proyecto han sido utilizadas una serie de herramientas que podemos clasificar en dos bloques: *software* y *hardware*. Las herramientas software se corresponden con las diferentes APIs y entornos empleados para el desarrollo y la ejecución de las distintas rutinas paralelas. Como herramientas hardware tendremos a un conjunto de máquinas —cada una con sus propias características—, las cuales utilizaremos para comparar el comportamiento de las diferentes versiones paralelas implementadas en función del sistema utilizado.

2.1. Herramientas software

Los avances tecnológicos logrados en el ámbito del hardware, concretamente, la aparición de los sistemas multiprocesadores, ya sean de propósito general o específico, ha llevado a la especificación e implementación de plataformas software para dar soporte a los diferentes paradigmas de programación paralela existentes. De entre toda la gama disponible, en este proyecto se utilizarán OpenMP, MPI y CUDA; sobre sistemas de memoria compartida, memoria distribuida y unidades de procesamiento gráfico, respectivamente.

2.1.1. *OpenMP*: un estándar para programación en memoria compartida

OpenMP (*Open Multi-Processing*) es una API (*Application Program Interface*) multiplataforma para programación paralela multithread sobre arquitecturas de memoria compartida. Está compuesto de una serie de rutinas de librería (*Runtime Library*), directivas de compilación y variables de entorno que permiten ejecutar en paralelo aplicaciones escritas en C/C++ y Fortran.

El tipo de arquitecturas hardware para las que está pensado OpenMP son los *multiprocesadores de memoria compartida*. Estos se componen de una serie de elementos de proceso que comparten el acceso a un espacio de memoria común a todos ellos, pudiendo este espacio estar dividido y distribuido a lo largo de varios módulos, a los que se accedería mediante un BUS o una red de interconexión, pero utilizando un único espacio de direccionamiento lógico, de forma que la memoria sea vista por todos las unidades de procesamiento como un único recurso.

La comunicación entre los diferentes procesadores se consigue utilizando dicho espacio de direcciones común de la siguiente manera: uno de ellos escribe en cierta posición de memoria y otro cualquiera puede leer los datos contenidos en esa misma posición. Este mecanismo implícito de comunicación entre los distintos procesadores implica que puede producirse gran cantidad de movimientos de datos entre los elementos de proceso y el sistema de memoria. Además, el hecho de que varios de ellos puedan intentar acceder a la misma dirección de memoria simultáneamente, tanto para leer como para escribir, convierte al sistema de memoria en un cuello de botella bastante importante, que puede mermar la eficiencia de una aplicación paralela al aumentar el número de unidades de proceso que actúan en paralelo. Esto quiere decir que la aplicación dejaría de ser escalable.

La API fue definida de forma conjunta y colaborativa entre grandes empresas de hardware y software, y entornos gubernamentales y académicos, conformando el ARB (*Architecture Review Board*). Según [4], los actuales miembros del ARB son: como permanentes, AMD, CAPS-Entreprise, Convey Computer, Cray, Fujitsu, HP , IBM, Intel, Microsoft, NEC, NVIDIA, Oracle Corporation, The Portland Group Inc. y Texas Instruments; y como auxiliares, ANL, ASC/LLNL, cOMPunity, EPCC, LANL, NASA, ORNL, RWTH Aachen University y Texas Advanced Computing Center. El objetivo principal era proponer OpenMP como un estándar para programación en memoria compartida. Si bien aún no es un estándar oficial, puede ser considerado claramente el estándar *de facto* para la programación paralela en máquinas de memoria compartida.

2.1.2. *MPI*: un estándar de paso de mensajes

MPI (*Message Passing Interface*) es una especificación de interfaz para una librería de paso de mensajes que define la sintaxis y la semántica de un conjunto de más de 120 funciones destinadas a facilitar la escritura de programas paralelos para arquitecturas de memoria distribuida. Con ello, ofrece un estándar portable, eficiente, con una extensa funcionalidad y una gran variedad de implementaciones de calidad. Además, aunque esté más enfocado al desarrollo sobre sistemas de memoria distribuida, también es posible utilizarlo sobre otro tipo de arquitecturas.

Los multiprocesadores de memoria distribuida, también conocidos como *multicomputadores*, tienen una organización bien distinta a los sistemas comentados en el apartado 2.1.1. Aunque están formados también por varios elementos de proceso, cada uno de ellos tiene su propia memoria local donde almacena sus datos, con su espacio de direcciones privado, eliminando el concepto de memoria global común presente en los multiprocesadores de memoria compartida.

El bloque formado por cada procesador y su espacio de memoria propio conforma un nodo computacional. Para permitir comunicaciones entre los diferentes nodos, este tipo de sistemas requiere una red de interconexión que los una. Esta interacción entre los distintos nodos se realizará mediante las operaciones de paso de mensajes disponibles en la API —en nuestro caso MPI—, controladas de manera explícita por parte del programador.

Para la definición del estándar MPI, fue constituido el *MPI Forum*, formado por un gran número de organizaciones diferentes, principalmente de Estados Unidos y Europa,

entre las que estaban presentes la mayoría de los principales vendedores de computadores paralelos, investigadores de universidades, laboratorios gubernamentales e industria. El objetivo que se perseguía no era un estándar oficial, sino un estándar por consenso, capaz de atraer a gran cantidad de usuarios. Al ser una especificación y no una implementación particular, cada proveedor ofrecerá sus propias implementaciones de MPI optimizadas para sus máquinas, además de las implementaciones gratuitas que podemos descargar de Internet.

2.1.3. *CUDA*: computación de propósito general sobre GPUs

CUDA (*Compute Unified Device Architecture*), introducido por NVIDIA en noviembre de 2006, es una arquitectura hardware y software de computación paralela para propósito general sobre unidades de procesamiento gráfico (GPUs). Se desarrolló inicialmente como una extensión del lenguaje C, pero ya se puede acceder desde C++, OpenCL, DirectCompute, CUDA Fortran y otros [8]. CUDA nos ofrece un nuevo modelo de programación, junto con una serie de librerías y herramientas de desarrollo, para facilitar el desarrollo y la depuración de aplicaciones que empleen un dispositivo con *CUDA Compute Capability* (que soporte CUDA) para resolver problemas de propósito general, en contraposición a las clásicas APIs orientadas al procesamiento gráfico de vértices y pixels como, por ejemplo, OpenGL y Direct3D.

Arquitectura hardware

Para entender mejor el modelo de programación de CUDA se hace necesario hacer una breve introducción a la arquitectura hardware subyacente, ya que la estructura de ésta es más compleja que la de un típico multiprocesador de memoria compartida o distribuida. Como ejemplo, nos apoyaremos en la GPU GeForce 8800. La figura 2.1 muestra un diagrama de bloques de dicha GPU, la cual está basada en una arquitectura *Tesla*, la primera en seguir el modelo unificado de CUDA [18]. Ésta está formada por 128 núcleos de procesamiento denominados *streaming processors* (*SPs*) repartidos en 16 *streaming multiprocessors* (*SMs*).

Cada SM es, básicamente, un multiprocesador encargado de ejecutar partes de programas de procesamiento gráfico o programas paralelos, que es en lo que realmente estamos interesados. Como podemos ver en la figura 2.2, cada SM está compuesto de 8 SP totalmente segmentados, dos *special-function units* (*SFUs*), una unidad de búsqueda/emisión de instrucciones multihilo (*MT issue*), una caché de instrucciones (*I cache*), otra de sólo lectura para datos constantes (*C cache*), y 16 kB de memoria compartida de lectura/escritura.

Lo que resulta más interesante del SM es la unidad de emisión de instrucciones multihilo hacia los SP, la cual permite al SM ejecutar concurrentemente varios hilos. Para poder hacerlo de manera eficiente con cientos de hilos, el hardware del SM está *multithreaded*. Esto significa que puede llegar a ejecutar hasta 768 hilos sin sufrir ningún tipo de sobrecarga en la planificación.

Para poder conseguir lo anterior, el SM tiene una arquitectura de procesador llamada *single-instruction, multiple-thread* (*SIMT*). El SM crea, maneja, planifica y ejecuta los hilos

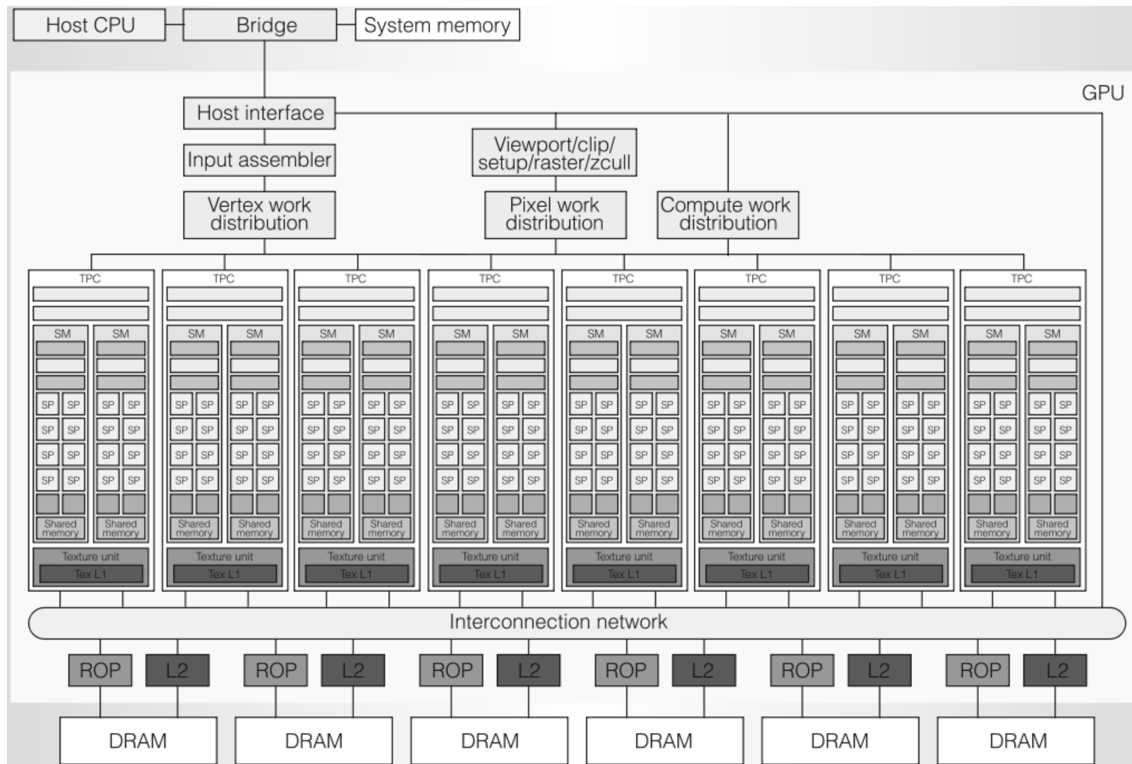

Figura 2.1: Arquitectura unificada NVIDIA *Tesla* de la GPU GeForce 8800 (fuente: [18])

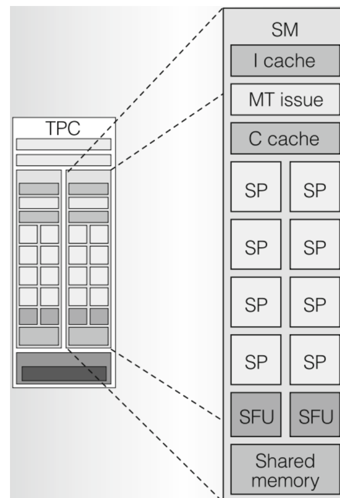

Figura 2.2: Streaming multiprocessor (SM) de la GPU GeForce 8800 (fuente: [18])

en grupos de 32 hilos paralelos, llamados *warps*. En la figura 2.3 se ilustra el funcionamiento de la planificación SIMT. El tamaño del warp a 32 hilos paralelos es perfecto para extraer paralelismo de grano fino.

Cada SM maneja un pool de 24 warps, que hacen un total de 768 hilos. Los hilos de un warp han de ser del mismo tipo y comienzan juntos en la misma instrucción del programa, pero que a su vez pueden ramificarse y ejecutarse de forma independiente al grupo. Para emitir una instrucción, se selecciona un warp que esté preparado para su ejecución y se emite la instrucción SIMT hacia ese warp. Esta emisión no es más que un broadcast

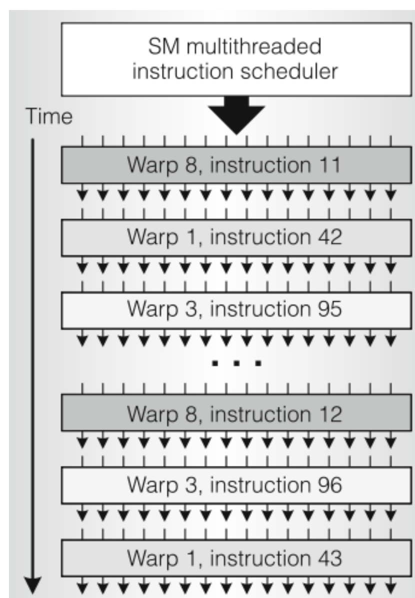

Figura 2.3: Planificación de warps SIMT en la arquitectura unificada NVIDIA *Tesla* (fuente: [18])

sincronizado a todos los threads del warp, indicando que ejecuten la instrucción correspondiente.

El SM entonces asigna los hilos del warp a los SP, de forma que cada hilo se ejecute independientemente con su propio contador de programa y registros. Si los 32 hilos del warp siguen el mismo "camino" de ejecución, es decir, no se producen saltos ni divergencias entre los hilos, el SM consigue la máxima eficiencia. Si por el contrario, algunos hilos sufren saltos independientes. La ejecución de cada camino se serializa hasta que se regrese al punto de ejecución común.

Esta arquitectura SIMT es similar al modelo SIMD de la taxonomía de Flynn, donde una única instrucción opera sobre un conjunto más o menos grande de datos. La principal diferencia es que el diseño SIMT indica una instrucción como punto de partida a varios hilos independientes que se ejecutarán en paralelo, y no necesariamente sobre los datos de un vector. La filosofía SIMT es, por ello, más compleja que un diseño SIMD, pero también más completa, ya que permite que cada hilo tome caminos independientes, aunque por ello se sufra una penalización en el rendimiento, como ya hemos visto.

Debemos destacar que, aunque una GPU posea un número elevado de núcleos —la arquitectura mostrada dispone de 128— cada uno de estos trabaja a una frecuencia inferior en comparación a un procesador de propósito general. La mejora que obtenemos utilizando este tipo de dispositivos es gracias al alto grado de paralelismo del hardware y al bajo coste en la planificación de los hilos.

Por último, si trabajamos con números reales, debemos tener en cuenta también que el rendimiento será muy superior si utilizamos punto flotante de simple precisión, pues, además de que el soporte para reales de doble precisión fue introducido para las GPUs con CUDA Compute Capability ≥ 1.3, este último se realiza mediante la utilización de unidades funcionales compartidas por todos los SP del mismo SM.

Modelo de programación

Vista la arquitectura interna de una GPU con soporte CUDA, pasamos a ver como se pueden aprovechar las capacidades de estos dispositivos desde el punto de vista del programador. La GPU es simplemente un coprocesador de la CPU, en el cuál se ejecutarán las partes del código que se identifiquen como intensivas en paralelismo de datos y se programen para tal efecto, quedando otras partes para su ejecución en la CPU. Por tanto, al utilizar CUDA estamos tratando con un entorno de computación heterogéneo, formado por el *host* y el *device*, es decir, la CPU y la GPU según la terminología del modelo CUDA.

Las partes del código que se ejecutan en el device se les conoce como *kernels*. Los kernels no son más que funciones que, cuando son llamadas, se ejecutan en paralelo por varios *CUDA threads* diferentes, en contraposición a una función C secuencial. Estos tienen un identificador único según su posición en la jerarquía de hilos de CUDA. Esta organización se define como una malla (*grid*) de bloques de hilos (*thread blocks*). En la figura 2.4 se refleja el modelo de ejecución con la jerarquía de hilos, donde la CPU está ejecutando código secuencial hasta que realiza la llamada a un kernel, el cual se ejecuta en la GPU. Entonces la CPU, o bien espera a que el kernel termine o bien continúa ejecutando código secuencial. Se destaca el hecho de que el segundo kernel deberá esperar a que el primero termine —si no lo ha hecho— para poder ejecutarse.

Además de todo lo anterior, debemos considerar que, al tener espacios separados de memoria la CPU y la GPU, los movimientos de datos entre ambas memorias se harán explícitamente desde el código por parte del programador utilizando funciones de la API que proporciona CUDA.

Modelo de memoria

Los hilos de CUDA tienen acceso a diferentes espacios de memoria durante su ejecución, como se puede observar en la figura 2.5. Cada uno tendrá un tiempo de acceso y un fin diferente a los del resto. Además de los que se muestran en la figura, existen otros que explicamos a continuación:

Banco de registros (*register file*): Están repartidos entre todos los threads blocks en ejecución, y el tiempo requerido para acceder es muy pequeño.

Memoria compartida (*shared memory*): Al igual que el banco de registros, es un espacio repartido entre todos los threads blocks en ejecución. Como su propio nombre indica, es compartida, pero solamente por todos los hilos de un bloque concreto con un tiempo de acceso similar a los registros. Actúa como una especie de caché, almacenando datos de forma temporal para los hilos de un bloque.

Memoria global (*global memory*): Se trata de memoria compartida por todos los thread blocks y con un tiempo de acceso elevado (cientos de ciclos). Será esta región de memoria la que se utilizará para escribir y leer los datos por parte de la CPU para comunicarse con la GPU, por lo que es fundamental una buena utilización de la misma para conseguir un mejor rendimiento de las aplicaciones.

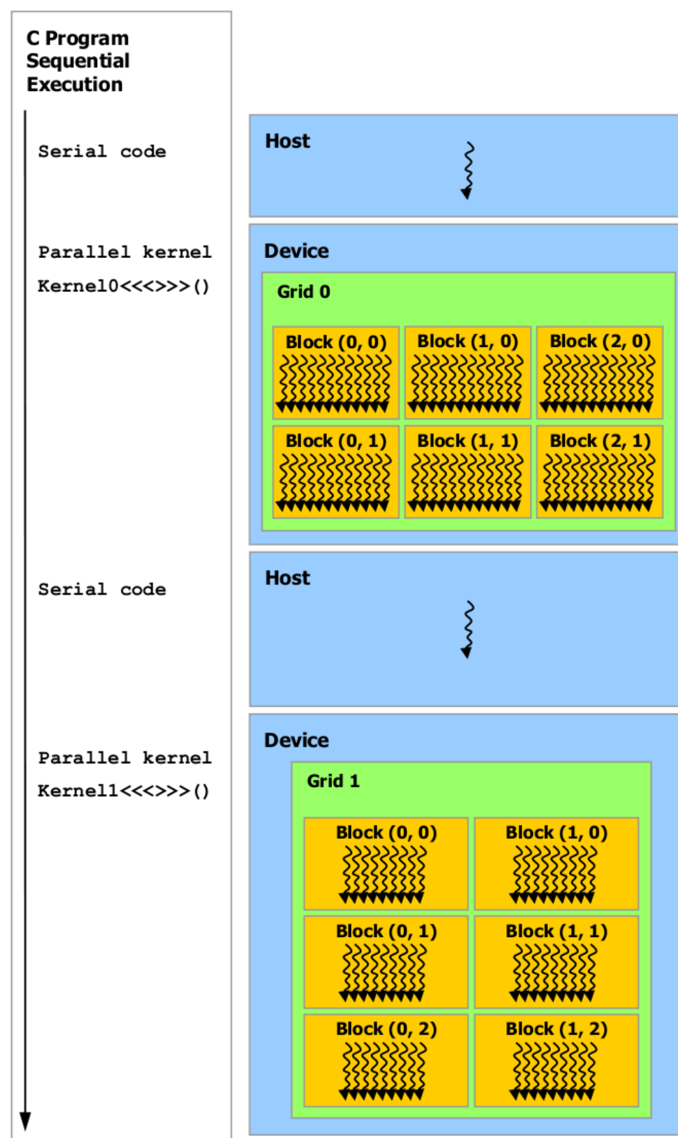

Figura 2.4: Ejecución heterogénea y jerarquía de hilos de CUDA (fuente: [8])

Memoria local (*local memory*): Se trata de memoria privada para cada hilo, para la pila y las variables locales; con propiedades similares a la memoria global.

Memoria constante (*constant memory*): Memoria de sólo lectura con un tiempo de acceso similar a los registros, en la que todos los hilos de un warp pueden leer el mismo valor de forma simultánea en un ciclo de reloj.

Memoria de texturas (*texture memory*): También de sólo lectura, con un tiempo de acceso elevado pero menor que para memoria global. Se utiliza especialmente para explotar localidad espacial al trabajar con imágenes.

Se destaca que las memorias global, constante y de texturas son persistentes entre la ejecución de varios kernels de la misma aplicación.

Patrón de acceso. En CUDA, es crucial la manera en que los hilos de un warp acceden a memoria global. En particular, lo ideal sería que los 16 hilos de un *half-warp* accedieran

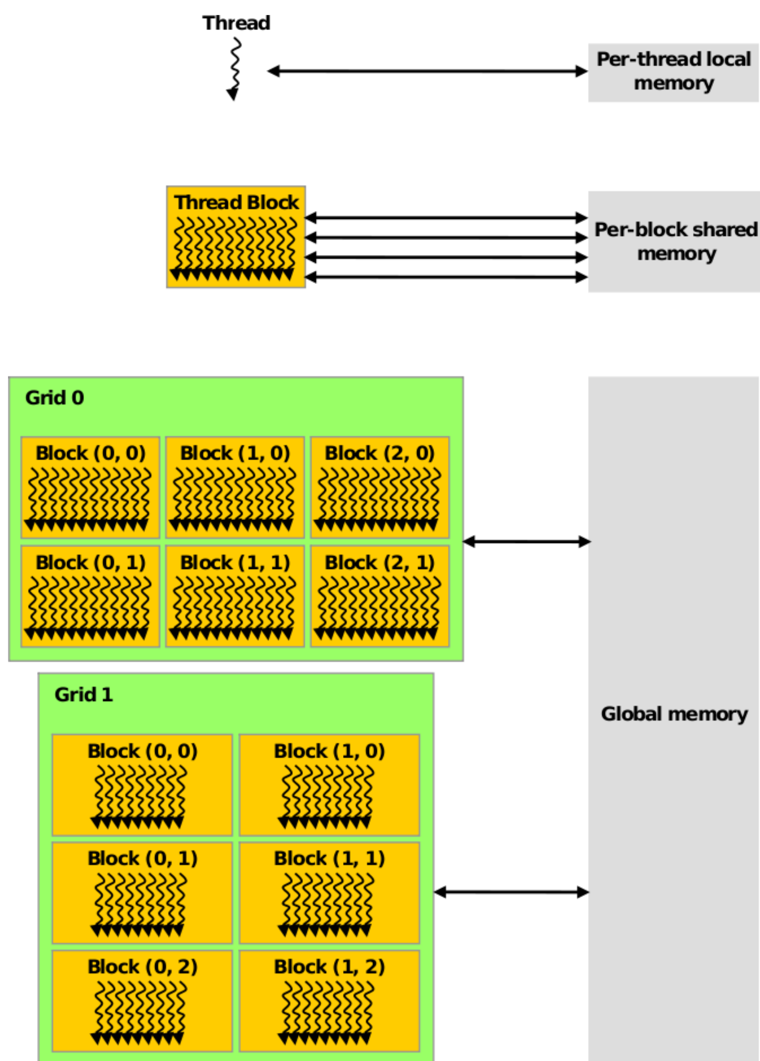

Figura 2.5: Jerarquía de memoria de CUDA (fuente: [8])

a 16 bloques de memoria contiguos, de forma que se pudiera realizar la lectura/escritura simultáneamente. Esto quiere decir que, si el k-ésimo thread de un half warp (16 threads) accede al k-ésimo elemento de un mismo segmento, los 16 accesos a memoria global se combinan en uno solo. Esto es lo que se considera un acceso *coalesced* a memoria global frente a un acceso *non-coalesced* (ver figura 2.6).

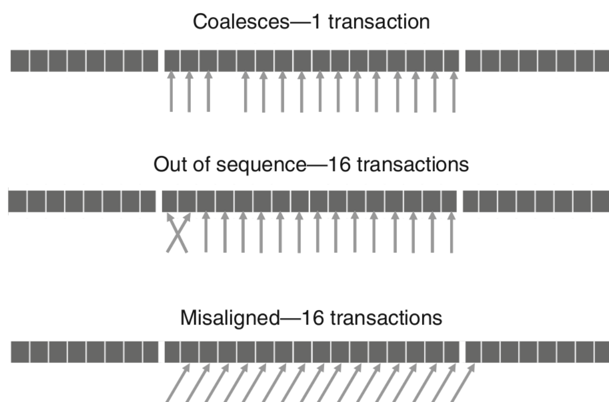

Figura 2.6: Accesos a memoria global con CUDA Compute Capability < 1.2 (fuente: [17])

Como se puede observar en la figura 2.6, no todos los hilos tienen la obligación de participar en el acceso, pero sí debe ser el k-ésimo thread del warp el que acceda al k-ésimo elemento del segmento. Sin embargo, a partir de la CUDA Compute Capability 1.2 las restricciones para conseguir un acceso coalesced son bastante más flexibles, eliminando este requisito de correspondencia hilo/elemento en los accesos.

Este tipo de patrón de acceso también se aplicaría de manera muy similar a la utilización de la memoria compartida. Además, ignorarlo supondría una penalización en el ancho de banda de acceso a la memoria global de aproximadamente un orden de magnitud, pues se estarían realizando 16 transacciones de memoria frente a una.

2.2. Herramientas hardware

Para evaluar el rendimiento de las diferentes rutinas desarrolladas se dispondrá de un conjunto de sistemas multiprocesadores. Algunos de éstos son parte del equipamiento de los laboratorios de los grupos de investigación participantes en este proyecto, mientras que otros son potentes supercomputadores facilitados por servicios externos de cálculo y apoyo a la investigación.

Los resultados de la evaluación realizada en estos sistemas será comparada con los obtenidos en [21], donde se utilizó el servidor *luna* del *Grupo de Computación Científica y Programación Paralela* de la *Universidad de Murcia*, una máquina de memoria compartida con un procesador de 4 núcleos a 2,4 GHz y una GPU con un total de 112 cores a 1,5 GHz.

2.2.1. Estación de trabajo *geatpc2*

El *Grupo de Electromagnetismo Aplicado a las Telecomunicaciones* de la *Universidad Politécnica de Cartagena* dispone de varias estaciones de trabajo, localizadas en sus laboratorios, para realizar cálculos y simulaciones por computador. A lo largo del proyecto, evaluaremos las rutinas desarrolladas en uno de estos equipos, *geatpc2*, un *HP xw8600 Workstation* con los siguientes recursos computacionales:

- Sistema Operativo Linux OpenSUSE 11.0 para arquitecturas de 64 bits.
- 2 procesadores Quad-Core Intel Xeon CPU E5440, con cuatro núcleos cada uno —lo que hace un total de 8 cores en memoria compartida— a una frecuencia de reloj de 2,83 GHz y con arquitectura de 64 bits. Caches L1 de 32 KB privadas a cada core y L2 de 6 MB compartidas por cada dos núcleos.
- Un módulo de memoria RAM de 63 GB.
- Un dispositivo de aceleración gráfica NVIDIA QuadroFX 4600, con CUDA Compute Capability 1.0. Este dispone de 12 multiprocesadores de 8 cores, haciendo un total de 96 CUDA cores a 1,2 GHz. La cantidad total de memoria disponible para la GPU es de 768 MB.

Gracias al comando `lstopo` del paquete `hwloc` [5] podemos obtener una representación visual de la topología del sistema, la cual se muestra en la figura 2.7. Cabe destacar

que en dicho esquema faltaría incluir también la GPU disponible como elemento de procesamiento.

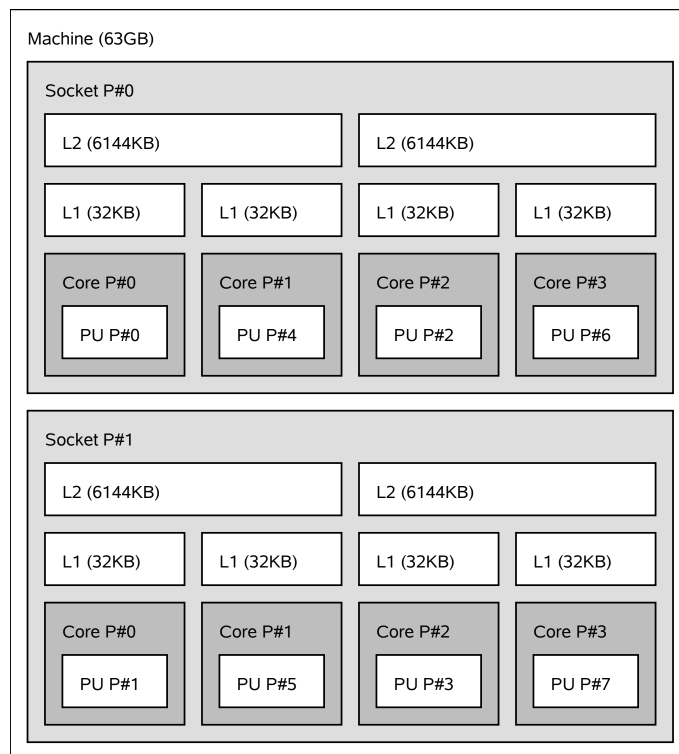

Figura 2.7: Topología de *geatpc2* obtenida con `lstopo`

2.2.2. Cluster *hipatia*

El *Servicio de Diseño Industrial y Cálculo Científico*, que forma parte del *Servicio de Apoyo a la Investigación Tecnólogica* de la *Universidad Politécnica de Cartagena* dispone del servidor *hipatia* para realizar cálculos científicos, el cuál también emplearemos para realizar evaluaciones de las rutinas. Se trata de un cluster para cálculo de alto rendimiento con 40 procesadores Intel Xeon (152 cores en total), 300 Gbytes de RAM y 8,7 Terabytes de almacenamiento; y con las siguientes características [6]:

- Sistema Operativo Linux RedHat.
- Gestor de Trabajos de usuario Torque/Maui.
- 14 nodos con 2 Intel Xeon Quad-Core a 2,80 GHz y FSB de 1600 MHz.
- 2 nodos con 4 Intel Xeon Quad-Core a 2,93 GHz y FSB a 1066 MHz.
- 2 nodos con 2 Intel Xeon 5160 Dual-Core a 3,00 GHz y FSB a 1333 MHz.
- Interconexión Infinibad 4X DDR2 de todos los nodos de cálculo.
- Almacenamiento compartido Fiber Channel a 4 Gbps.
- Acceso remoto a través de Internet.

Los nodos a los que se envían trabajos a través del sistema de colas son los primeros 14 (nodos del *cn5* al *cn18*) y los 2 siguientes (nodos *cn3* y *cn4*). El resto son para trabajos en modo interactivo (nodos *cn19* y *cn20*).

La organización topológica de cada uno los nodos de *hipatia* del *cn5* al *cn18* es igual a la topología de *geatpc2* mostrada en la figura 2.7, pero con 16 GB de RAM en puesto de 63 GB. En cuanto a la topología de los nodos *cn3* y *cn4*, podemos observarla en la figura 2.8. Esta es similar a la de los nodos anteriores, pero con el doble de procesadores y con algo menos de memoria cache L2.

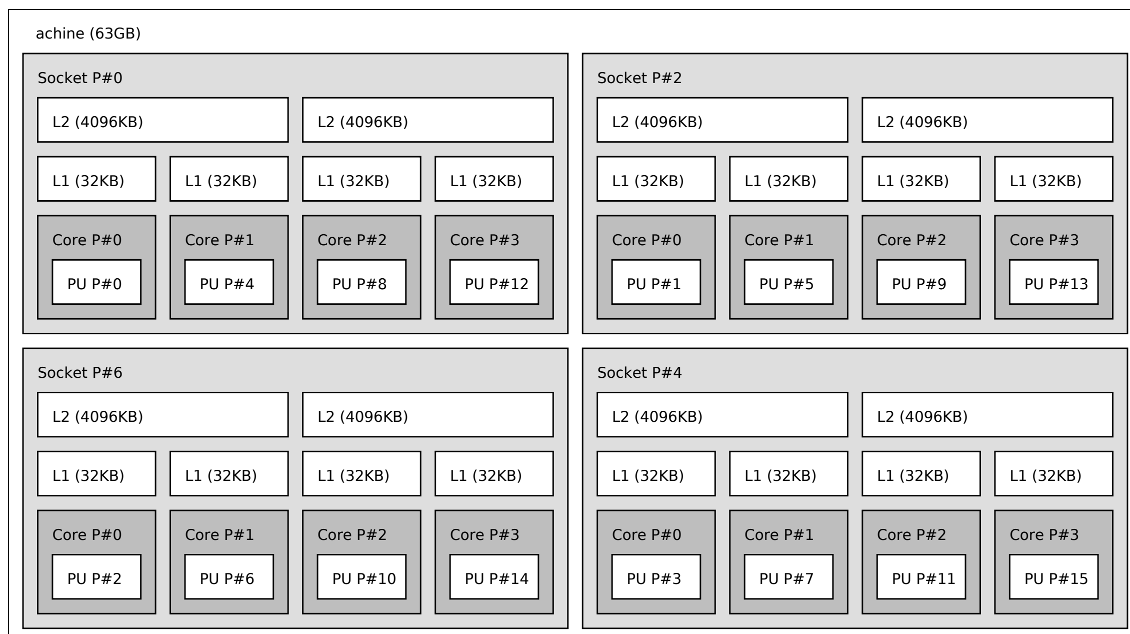

Figura 2.8: Topología de *hipatia* (nodos *cn3* y *cn4*) obtenida con `lstopo`

Para la evaluación de las rutinas MPI se utilizarán, como máximo, 2 nodos con 8 procesos en cada uno (nodos del *cn5* al *cn18*); y para OpenMP uno de los nodos con 16 cores (*cn3* o *cn4*).

2.2.3. Servidores *marte* y *mercurio*

Además de *luna*, el *Grupo de Computación Científica y Programación Paralela* de la *Universidad de Murcia* dispone de otros servidores de cálculo, situados en el *Departamento de Informática y Sistemas* de la *Facultad de Informática*, para las labores de investigación del grupo. Dos de estas máquinas son las "gemelas" *marte* y *mercurio*, en las que también realizaremos experimentos a lo largo del proyecto. Cada uno de estos sistemas cuenta con los siguientes recursos:

- Sistema Operativo Linux Ubuntu 10.04 i686 para arquitecturas de 32 bits.

- 1 procesador AMD Phenom II X6 1075T —6 cores en memoria compartida— a una frecuencia de reloj de 3 GHz y con arquitectura de 64 bits. Caches L1 y L2 privadas a cada core de 64 KB y 512 KB, respectivamente, y L3 compartida por todos los núcleos de 6 MB.

- Un módulo de memoria RAM, de 15 GB en *marte* y de 8066 MB en *mercurio*.
- Una tarjeta NVIDIA GeForce GTX 590 con 2 dispositivos de aceleración gráfica, con CUDA Compute Capability 2.0. Cada uno de ellos cuenta con 16 multiprocesadores de 32 cores, haciendo un total de 512 CUDA cores a 1,21 GHz. La cantidad total de memoria disponible para cada GPU es de 1,5 GB.

En la figura 2.9 mostramos visualmente la topología de *marte* proporcionada por `lstopo`, a la cual añadiríamos las 2 GPUs de la tarjeta gráfica como unidades de procesamiento extra. La topología de *mercurio* no se muestra porque es idéntica pero con menos memoria RAM.

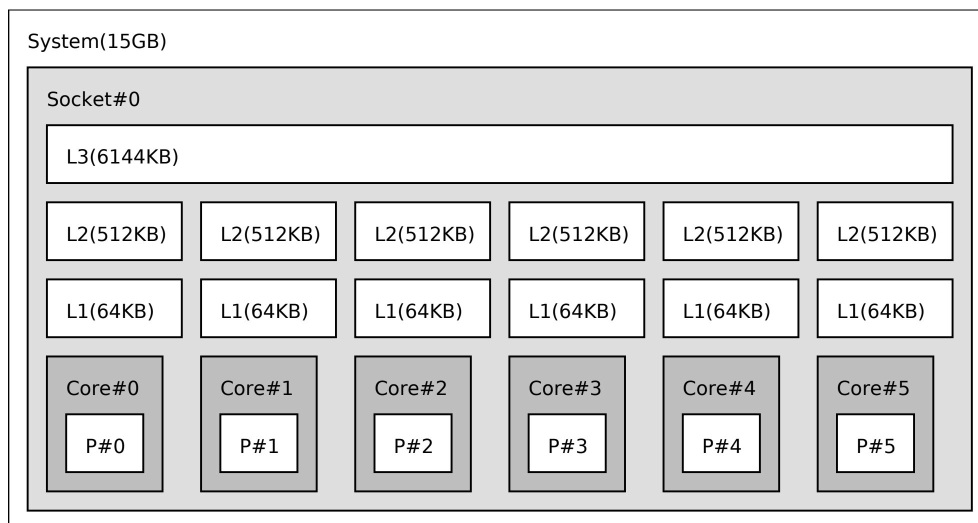

Figura 2.9: Topología de *marte* obtenida con `lstopo`

Por último, debemos destacar que ambas máquinas están conectadas en red entre sí y también con *luna*, lo que nos permite utilizar MPI sobre un sistema homogéneo si utilizamos solamente *marte* y *mercurio*, o heterogéneo si además contamos con *luna*.

Capítulo 3

Rutinas de cálculo de funciones de Green

En este capítulo se mostrará un análisis teórico de las rutinas hasta ahora desarrolladas, especificando el esquema algorítmico que siguen y modelando el tiempo teórico de ejecución, con objeto de predecir cómo se comportarían en un sistema real. Primero explicaremos todas las implementaciones para las funciones de Green unidimensionales y, después, las correspondientes a las bidimensionales.

El estudio de estas rutinas nos servirá para identificar, en capítulos posteriores, nuevas oportunidades de paralelismo que surjan de la combinación de varias de las implementaciones expuestas a continuación, y también para el diseño de un método de autotuning basado en los diferentes modelos teóricos del tiempo de ejecución de cada una de las rutinas.

3.1. Funciones de Green unidimensionales

Para las funciones de Green de la guía de placas paralelas en una dimensión disponemos de rutinas secuenciales escritas en Fortran y C++, y paralelas que utilizan los modelos ofrecido por OpenMP y CUDA para C++. El código Fortran fue proporcionado por el *Grupo de Electromagnetismo Aplicado a las Telecomunicaciones* de la *Universidad Politécnica de Cartagena* y, a partir de él, se desarrollaron en [21] las diferentes versiones en C++. Con respecto a las rutinas paralelas, tenemos dos implementaciones en OpenMP —de grano fino y de grano grueso—, una en CUDA y, finalmente, una versión híbrida que emplea al mismo tiempo los modelos de OpenMP y CUDA, beneficiándose de las ventajas de ambos.

3.1.1. Algoritmo secuencial

Para ilustrar el comportamiento del algoritmo secuencial, nos centraremos en la implementación C++, obviando la escrita en Fortran, ya que es similar y no está optimizada. El esquema algorítmico en pseudocódigo de esta implementación lo podemos ver en el código 3.1. Básicamente consiste en leer los parámetros de entrada al problema y aplicar la función de Green para cada una de las `n*m` parejas de puntos fuente y observación

(función `sbyparts()`). Dentro de esta función, llamamos a `spectgf_dsum()`, la cual entra en un bucle de `nmod` iteraciones, una para cada término de la serie. Debemos destacar que este parámetro del problema realmente es dependiente de la distancia entre los puntos involucrados en el cálculo de la función de Green, pero para facilitar el estudio del algoritmo vamos a suponer que es un parámetro fijado a la entrada del problema.

Código 3.1: Pseudocódigo secuencial para calcular funciones de Green unidimensionales

```
main() {
    Leer parametros m, n, nmod, ...;
    GF_Capa2Dt_Tex_conv(n, m, nmod, ...);
}

GF_Capa2Dt_Tex_conv(n, m, nmod, ...) {
    para cada punto fuente de 1 hasta n {
        para cada punto observacion de 1 hasta m {
            sbyparts(nmod, ...);
        }
    }
}

sbyparts(nmod, ...) {
    spectgf_dsum(nmod, ...);
    Metodo Aceleracion Kummer;
}

spectgf_dsum(nmod, ...) {
    repetir nmod iteraciones {
        Operaciones exponenciales;
        Operaciones trignometricas;
    }
}
```

Si analizamos el código 3.1 detenidamente podemos concluir que el tiempo teórico de ejecución del mismo corresponde a la ecuación siguiente:

$$t(n,m,nmod) = \sum_{1}^{n}\sum_{1}^{m}\left(\sum_{1}^{nmod}(E+T)+K\right) = nm\,(nmod\,(E+T)+K) \qquad (3.1)$$

en la que n y m se corresponden con el número de puntos fuente y observación, respectivamente; $nmod$ es el número de modos para calcular en las series; y las constantes E, T y K representan en el código 3.1, respectivamente, al tiempo empleado en los cálculos exponenciales (línea 21) y trigonométricos (línea 22) de `spectgf_dsum()`, y el requerido para la aceleración de la suma mediante el método de Kummer (línea 16).

Una vez que hemos visto el funcionamiento del algoritmo secuencial para calcular funciones de Green en una dimensión, pasamos a describir las diferentes versiones paralelas de partida.

3.1.2. Paralelismo de grano fino con OpenMP

El primer paso para la paralelización del algoritmo secuencial se lleva a cabo para un sistema de memoria compartida con OpenMP. Se aplicará paralelismo a nivel de una única función de Green, lo que quiere decir que estamos tratando con paralelismo de grano fino. El esquema paralelo solamente se aplicará a la función `spectgf_dsum()`

vista en el código 3.1, manteniendo el resto de funciones exactamente iguales, por lo que el código 3.2 sólo muestra el nuevo pseudocódigo para esta función utilizando OpenMP.

Código 3.2: Pseudocódigo OpenMP de la función `spectgf_dsum()`

```
spectgf_dsum(nmod, ...) {
    #repartir iteraciones entre h hilos
    repetir nmod iteraciones {
        Operaciones exponenciales;
        Operaciones trignometricas;
    }
}
```

Basándonos en los códigos 3.1 y 3.2, podemos analizar teóricamente el tiempo de ejecución de esta versión paralela. En la ecuación siguiente se muestra dicho tiempo en función de n, m, $nmod$ y el número de hilos OpenMP empleados para la resolución, h:

$$t(n,m,nmod,h) = \sum_{1}^{n}\sum_{1}^{m}\left(\frac{\sum_{1}^{nmod}(E+T)}{h}+K\right) = nm\left(\frac{nmod\,(E+T)}{h}+K\right) \quad (3.2)$$

donde se debe notar que, por el momento, en las rutinas de este capítulo, solamente vamos a tener en cuenta el tiempo de computación, ignorando el tiempo de sobrecarga que conlleva la gestión de varios hilos ejecutándose en paralelo[1].

3.1.3. Paralelismo de grano fino con CUDA

Siguiendo la misma idea de paralelismo que en la sección 3.1.2, disponemos de una implementación para ejecutarla sobre una GPU NVIDIA con soporte para CUDA. En el código 3.3 podemos ver el pseudocódigo de la única rutina modificada respecto a la versión secuencial (código 3.1), `spectgf_dsum()`. Ahora lo que realiza es calcular las dimensiones del grid respecto al número de hilos por bloque (`tpb`) y del tamaño del problema (`nmod`) para después llamar al kernel. El grid tendrá un total de $\lceil \frac{nmod}{tpb} \rceil$ bloques de tpb hilos, cada uno de los cuales se encargará de calcular una iteración concreta.

Código 3.3: Pseudocódigo CUDA de la función `spectgf_dsum()`

```
spectgf_dsum(nmod, ...) {
    block = tpb;
    grid = ceil(nmod / tpb);

    spectgf_cuda_kernel<<<grid, block>>>(nmod, ...)
}

spectgf_dsum_kernel(nmod, ...) {
    tid = gridDim.x * blockId.x + threadId.x;

    if (tid < nmod) {
        Operaciones exponenciales;
        Operaciones trignometricas;
    }
}
```

[1]Para un estudio más detallado sobre modelado de costes de gestión de threads, véase [11].

Al igual que con OpenMP, partiendo de los códigos 3.1 y 3.3, podemos modelar el tiempo de ejecución de esta rutina. La ecuación siguiente muestra dicho modelo, en función de los parámetros del problema:

$$t(n, m, nmod) = \sum_{1}^{n}\sum_{1}^{m}\left(\sum_{1}^{nmod}(E_{GPU} + T_{GPU}) + K\right) = nm\,(nmod\,(E_{GPU} + T_{GPU}) + K) \tag{3.3}$$

En esta implementación debemos destacar que los valores de los tiempos E_{GPU} y T_{GPU} equivaldrían a las constantes E y T, pero obtenidas en función del rendimiento de la GPU, y no de la CPU. Sin embargo, la constante K valdrá lo mismo que en la implementación secuencial, siempre y cuando la CPU utilizada sea la misma.

3.1.4. Paralelismo de grano grueso con OpenMP

Una vez que hemos visto las alternativas desarrolladas que explotan un paralelismo de grano fino, es decir, a nivel de una única función de Green, veamos a continuación como se ha utilizado OpenMP para calcular simultáneamente varias funciones de Green. Partiendo de la versión secuencial (ver código 3.1), se modificaría solamente la función `GF_Capa2Dt_Tex_conv()` para paralelizar el bucle más externo, obteniendo el esquema del código 3.4.

Código 3.4: Pseudocódigo OpenMP de la función `GF_Capa2Dt_Tex_conv()`

```
1  GF_Capa2Dt_Tex_conv(n, m, nmod, ...) {
2      #repartir iteraciones entre h hilos
3      para cada punto fuente de 1 hasta n {
4          para cada punto observacion de 1 hasta m {
5              sbyparts(nmod, ...);
6          }
7      }
8  }
```

En la ecuación siguiente modelamos el tiempo de ejecución de esta versión de la rutina a partir de los códigos 3.1 y 3.4:

$$t(n, m, nmod, h) = \frac{\sum_{1}^{n}\sum_{1}^{m}\left(\sum_{1}^{nmod}(E + T) + K\right)}{h} = \frac{nm\,(nmod\,(E + T) + K)}{h} \tag{3.4}$$

Como se puede observar, no es más que el tiempo de ejecución secuencial (ver ecuación 3.1) dividido entre los h threads OpenMP empleados para la resolución del problema. Frente a la implementación OpenMP de grano fino, introduce la ventaja de paralelizar también los cálculos referentes a la constante K, además de reducir el coste de gestión de threads por cada iteración interna, aunque no esté reflejado en el modelo.

3.1.5. Paralelismo híbrido combinando OpenMP y CUDA

Por último, disponemos de una versión que utiliza paralelismo híbrido combinando OpenMP y CUDA, con el simple objetivo de obtener los beneficios de ambos modelos de programación. La idea seguida es bastante sencilla si partimos de la implementación mostrada en el apartado 3.1.4. Utilizaremos paralelismo de grano grueso con OpenMP,

generando un hilo adicional que se encargará de comunicarse con la GPU para calcular las funciones de Green mediante CUDA. Dicho hilo utilizará el paralelismo de grano fino expuesto en la sección 3.1.3. En el código 3.5 se muestra el esquema completo en pseudocódigo de esta versión.

Código 3.5: Pseudocódigo OpenMP+CUDA para calcular funciones de Green unidimensionales

```
1  main() {
2      Leer parametros m, n, nmod, ...;
3      GF_Capa2Dt_Tex_conv(n, m, nmod, ...);
4  }
5
6  GF_Capa2Dt_Tex_conv(n, m, nmod, ...) {
7      #repartir iteraciones entre h+1 hilos
8      para cada punto fuente de 1 hasta n {
9          para cada punto observacion de 1 hasta m {
10             sbyparts(nmod, ...);
11         }
12     }
13 }
14
15 sbyparts(nmod, ...) {
16     if (omp_get_thread_num() == 0)
17         spectgf_dsum_cuda(nmod, ...);    // Hilo CUDA
18     else
19         spectgf_dsum(nmod, ...);         // Hilos OpenMP
20     Metodo Aceleracion Kummer;
21 }
22
23 spectgf_dsum(nmod, ...) {
24     repetir nmod iteraciones {
25         Operaciones exponenciales;
26         Operaciones trignometricas;
27     }
28 }
29
30 spectgf_dsum_cuda(nmod, ...) {
31     block = tpb;
32     grid = ceil(nmod / tpb);
33
34     spectgf_dsum_kernel<<<grid, block>>>(nmod, ...)
35 }
36
37 spectgf_dsum_kernel(nmod, ...) {
38     tid = gridDim.x * blockId.x + threadId.x;
39
40     if (tid < nmod) {
41         Operaciones exponenciales;
42         Operaciones trignometricas;
43     }
44 }
```

A partir del código mostrado, podemos analizar teóricamente el tiempo de ejecución esperado en este caso, el cúal se corresponde con la ecuación siguiente:

$$t(n, m, nmod, h) = \frac{\sum_1^n \sum_1^m \left(\sum_1^{nmod} (E+T) + K\right)}{h + S_{GPU/CPU}} = \frac{nm\,(nmod\,(E+T)+K)}{h + S_{GPU/CPU}} \tag{3.5}$$

donde $S_{GPU/CPU}$ representa al speedup obtenido en CUDA respecto a la implementación secuencial.

3.2. Funciones de Green bidimensionales

Con respecto a las funciones de Green bidimensionales de la guía rectangular, se dispone, al igual que en el caso unidimensional, de implementaciones secuenciales escritas en Fortran y C++. En cuanto a las versiones paralelas, contamos con una implementación de cada uno de los modelos de programación paralela descritos en la sección 2.1: OpenMP, MPI y CUDA; explotando en todas ellas un paralelismo de grano fino, es decir, al igual que en el caso unidimensional, a nivel de una única función de Green.

3.2.1. Algoritmo secuencial

Al igual que con las funciones de Green en una dimensión, solamente vamos a comentar en este apartado la versión secuencial escrita en C++, ya que la versión en Fortran es el mismo algoritmo pero sin la optimización previa basada en la eliminación de código redundante. En el código 3.6 se muestra el pseudocódigo correspondiente a esta implementación. Lo primero que hace es leer los parámetros de entrada al problema y aplicar las tres funciones de inicialización: `modos_grecta()`, `reduce_modos_grecta()` e `ini_ewald()`. Los principales parámetros a considerar serán: `x` e `y` como el número de puntos en los ejes X e Y, respectivamente, en el plano de la observación; `nmod` el número de modos a calcular en la parte espectral de la serie; y `n` y `m` como el número de imágenes en los ejes X e Y, respectivamente, utilizados para calcular la parte espacial. El resto de parámetros se considerarán constantes en el estudio, así como el valor de `nmod` fijado a 100, igual que en [21].

Código 3.6: Pseudocódigo secuencial para calcular funciones de Green bidimensionales

```
1  main() {
2      Leer parametros x, y, nmod, n, m, ...;
3      modos_grecta(...);
4      reduce_modos_grecta(...);
5      ini_ewald(...);
6      GF_wg_ewald_xy_ord(x, y, nmod, n, m, ...);
7  }
8
9  GF_wg_ewald_xy_ord(x, y, nmod, n, m, ...) {
10     para cada punto en plano observacion de 1 hasta x {
11         para cada punto en plano observacion  de 1 hasta y {
12             repetir nmod iteraciones {
13                 Parte espectral de la funcion de Green;
14             }
15
16             para cada imagen en el eje Y de -m hasta m {
17                 para cada imagen en el eje X de -n hasta n {
18                     Parte espacial de la funcion de Green;
19                 }
20             }
21         }
22     }
23 }
```

Tras la fase de inicialización, se llama a la función `GF_wg_ewald_xy_ord()`, que calcula la función de Green para cada uno de los `x*y` puntos del plano de la observación. Cada una de estas tendrá una parte espectral y otra espacial, como ya se había adelantado. Para modelar el tiempo de ejecución de esta implementación solamente tendremos

en cuenta la función `GF_wg_ewald_xy_ord()`, descartando las tres de inicialización, ya que su coste es despreciable con respecto al cómputo de las funciones de Green. La ecuación siguiente muestra dicho tiempo teórico:

$$t(x,y,nmod,n,m) = \sum_{1}^{x}\sum_{1}^{y}\left(\sum_{1}^{nmod} S_1 + \sum_{-m}^{m}\sum_{-n}^{n} S_2\right) = xy\,(nmod \cdot S_1 + (2m+1)(2n+1)S_2) \quad (3.6)$$

En ella, identificamos las constantes S_1 y S_2 para representar en el código 3.6 al tiempo empleado en cada iteración de los cálculos de la parte espectral (línea 13) y espacial (línea 18), respectivamente. En ambas zonas, las operaciones son realizadas sobre números reales y complejos.

Visto el funcionamiento del algoritmo secuencial para calcular funciones de Green bidimensionales, pasamos a describir las diferentes implementaciones paralelas de las que disponemos inicialmente.

3.2.2. Paralelismo de grano fino con OpenMP

Disponemos de 3 implementaciones paralelas conformes al modelo de programación de OpenMP. Todas ellas reparten el cálculo de la serie de una única función de Green entre varios hilos, siendo el resultado final la suma de todas estas partes. Para realizar esta suma en la misma variable compartida, cada una de estas implementaciones sigue un mecanismo distinto:

- Utilización de una sección crítica OpenMP, lo cual supone sincronizar todos los hilos para el acceso.
- Calcular cada parte de la serie en variables privadas a cada hilo y realizar una reducción manual al final.
- Misma idea que la anterior, pero utilizando la cláusula `reduction` al definir la región paralela, de forma que sea OpenMP el encargado de realizar la reducción.

La implementación que mejor rendimiento obtuvo es la última, por lo que nos centraremos en ella para nuestro estudio. El pseudocódigo de la función `GF_wg_ewald_xy_ord()` se muestra en el código 3.7. Las regiones paralelas que se definen son dos, una para la parte espectral y otra para la espacial. Las líneas 8 y 16 representan la reducción que realizaría OpenMP automáticamente al indicar la cláusula `reduction` en la definición de la región paralela, como ya se ha mencionado.

Una vez que hemos visto el funcionamiento de este algoritmo paralelo, podemos modelar el tiempo teórico de ejecución del mismo:

$$\begin{aligned} t(x,y,nmod,n,m,h) &= \sum_{1}^{x}\sum_{1}^{y}\left(\frac{\sum_{1}^{nmod} S_1}{h} + \frac{\sum_{-m}^{m}\sum_{-n}^{n} S_2}{h} + R_{OMP}(h)\right) \\ &= xy\left(\frac{nmod \cdot S_1 + (2m+1)(2n+1)S_2}{h} + R_{OMP}(h)\right) \end{aligned} \quad (3.7)$$

donde nos encontramos con R_{OMP}, una función que representa el tiempo necesario para realizar las dos operaciones de reducción con h hilos en OpenMP de las partes espectral y espacial de la función de Green, que por el momento no vamos a modelar.

Código 3.7: Pseudocódigo OpenMP de la función `GF_wg_ewald_xy_ord()`

```
GF_wg_ewald_xy_ord(x, y, nmod, n, m, ...) {
    para cada punto en plano observacion de 1 hasta x {
        para cada punto en plano observacion  de 1 hasta y {
            #repartir iteraciones entre h hilos
            repetir nmod iteraciones {
                Parte espectral de la funcion de Green;
            }
            #reducir parte espectral

            #repartir iteraciones entre h hilos
            para cada imagen en el eje Y de -m hasta m {
                para cada imagen en el eje X de -n hasta n {
                    Parte espacial de la funcion de Green;
                }
            }
            #reducir parte espacial
        }
    }
}
```

3.2.3. Paralelismo de grano fino con CUDA

Con respecto a CUDA, partimos con dos implementaciones. La primera utiliza solamente un kernel para calcular la parte espacial, y la segunda incluye otro para calcular la parte espectral. El mejor rendimiento se obtuvo para la versión con un único kernel, ya que al fijar el parámetro *nmod* a 100 no hay suficiente carga computacional como para que sea necesario la paralelización de la parte espectral. Como en este proyecto también mantenemos fijo este parámetro, solamente ilustraremos la primera implementación, visible en el código 3.8 en forma de pseudocódigo.

Código 3.8: Pseudocódigo CUDA de la función `GF_wg_ewald_xy_ord()`

```
GF_wg_ewald_xy_ord(x, y, nmod, n, m, ...) {
    para cada punto en plano observacion de 1 hasta x {
        para cada punto en plano observacion  de 1 hasta y {
            repetir nmod iteraciones {
                Parte espectral de la funcion de Green;
            }

            // <<<total imagenes en el eje Y, total imagenes en el eje X>>>
            spacialgf_cuda_kernel<<<2*m + 1, 2*n + 1>>>(...);

            para cada imagen en el eje Y de -m hasta m {
                para cada imagen en el eje X de -n hasta n {
                    Reducir parte espacial de la funcion de Green;
                }
            }
        }
    }
}

spacialgf_cuda_kernel(...) {
    m = blockIdx.x - (gridDim.x - 1) / 2;
    n = threadIdx.x - (blockDim.x - 1) / 2;

    Parte espacial de la funcion de Green para imagenes (m, n);
}
```

Al contrario que en el caso unidimensional, esta implementación aprovecha la naturaleza del problema junto con el modelo de ejecución de CUDA para definir un grid con un bloque por cada imagen en el eje Y y un hilo (dentro de cada bloque) por cada imagen en el eje X para posteriormente llamar al kernel. Después, se realiza la reducción de la parte espacial de la función de Green.

Visto el funcionamiento de esta implementación paralela, pasamos a modelar el tiempo teórico necesario para la ejecución del mismo:

$$\begin{aligned} t(x, y, nmod, n, m) &= \sum_{1}^{x}\sum_{1}^{y}\left(\sum_{1}^{nmod} S_1 + \sum_{-m}^{m}\sum_{-n}^{n} S_{2(GPU)} + \sum_{-m}^{m}\sum_{-n}^{n} R\right) \\ &= xy\left(nmod \cdot S_1 + (2m+1)(2n+1)\left(S_{2(GPU)} + R\right)\right) \end{aligned} \tag{3.8}$$

donde nos encontramos con la nueva constante $S_{2(GPU)}$, que es equivalente a S_2, pero acorde al rendimiento de la GPU. También tenemos a la constante R, la cual representa el coste de una iteración de la reducción en secuencial de la parte espacial de la función de Green (ver línea 13 del código 3.8), que por sí solo es un tiempo despreciable, pero como el número de iteraciones depende del número de imágenes en los ejes X e Y es importante tenerlo en cuenta.

3.2.4. Paralelismo de grano fino con MPI

Por último, también disponemos de una implementación con MPI para sistemas de memoria distribuida de esta rutina, con paralelismo a nivel de una única función de Green en la parte espacial de la serie. En el código 3.9 mostramos el pseudocódigo de esta rutina, donde podemos ver que el proceso maestro (el cero) es el que calcula la parte espectral y después cada proceso MPI establece sus límites de trabajo para la parte espacial, para reducirlo después con la función `MPI_Reduce()` de la librería de MPI.

A continuación, modelamos el tiempo de ejecución teórico de este algoritmo paralelo:

$$\begin{aligned} t(x, y, nmod, n, m, p) &= \sum_{1}^{x}\sum_{1}^{y}\left(\sum_{1}^{nmod} S_1 + \frac{\sum_{-m}^{m}\sum_{-n}^{n} S_2}{p} + R_{MPI}(p)\right) \\ &= xy\left(nmod \cdot S_1 + \frac{(2m+1)(2n+1)S_2}{p} + R_{MPI}(p)\right) \end{aligned} \tag{3.9}$$

donde nos encontramos con R_{MPI}, una función que representa el tiempo necesario para realizar la reducción con p procesos en MPI (línea 15 del código 3.9) de la parte espacial de la función de Green, que por el momento no vamos a modelar.

Para finalizar, destacamos que las únicas comunicaciones realizadas entre los diferentes procesos son las correspondientes a la operación de reducción comentada en el párrafo anterior.

Código 3.9: Pseudocódigo MPI de la función `GF_wg_ewald_xy_ord()`

```
GF_wg_ewald_xy_ord(x, y, nmod, n, m, ...) {
    id = MPI_Comm_rank();
    p = MPI_Comm_size();

    para cada punto en plano observacion de 1 hasta x {
        para cada punto en plano observacion  de 1 hasta y {
            if (id == 0) {
                repetir nmod iteraciones {
                    Parte espectral de la funcion de Green;
                }
            }

            spacialgf(id, p, n, m, ...);

            MPI_Reduce(...);    // Reducir parte espacial
        }
    }
}

spacialgf(id, p, n, m, ...) {
    total_size = 2 * m + 1;
    work_size = ceil(total_size / p);

    start = id * work_size - m;
    end = min(start + work_size, total_size - m);

    para cada imagen en el eje Y de start hasta end {
        para cada imagen en el eje X de -n hasta n {
            Parte espacial de la funcion de Green;
        }
    }
}
```

Capítulo 4

Evaluación de las rutinas previas

Concluido el primer análisis teórico de las rutinas paralelas desarrolladas en [21], procedemos a evaluar dichas rutinas en otros sistemas distintos a donde fueron desarrolladas. Esto tiene como principal objetivo validar los resultados previos, de forma que obtengamos una visión general del funcionamiento de las rutinas que no esté sujeta a un único sistema. Al finalizar el capítulo, además de obtener una comparativa del rendimiento de las rutinas entre sistemas diferentes, tendremos una serie de resultados que podrán ser empleados para desarrollar metodologías de autotuning.

Al igual que en el capítulo anterior, se estudiarán primero las funciones de Green unidimensionales y posteriormente las bidimensionales.

4.1. Funciones de Green unidimensionales

Como ya se introdujo en la sección 3.1, partimos de 2 implementaciones secuenciales —Fortran original y C++ optimizada— y 4 paralelas —OpenMP (grano fino y grueso), CUDA, e híbrida combinando OpenMP y CUDA— de las rutinas que calculan funciones de Green de la guía de placas paralelas en una dimensión, cuyo rendimiento será evaluado en un nuevo sistema, *geatpc2* (descrito en sección 2.2.1).

Primero, evaluaremos si este sistema experimenta una mejora en la rutina al realizar su traducción y optimización de Fortran a C++. Para ello, emplearemos los compiladores de Intel disponibles para estos lenguajes, `ifort` e `icpc`, respectivamente. Los parámetros variados en este estudio serán n (nº de puntos fuente), m (nº de puntos observación) y $nmod$ (nº de modos o términos de la serie).

Los resultados que obtenemos se muestran en la figura 4.1, donde podemos comprobar que la implementación C++ llega hasta casi doblar el tiempo de ejecución de la implementación original Fortran al ir aumentando el tamaño del problema. A este fenómeno, que es el inverso al speedup, lo denominamos *slowdown* y no ocurría en [21], donde se experimentaba todo lo contrario. Una de las razones por la que esto puede estar ocurriendo sería las versiones de los compiladores empleados. Otra diferencia a tener en cuenta es el sistema operativo, pues ahora estamos empleando una versión de linux para 64 bits y anteriormente se empleó una versión de 32 bits. A estas hipótesis podríamos añadir las diferencias que pueden existir en la arquitectura del hardware subyacente.

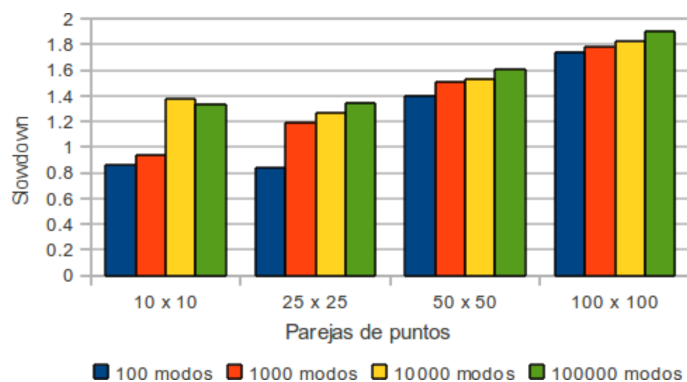

Figura 4.1: Slowdown producido en las funciones de Green unidimensionales al adaptar el código Fortran a C++ (*geatpc2*)

Tras comprobar que en este sistema la adaptación del código desde Fortran a C++ ha resultado un factor negativo, procedemos a estudiar la mejora obtenida con las implementaciones paralelas, donde comenzaremos empleando el número máximo de núcleos del sistema, 8, para las implementaciones que utilizan OpenMP. En la figura 4.2 se muestran cuatro gráficas que comparan el speedup obtenido para cada versión paralela respecto de la versión secuencial en C++ (se hará siempre así mientras no especifiquemos lo contrario), para diferentes tamaños de problema.

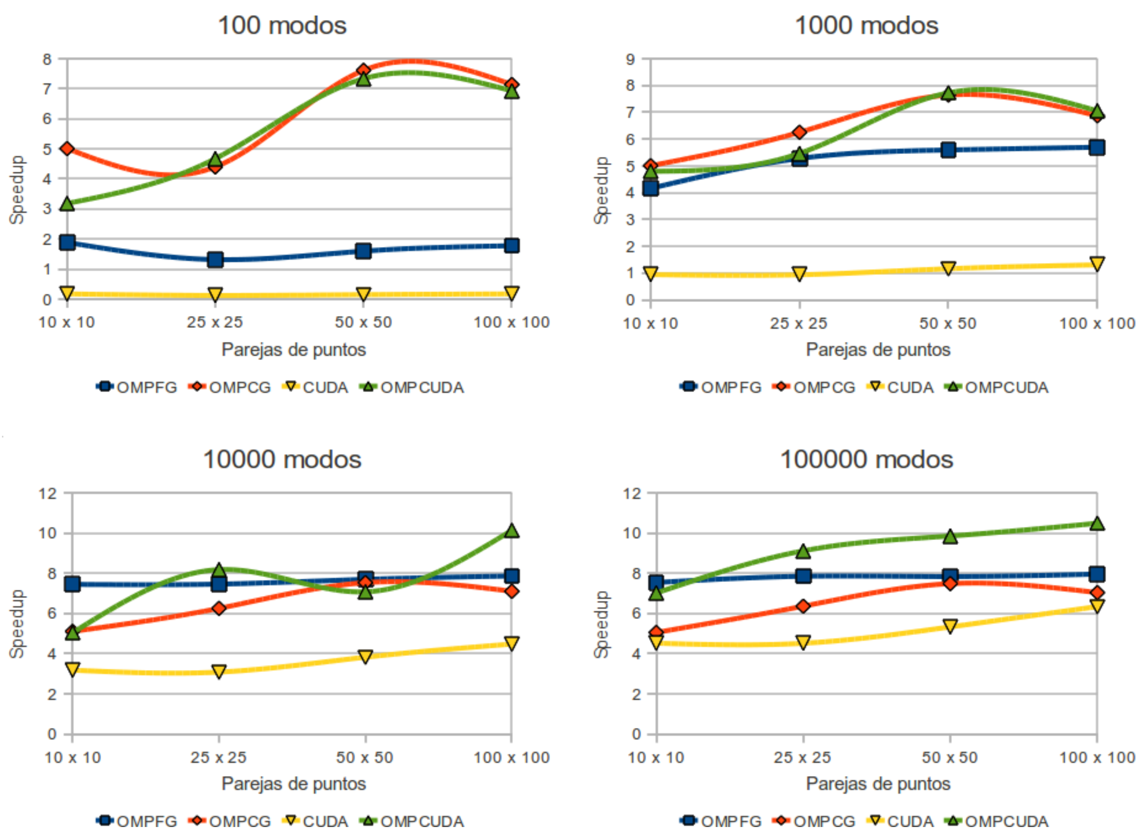

Figura 4.2: Comparativa de los speedups de las diferentes implementaciones paralelas de las funciones de Green unidimensionales (*geatpc2*)

Tras observar las gráficas detenidamente, notamos que CUDA es el modelo que peores resultados obtiene en todos los casos, aunque para los tamaños de problema más altos obtiene una mejora considerable, llegando hasta valores de speedup de 6.

La paralelización con OpenMP a nivel de una única función de Green (*OMPFG*) mejora el algoritmo al aumentar el número de modos, lo que es lógico ya que la carga de cómputo que se distribuye a los hilos son las iteraciones (modos) necesarias para calcular la serie. En cuanto a la implementación OpenMP que calcula varias funciones de Green de forma simultánea (*OMPCG*), comprobamos que supera en rendimiento a la versión anterior para un número bajo de modos, ocurriendo lo contrario para un número alto. El deterioro del rendimiento que se produce al utilizar paralelismo de grano grueso se produce al dividir entre 8 cores un número de iteraciones con una alta carga de trabajo que no es múltiplo de 8, algo que no se produce con grano fino aunque haya que generar y destruir los hilos para cada par de puntos, pues el número de modos es más alto que el de puntos fuente y debido a ello el reparto de la carga de trabajo es más equitativo en la implementación de grano fino.

Con respecto a la implementación híbrida que combina el paralelismo de grano grueso de OpenMP con el paralelismo de grano fino de CUDA (*OMPCUDA*), podemos considerarla como ganadora para un número alto de modos, ya que consigue superar a las implementaciones OpenMP, sobrepasando incluso un speedup de 10. Para un número bajo de modos el rendimiento está solamente un poco por debajo de la implementación OpenMP de grano grueso. Por ello, en términos generales podríamos decir que es la que mejor se comporta.

4.1.1. Comparación de rendimiento entre `double` y `float`

Acabamos de ver la evaluación del rendimiento de las rutinas paralelas que calculan funciones de Green en una dimensión, pero no hemos mencionado un detalle. Al utilizar una GPU que no soporta números reales en doble precisión, la rutina CUDA está empleando datos en simple precisión para realizar los cálculos. Sin embargo, las versiones secuencial y OpenMP sí que utilizan `double`, por lo que la comparación podría no resultar justa y, por este motivo, vamos a comprobar el rendimiento que se obtendría si utilizáramos `float` en las implementaciones secuencial y OpenMP, ya que se considera que este tipo de datos es suficiente para la precisión que se necesita en la aplicación en la que estamos trabajando.

El resultado podemos observarlo en la figura 4.3, donde mostramos el speedup alcanzado por las versiones que utilizan reales en simple precisión respecto de las que emplean doble precisión. En esta comprobamos que la utilización de un tipo de datos u otro resulta prácticamente irrelevante, al menos en este sistema, ya que los speedups alcanzados oscilan entre 0,6 y 1,6. Las mejoras en el rendimiento que se producen son para tamaños de problema grandes, probablemente debido a un mejor uso del espacio en las memorias cache, ya que al ser necesario un espacio menor se evitan los fallos de cache que se provocan al tener que reemplazar una línea de cache por falta de espacio.

Como el rendimiento con `float` no mejora excesivamente al obtenido con `double`, continuaremos con el estudio de las rutinas empleando para el cálculo números reales de doble precisión.

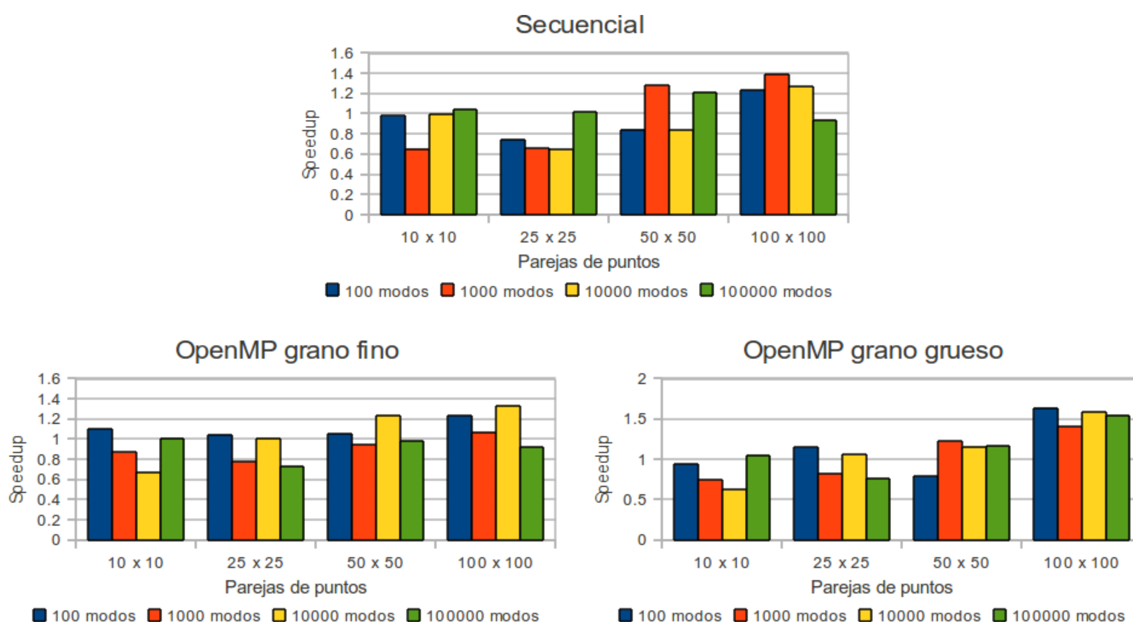

Figura 4.3: Speedup alcanzado en las implementaciones secuencial y OpenMP (8 threads) de las funciones de Green unidimensionales al utilizar `float` en lugar de `double` (*geatpc2*)

4.1.2. Evolución del speedup

En los experimentos anteriores hemos ejecutado las rutinas paralelas utilizando el mayor número posible de unidades de procesamiento disponibles en el sistema, pero esto no tiene por qué garantizarnos el mejor resultado posible en la práctica, ya que al incrementar el número de hilos en ejecución el tiempo de creación y eliminación de hilos también se incrementará. Debido a esto, vamos a estudiar empíricamente la evolución del speedup de todas las implementaciones que utilizan OpenMP al variar el número de cores empleados para la resolución del problema, teniendo en cuenta que la implementación híbrida añade uno extra para solicitar cálculos a la GPU.

En la figura 4.4 se muestran cuatro gráficas como extracto de los resultados obtenidos. Éstas recogen ejemplos de problemas de tamaño pequeño (100 modos y 10×10 puntos) hasta tamaño grande (100000 modos y 100×100 puntos). En ellas podemos observar como para 100 modos la implementación OpenMP de grano fino sufre un speedup muy por debajo del ideal y, además, el rendimiento a partir del séptimo core empeora. También es especialmente importante este experimento para averiguar que, para 100000 modos, usando conjuntamente CUDA y OpenMP podemos alcanzar un speedup superior a 12 simplemente utilizando, junto a la GPU, 3 cores, frente al que obtenemos empleando el sistema completo, que es alrededor de 10,5. Esto se debe a que utilizar el sistema completo añade un hilo más que gestionar y que además ocupa uno de los cores mientras envía y recibe datos hacia y desde la GPU, evitando que pueda estar realizando cálculos.

Los tiempos de ejecución obtenidos en este apartado nos serán muy útiles posteriormente para poder desarrollar un mecanismo de autotuning que elija la rutina más adecuada entre las disponibles en tiempo de ejecución.

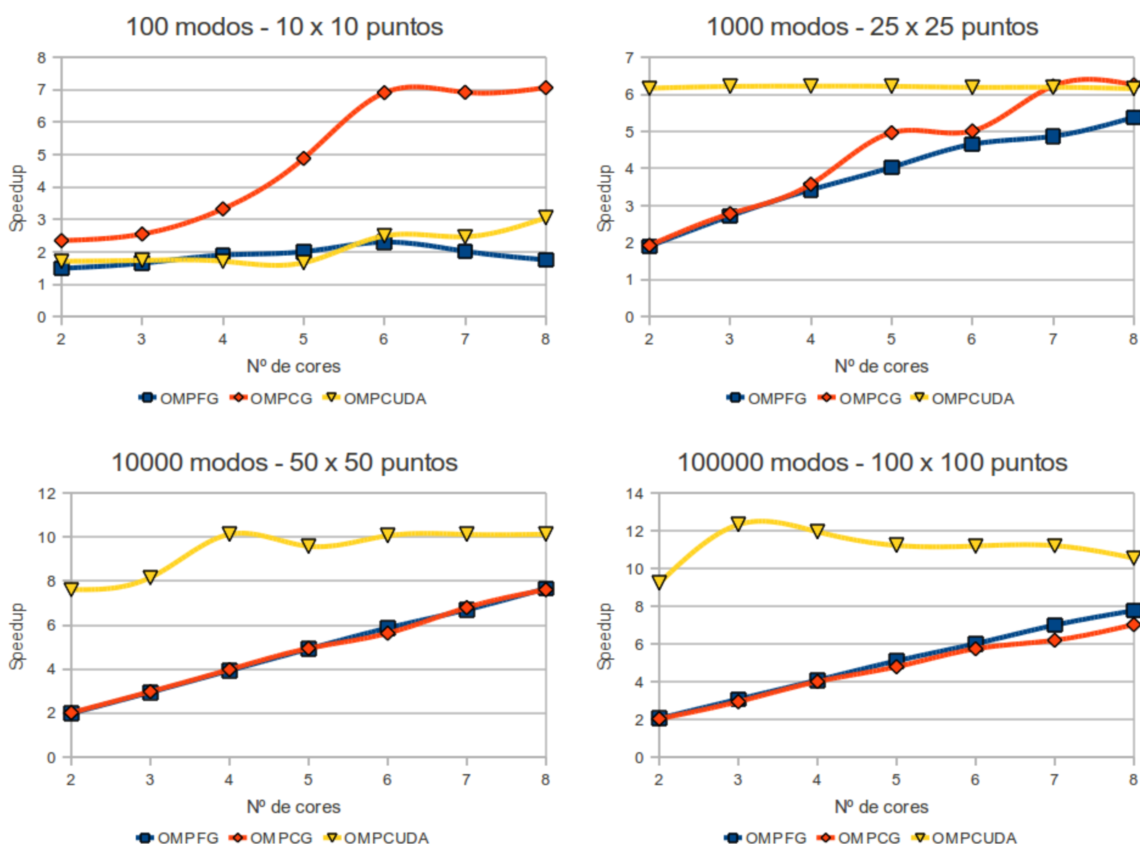

Figura 4.4: Evolución del speedup de las implementaciones OpenMP y OpenMP+CUDA de las funciones de Green unidimensionales (*geatpc2*)

4.1.3. Resumen de resultados y contraste con resultados previos

Para finalizar con este estudio de las rutinas de cálculo de funciones de Green unidimensionales mostramos la tabla 4.1, la cual nos indica las rutinas preferidas con el número de cores que se deben emplear (en formato RUTINA/CORES) para conseguir el máximo speedup para cada tamaño de problema estudiado.

$n \times m \setminus nmod$	100	1000	10000	100000
10×10	OMPCG/8	OMPFG/8	OMPCUDA/6	OMPCUDA/5
25×25	OMPCG/8	OMPCG/8	OMPCUDA/5	OMPCUDA/6
50×50	OMPCUDA/5	OMPCUDA/3	OMPCUDA/8	OMPCUDA/6
100×100	OMPCG/8	OMPCUDA/4	OMPCUDA/8	OMPCUDA/3

Tabla 4.1: Preferencia de rutina y número de cores para las funciones de Green unidimensionales en función del tamaño del problema (*geatpc2*)

Se observa claramente que la rutina híbrida OpenMP+CUDA es la ganadora para la mayoría de tamaños de problema, empleando cada vez un número diferente de hilos OpenMP. Para el resto de tamaños de problema (los más pequeños), la implementación de grano grueso con OpenMP es la que mejor se comporta. Solamente en un caso ha sido necesario emplear paralelismo OpenMP a nivel de una única función de Green.

Para complementar lo anterior, en la tabla 4.2 mostramos los speedups obtenidos con las rutinas preferidas, donde vemos por qué la implementación híbrida es, en términos

generales, mejor. Como podemos comprobar, en más de la mitad de los casos supera el speedup ideal de 8 que podríamos obtener con el máximo número de cores utilizando únicamente OpenMP.

$n \times m \setminus nmod$	100	1000	10000	100000
10×10	7,064	5,491	9,931	8,950
25×25	5,739	6,258	8,292	9,710
50×50	7,164	7,790	10,135	11,101
100×100	6,970	8,054	10,500	12,334

Tabla 4.2: Speedups obtenidos con las rutinas preferidas para el cálculo de las funciones de Green unidimensionales en función del tamaño del problema (*geatpc2*)

Finalmente, recuperando los tiempos de ejecución obtenidos en [21] para estas mismas rutinas podemos crear la tabla 4.3, que es homóloga a la tabla 4.1, pero con las preferencias para el sistema en que se evaluaron las rutinas, el servidor *luna* del *Grupo de Computación Científica y Programación Paralela*, utilizando siempre sus 4 cores.

$n \times m \setminus nmod$	100	1000	10000	100000
10×10	OMPCG/4	OMPCG/4	OMPFG/4	OMPCUDA/4
25×25	OMPCG/4	OMP[F,C]G/4	OMPCUDA/4	OMPCUDA/4
50×50	OMPCG/4	OMP[F,C]G/4	OMPCUDA/4	OMPCUDA/4
100×100	OMPCG/4	OMPCUDA/4	OMPCUDA/4	OMPCUDA/4

Tabla 4.3: Preferencia de rutina y número de cores para las funciones de Green unidimensionales en función del tamaño del problema (*luna*)

Si comparamos ambas tablas, se observa una clara similitud en cuanto a la preferencia por las rutinas. En general, predomina la predilección por la rutina híbrida OpenMP+CUDA cuando tratamos con un volumen de cálculo grande y por el grano grueso de OpenMP con problemas pequeños. *OMP[F,C]G* significa que tanto las rutinas OMPFG como OMPCG pueden ser empleadas en estos casos.

Sin embargo, a pesar de la similitud, existen pequeñas diferencias que nos indican que la rutina no se comporta exactamente igual en todos los sistemas, por lo que resultaría bastante interesante el desarrollo de un mecanismo de autotuning que permita escoger la mejor rutina en función de los parámetros del problema y de la máquina donde se esté ejecutando.

4.2. Funciones de Green bidimensionales

Concluido el estudio del rendimiento de las funciones de Green unidimensionales en *geatpc2*, pasamos a evaluar a continuación las rutinas que calculan las funciones de Green bidimensionales de una guía rectangular. Como ya vimos en el apartado 3.2, disponemos de 2 implementaciones secuenciales —Fortran original y C++ optimizada, igual que en el caso unidimensional— y 3 paralelas —OpenMP, CUDA y MPI— que explotan el paralelismo a nivel de una única función de Green. Como este problema es de mayor complejidad que el anterior y disponemos de una implementación en MPI, además de

evaluar las rutinas en *geatpc2* también las evaluaremos en *hipatia* que, como vimos en la sección 2.2.2, dispone de varios nodos para la ejecución en paralelo con MPI.

En primer lugar, y al igual que hicimos con las funciones unidimensionales, evaluaremos si se experimenta una mejora al traducir de Fortran a C++, empleando también los compiladores de Intel `ifort` e `icpc`. Esta vez, los parámetros a variar son x (número de puntos en el eje X), y (número de puntos en el eje Y), n (número de imágenes en el eje X) y m (número de imágenes en el eje Y). nm se especificará como número total de imágenes en el estudio.

En la figura 4.5 y 4.6 se muestran los speedups alcanzados en *geatpc2* e *hipatia*, respectivamente, al adaptar el código secuencial de Fortran a C++, que varía desde 1,5 hasta 2,5 en *geatpc2* y entre 1,2 y 1,8 en *hipatia*. Ambas máquinas superan el speedup que se producía al traducir el código Fortran a C++ en *luna* [21], que estaba entre 0,9 y 1,35.

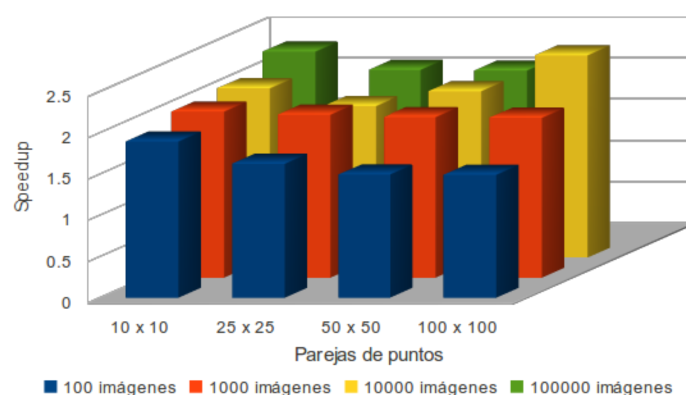

Figura 4.5: Speedup alcanzado en las funciones de Green bidimensionales al adaptar el código Fortran a C++ (*geatpc2*)

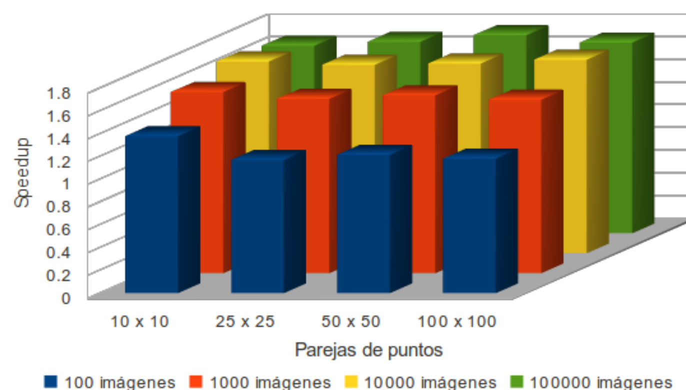

Figura 4.6: Speedup alcanzado en las funciones de Green bidimensionales al adaptar el código Fortran a C++ (*hipatia*)

Como hemos podido observar, en este caso, adaptar el código de Fortran a C++ sí que supone una mejora considerable en el rendimiento —para ambos sistemas—, en contraposición al resultado obtenido para las funciones de Green unidimensionales en *geatpc2*. Al parecer, una baja carga computacional, como el problema unidimensional, se comporta mejor con Fortran que con C++. Esto también explicaría por qué al aumentar el número de imágenes en el problema bidimensional, el speedup aumenta tanto en *geatpc2* como en *hipatia*. A continuación, pasamos a estudiar el speedup que se alcanza al utilizar las rutinas paralelas en lugar de la secuencial C++.

En la figura 4.7 se muestra el speedup alcanzado en *geatpc2* con las rutinas paralelas OpenMP (8 threads) y CUDA, sin tener en cuenta MPI de momento, al tratarse de un sistema con un único nodo. En ella podemos ver que el rendimiento de ambas depende principalmente del número de imágenes, lo cual es lógico, ya que lo que se está paralelizando es el bucle interno que itera sobre las imágenes. Sin embargo, al variar únicamente el número de puntos no se observan cambios significativos.

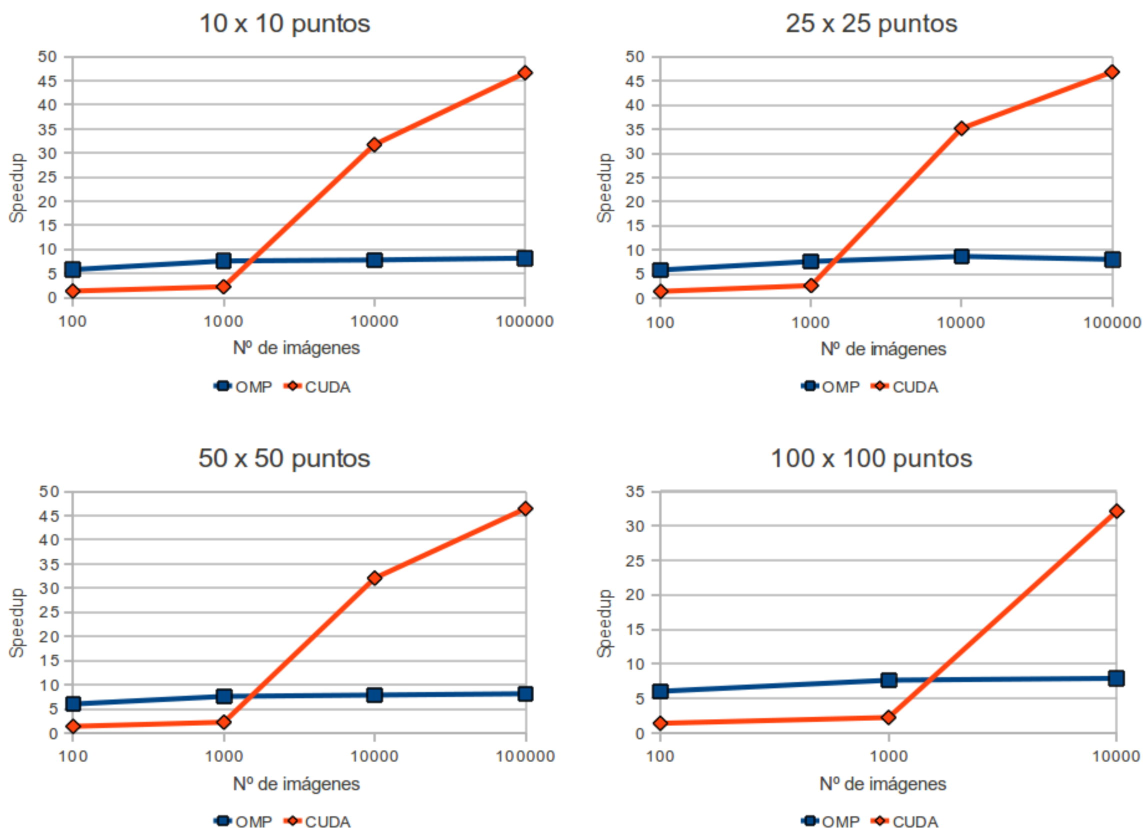

Figura 4.7: Comparativa de los speedups de las implementaciones paralelas OpenMP y CUDA de las funciones de Green bidimensionales (*geatpc2*)

En cuanto a los resultados obtenidos, OpenMP se comporta mejor que CUDA para números bajos de imágenes (100 y 1000). Pero este último modelo llega a alcanzar speedups superiores a 46 para 100000 imágenes, un resultado mucho más alentador que el que obtenemos para las funciones unidimensionales, en las que CUDA por sí solo no llegaba a superar a ninguna de las otras rutinas paralelas desarrolladas. Además, como se verá en la sección 4.2.2, la aceleración alcanzada con estas rutinas es similar a la que se obtuvo en el servidor *luna*.

Veamos ahora el comportamiento de las rutinas OpenMP y MPI en *hipatia*. Al no disponer de GPU, no evaluaremos la rutina CUDA en este sistema. Los speedups alcanzados para OpenMP (con 8 y 16 hilos) y MPI (16 procesos en 2 nodos, 8 procesos por nodo) podemos verlos en la figura 4.8, en función de los parámetros de entrada.

Antes de pasar a comentar los resultados obtenidos se destaca que OpenMP con 16 hilos utilizará un nodo de *hipatia* con 16 cores, mientras que OpenMP con 8 hilos y MPI utilizarán los de 8 cores. Debemos notar que aunque sean similares, tienen una gran diferencia entre sus anchos de banda del FSB y, por tanto, diferirán en el rendimiento obtenido para un mismo número de hilos/procesos en ejecución (sin tener en cuenta las

comunicaciones). También hemos de tener en cuenta que la rutina secuencial, a partir de la cual se calcula el speedup alcanzado puede haberse ejecutado en un nodo cualquiera del cluster, por lo que podríamos obtener speedups tanto por debajo como por encima de lo que esperamos para estas rutinas paralelas.[1]

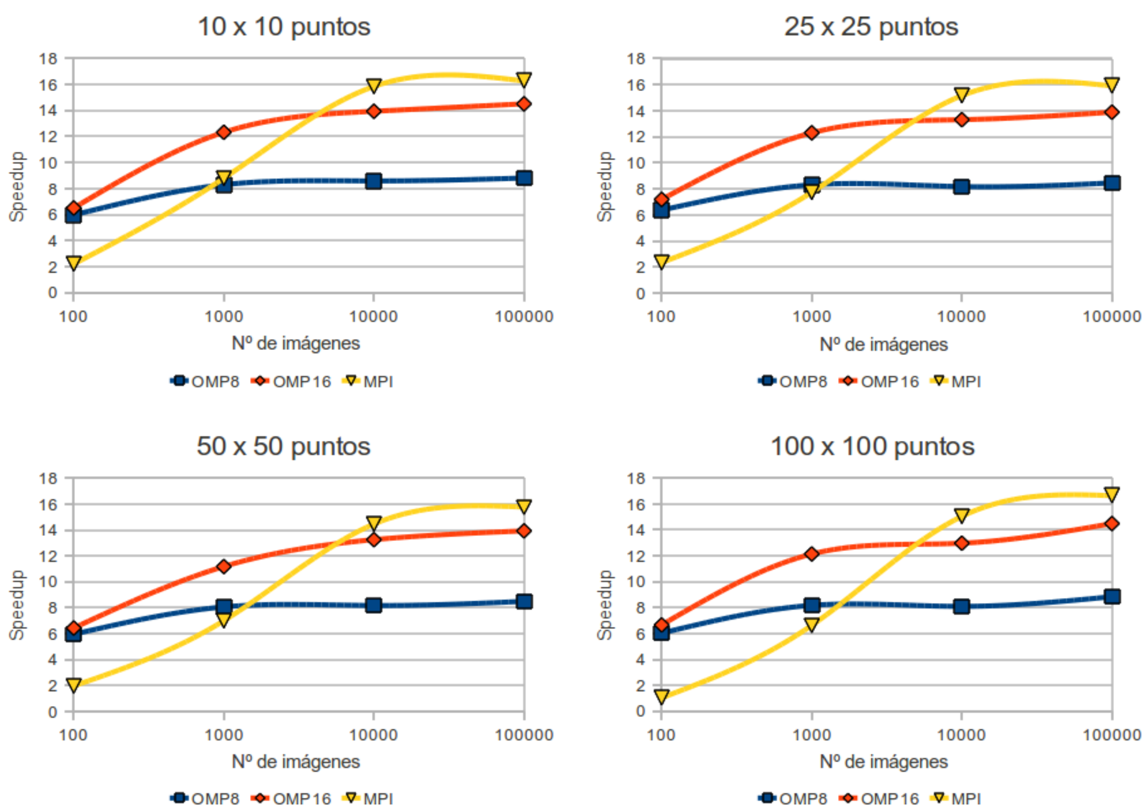

Figura 4.8: Comparativa de los speedups de las implementaciones paralelas OpenMP y MPI de las funciones de Green bidimensionales (*hipatia*)

La tendencia para todas estas rutinas es aumentar el rendimiento conforme aumenta el número de imágenes, ya que se consigue una granuralidad de cómputo mayor que permite dividir la carga de trabajo equitativamente entre los diferentes hilos/procesos. En cuanto a la variación del número de puntos, sólo afecta de forma significativa a MPI para pocas imágenes al introducir más comunicaciones sin aumentar los cálculos. Estas comunicaciones también son las causantes del bajo rendimiento de MPI frente a OpenMP para pocas imágenes, pues para 100 imágenes no alcanza ni un 2,5 de speedup. Sin embargo, para un número alto de imágenes, MPI se acerca al speedup ideal de 16 e incluso lo supera —también lo hace OpenMP con 8 hilos—, mientras que OpenMP con 16 hilos solamente llega a alcanzar un 14,5 de speedup, lo que se debe a lo ya comentado sobre la utilización de nodos con diferentes características.

Finalmente, a modo de contraste con el modelo teórico de las rutinas dado por las ecuaciones 3.7 y 3.9, se comprueba que, como se espera al utilizar paso de mensajes, la operación de reducción en MPI, modelada como $R_{MPI}(p)$, es más costosa que la reducción automática que realiza OpenMP, modelada como $R_{OMP}(h)$, para $p = h$. Si esto no fuera así, MPI alcanzaría speedups similares a los obtenidos con OpenMP para un número bajo de imágenes, al contrario de lo que está ocurriendo.

[1] Para una información más detallada sobre los nodos del cluster *hipatia*, véase la sección 2.2.2 de este proyecto.

4.2.1. Evolución del speedup

Al igual que con las funciones unidimensionales, hasta ahora hemos ejecutado las rutinas paralelas utilizando el mayor número posible de unidades de procesamiento disponibles en el sistema, pero esto no nos garantiza el mejor resultado posible en la práctica, ya que al incrementar el número de hilos/procesos en ejecución el tiempo de creación y eliminación de hilos/procesos también se incrementará, así como el de reducción, tanto en MPI ($R_{MPI}(p)$) como en OpenMP ($R_{OMP}(h)$). Debido a esto, vamos a estudiar empíricamente la evolución del speedup de las implementaciones OpenMP y MPI al variar el número de cores empleados para la resolución del problema.

Aunque hemos realizado el experimento en *geatpc2* e *hipatia*, sólo mostraremos los resultados obtenidos en *hipatia*, ya que en *geatpc2* solamente tenemos la rutina OpenMP para ver la evolución y se ha podido comprobar que el speedup aumenta linealmente con el número de cores en la mayoría de los casos.

En la figura 4.9 se muestran cuatro gráficas a modo de resumen de los resultados obtenidos. Éstas recogen ejemplos de problemas de tamaño pequeño (100 imágenes y 10×10 puntos) hasta tamaño grande (100000 imágenes y 100×100 puntos). Para OpenMP se muestra la evolución utilizando el nodo de 8 cores y el de 16.

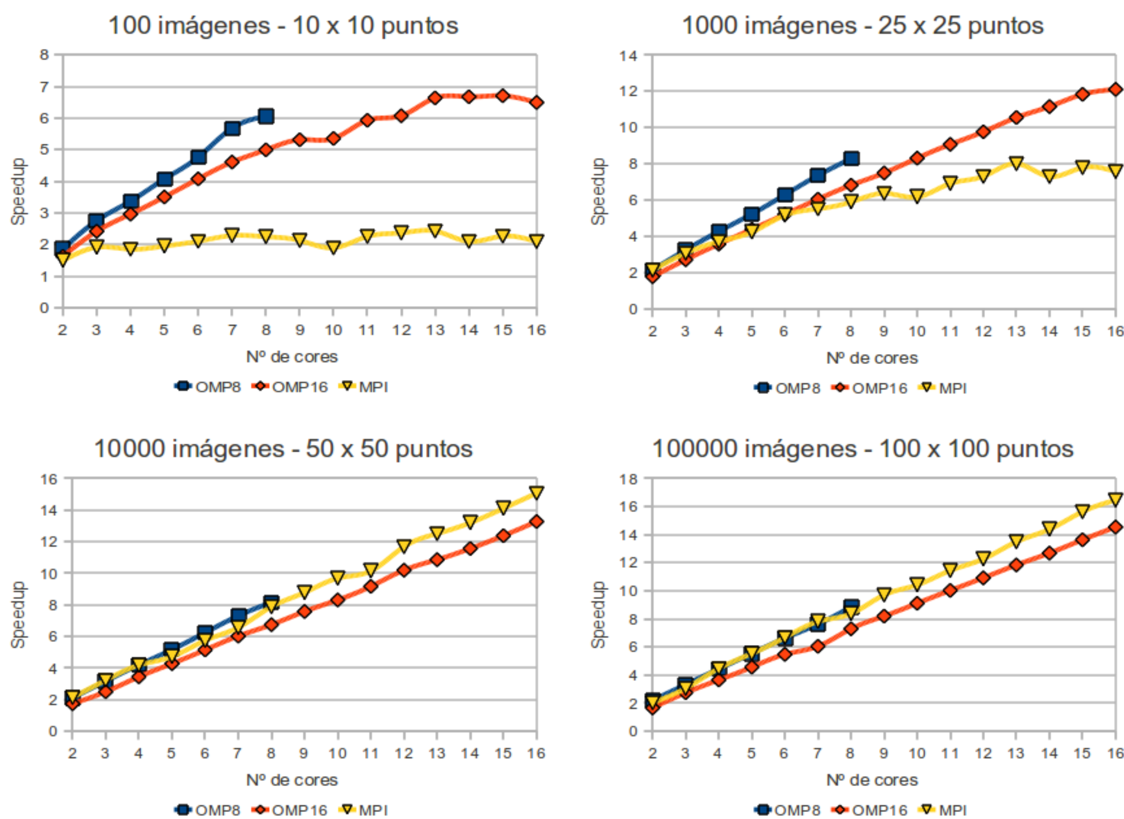

Figura 4.9: Evolución del speedup de las implementaciones OpenMP y MPI de las funciones de Green bidimensionales (*hipatia*)

Como se puede observar, el speedup de la implementación MPI es el que peor evoluciona para números bajos de imágenes, debido a la sobrecarga de comunicaciones frente a la poca carga computacional. Sin embargo, para valores altos el speedup evoluciona de forma prácticamente lineal. Comprobamos que OpenMP también sufre una penalización

en el rendimiento al aumentar el número de cores para 100 imágenes, aunque no tan alta como MPI. En cuanto al nodo utilizado en OpenMP, para un número de hilos inferior a 10, merece la pena emplear el nodo de 8 cores frente al de 16, pues el primero utilizando los 8 cores llega a igualar o incluso superar en rendimiento al último utilizando 10 cores. Esto se debe a que el nodo de 8 cores se beneficia de un mayor ancho de banda del FSB respecto al de 16, y por este motivo consigue un mejor rendimiento empleando un menor número de elementos de procesamiento.

Por último, para tamaños altos de problema, se detecta la similitud en la evolución de MPI y de OpenMP en el nodo de 8 cores, dado que se han ejecutado en nodos de las mismas características y porque la sobrecarga por comunicaciones en MPI es despreciable en este caso al haber una mayor carga computacional y realizarse las comunicaciones a través de la memoria compartida.

4.2.2. Resumen de resultados y contraste con resultados previos

Para finalizar con este estudio de las rutinas de cálculo de funciones de Green bidimensionales mostramos las tablas 4.4 y 4.5, las cuales nos indican las rutinas preferidas con el número de cores que se deben emplear (en formato RUTINA/CORES) para conseguir el máximo speedup para cada tamaño de problema estudiado en *geatpc2* e *hipatia*, respectivamente.

$x \times y \setminus nm$	100	1000	10000	100000
10×10	OMP/8	OMP/8	CUDA	CUDA
25×25	OMP/8	OMP/8	CUDA	CUDA
50×50	OMP/8	OMP/8	CUDA	CUDA
100×100	OMP/8	OMP/8	CUDA	

Tabla 4.4: Preferencia de rutina y número de cores para las funciones de Green bidimensionales en función del tamaño del problema (*geatpc2*)

En *geatpc2* comparamos solamente las rutinas OpenMP y CUDA, obteniendo que para un número bajo de imágenes se comporta mejor OpenMP (utilizando el número máximo de cores disponibles) que CUDA, debido a que este último necesita una mayor carga computacional para obtener una mejora considerable, como la que se produce para valores altos de imágenes.

$x \times y \setminus nm$	100	1000	10000	100000
10×10	OMP16/15	OMP16/16	MPI/16	MPI/15
25×25	OMP16/15	OMP16/16	MPI/16	MPI/16
50×50	OMP16/14	OMP16/16	MPI/16	MPI/16
100×100	OMP16/14	OMP16/16	MPI/16	MPI/16

Tabla 4.5: Preferencia de rutina y número de cores para las funciones de Green bidimensionales en función del tamaño del problema (*hipatia*)

En *hipatia* ocurre algo similar, solamente que es MPI el modelo que mejor se comporta para valores altos de imágenes, pues CUDA no se ha evaluado al no disponer *hipatia* de GPU. Debido a las comunicaciones, para un número bajo de imágenes se comporta mejor OpenMP. En casi todos los casos se utiliza el máximo de procesos/hilos, exceptuando

para valores bajos de imágenes en los que tener más hilos en OpenMP introduce una sobrecarga innecesaria. Como último detalle, resaltamos que OpenMP con 16 hilos no obtiene mejor resultado que MPI porque los nodos de 16 cores de *hipatia* tienen un menor rendimiento que los nodos de 8, que son los que usamos en MPI.

En las tablas 4.6 y 4.7 tenemos los valores máximos de speedup que se alcanzan con las rutinas expuestas antes, para *geatpc2* e *hipatia*, respectivamente. En la primera, podemos observar como CUDA llega a alcanzar speedups muy superiores a los que podríamos obtener con OpenMP utilizando 8 hilos. Y de ambas se puede deducir que se ha realizado paralelismo sobre el bucle de las imágenes, ya que el speedup óptimo aumenta con ellas, mientras que aumentar el número de puntos no afecta prácticamente al speedup.

$x \times y \setminus nm$	100	1000	10000	100000
10×10	5,815	7,612	31,725	46,635
25×25	5,858	7,613	35,152	46,834
50×50	6,007	7,590	32,041	46,433
100×100	6,030	7,637	32,124	

Tabla 4.6: Speedups obtenidos con las rutinas preferidas para el cálculo de las funciones de Green bidimensionales en función del tamaño del problema (*geatpc2*)

$x \times y \setminus nm$	100	1000	10000	100000
10×10	6,704	12,397	15,917	16,503
25×25	7,077	12,104	15,157	16,455
50×50	6,759	11,922	15,058	15,616
100×100	6,731	12,019	13,185	16,448

Tabla 4.7: Speedups obtenidos con las rutinas preferidas para el cálculo de las funciones de Green bidimensionales en función del tamaño del problema (*hipatia*)

Para finalizar, mostramos la tabla 4.8, que representa las rutinas preferidas en función del tamaño del problema en el servidor *luna*. Como vemos, las rutinas preferidas son similares a las de *geatpc2*, ya que son sistemas con características similares. La razón de preferir CUDA en vez de OpenMP para 1000 imágenes es simple: *luna* tiene la mitad de cores que *geatpc2* y, además, una GPU más potente. Estos dos factores hacen más sencillo a CUDA superar el speedup que se alcanza con OpenMP.

$x \times y \setminus nm$	100	1000	10000	100000
10×10	OMP/4	CUDA	CUDA	CUDA
25×25	OMP/4	CUDA	CUDA	CUDA
50×50	OMP/4	CUDA	CUDA	CUDA
100×100	OMP/4	CUDA	CUDA	

Tabla 4.8: Preferencia de rutina y número de cores para las funciones de Green bidimensionales en función del tamaño del problema (*luna*)

Como vemos, aunque los resultados son similares entre sistemas, no son exactamente iguales, como ocurría también en el problema unidimensional. Las diferencias se deben principalmente a la arquitectura del sistema y de los distintos dispositivos que lo conforman. Por ello, si queremos utilizar las rutinas en varios sistemas, puede ser muy útil

que éstas dispongan de un mecanismo de decisión en tiempo de ejecución para elegir la implementación más rápida en función de los parámetros de entrada al problema y de las características del sistema donde se están ejecutando, como se verá en el capítulo 6, dedicado a la autooptimización de estas rutinas. Además de esto, también resultaría interesante diseñar rutinas híbridas que combinen los tres paradigmas de programación paralela vistos hasta ahora y que permitan su compilación con un tipo de paralelismo u otro dependiendo de las características del sistema computacional donde se utilicen. El diseño de este tipo de rutinas se abordará en el capítulo 5.

Capítulo 5

Paralelismo híbrido

Una vez concluida la evaluación del rendimiento de las rutinas paralelas previas en otros sistemas distintos a donde se desarrollaron, en este capítulo trataremos de profundizar en el estudio del paralelismo que podemos introducir en las funciones de Green bidimensionales, que son las más costosas.

En el capítulo 4 se ha visto cómo se alcanzaban speedups de hasta 46 utilizando solamente CUDA en uno de los sistemas evaluados. Ahora lo que se pretende es ampliar el conjunto disponible de implementaciones paralelas para resolver el problema bidimensional. Concretamente, el objetivo del presente capítulo será desarrollar una implementación híbrida que, combinando el uso de varios modelos de programación paralela, aproveche la organización jerárquica de los sistemas actuales. Un ejemplo básico de sistema jerárquico aplicable a nuestro problema sería el sistema de la figura 5.1, en el que podemos identificar fuertes similitudes con *marte* y *mercurio* (descritos en la sección 2.2.3), puesto que se trata de dos nodos inerconectados entre sí y cada uno dispone de un procesador con varios núcleos y una GPU como elementos de procesamiento.

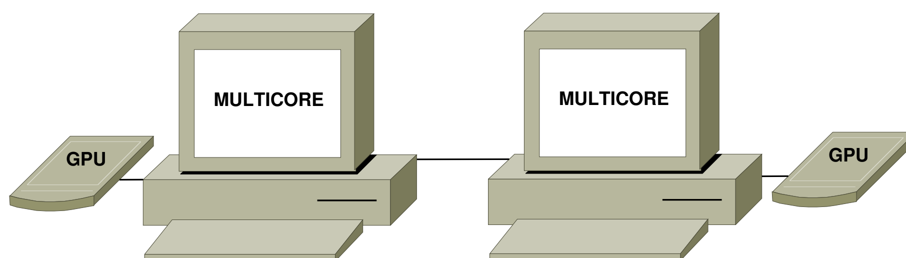

Figura 5.1: Ejemplo de un sistema computacional jerárquico básico

Esta nueva implementación utilizará simultáneamente los tres modelos de programación estudiados a lo largo de este proyecto: MPI, OpenMP y CUDA. El primero nos servirá para tener varios procesos distribuidos en diferentes nodos, el segundo para ejecutar varios hilos en los cores del mismo nodo compartiendo el espacio de memoria y el último para aprovechar la GPU instalada en cada nodo. De esta manera, se pretende mantener ocupados, en la medida de lo posible, a todos los recuros computacionales del sistema para aprovechar al completo la capacidad de cálculo de éste.

5.1. Diseño e implementación

Para el desarrollo de esta rutina híbrida se tomarán como punto de partida las rutinas secuencial y paralelas de la sección 3.2. Debido a que tratamos con varios modelos de programación paralela distintos entre sí, cada uno de ellos tendrá que explotar paralelismo a un nivel diferente al del resto. Por ello, no se trata simplemente de mezclar las implementaciones anteriores, ya que todas ellas explotaban el paralelismo al mismo nivel, al de una única función de Green —de grano fino—, sino que habrá que identificar paralelismo a diferentes niveles de granularidad para poder aplicarlos con cada uno de los modelos que van a intervenir.

Antes de comenzar con la descripción de la nueva rutina, nos remitimos al código 3.6 para recordar el esquema algorítimico de la implementación secuencial de la función que calcula las funciones de Green bidimensionales para varios puntos, `GF_wg_ewald_xy_ord()`. Una vez recordado, pasamos a explicar los niveles de paralelismo identificados y el modelo de programación encargado de aplicarlo en cada nivel, apoyándonos en el esquema en pseudocódigo de la rutina híbrida (ver código 5.1):

Grano grueso o paralelismo a nivel de varias funciones de Green. La idea a seguir es la misma que se aplicó con OpenMP en las funciones de Green unidimensionales (sección 3.1.4), donde se calculaban varias funciones en paralelo para diferentes pares de puntos. En este nivel, aplicaremos los modelos MPI y OpenMP para paralelizar el bucle externo (línea 12 del código 5.1), dando lugar a dos subniveles:

- Entre los diferentes procesos MPI se repartirán las iteraciones del bucle de forma estática (líneas 2–9 del código 5.1), estableciendo los límites en cada proceso, dando lugar a un bucle con un número de iteraciones $\frac{x}{p}$, donde x representa al número de puntos en el eje X y p al número de procesos MPI.
- El bucle anterior, a su vez, se paralelizará dinámicamente con OpenMP (línea 11 del código 5.1), generando h hilos por cada uno de los p procesos MPI.

Grano fino o paralelismo a nivel de una única función de Green. Este paralelismo ya ha sido explotado con los diferentes modelos de programación paralela estudiados, pero en esta rutina híbrida el paralelismo se extraerá con CUDA. Para ello, se emplearán g hilos OpenMP adicionales para invocar sucesivamente al kernel descrito en la sección 3.2.3 (línea 20 del código 5.1), de forma similar a como lo hace la rutina híbrida que utiliza conjuntamente OpenMP y CUDA para el cálculo de las funciones de Green unidimensionales (sección 3.1.5).

Como vemos en el código 5.1, el algoritmo se puede configurar con los parámetros `p` (procesos MPI), `h` (hilos OpenMP) y `g` (hilos OpenMP dedicados a CUDA). Además de ello, cada uno de los procesos MPI envía al proceso número 0 (*maestro*) las funciones de Green que ha calculado a través de la función `MPI_Gather()` de la librería de MPI.

Código 5.1: Pseudocódigo MPI+OpenMP+CUDA de la función `GF_wg_ewald_xy_ord()`

```
GF_wg_ewald_xy_ord(x, y, nmod, n, m, ...) {
    id = MPI_Comm_rank();
    p = MPI_Comm_size();

    total_size = x;
    work_size = ceil(total_size / p);

    start = id * work_size;
    end = min(start + work_size, total_size);

    #repartir iteraciones entre h+g hilos
    para cada punto en plano observacion de start hasta end {
        para cada punto en plano observacion  de 1 hasta y {
            repetir nmod iteraciones {
                Parte espectral de la funcion de Green;
            }

            if (omp_get_thread_num() < g) {
                // <<<total imagenes en el eje Y, total imagenes en el eje X>>>
                spacialgf_cuda_kernel<<<2*m + 1, 2*n + 1>>>(...);

                para cada imagen en el eje Y de -m hasta m {
                    para cada imagen en el eje X de -n hasta n {
                        Reducir parte espacial de la funcion de Green;
                    }
                }
            } else {
                para cada imagen en el eje Y de -m hasta m {
                    para cada imagen en el eje X de -n hasta n {
                        Parte espacial de la funcion de Green;
                    }
                }
            }
        }
    }

    MPI_Gather(...);  // Enviar al proceso 0 las funciones de Green
}
```

El tiempo de ejecución teórico de esta rutina lo podemos modelar de la siguiente forma:

$$\begin{aligned} t(x,y,nmod,n,m,p,h,g) &= \frac{\sum_1^x \sum_1^y \left(\sum_1^{nmod} S_1 + \sum_{-m}^{m} \sum_{-n}^{n} S_2\right)}{p(h+gS_{GPU/CPU})} + G_{MPI}(x,y,p) \\ &= \frac{xy(nmod \cdot S_1 + (2m+1)(2n+1)S_2)}{p(h+gS_{GPU/CPU})} + G_{MPI}(x,y,p) \end{aligned} \quad (5.1)$$

donde aparecen, además de los parámetros de entrada al problema, p, h y g como parámetros de configuración de la rutina, correspondientes a las variables de mismo nombre del código 5.1. Se puede apreciar fácilmente como este tiempo de ejecución no es más que el tiempo secuencial (ver ecuación 3.6) dividido entre $p(h+gS_{GPU/CPU})$, sumándole $G_{MPI}(x,y,p)$. El primer bloque representa al número total de hilos OpenMP (ph), más el speedup obtenido en CUDA respecto a la implementación secuencial ($S_{GPU/CPU}$) multiplicado por el número total de hilos dedicados a CUDA (pg). G_{MPI} es el tiempo necesario para realizar la operación `MPI_Gather()` (línea 37 del código 5.1) con p procesos MPI y el total de $x \times y$ funciones de Green que debe agrupar el proceso

maestro (el único punto donde se introducen explícitamente comunicaciones entre los diferentes procesos MPI).

Por último, se destaca que la posibilidad que ofrece la rutina para configurarla en función de los recursos del sistema a utilizar permite disponer de diferentes rutinas que exploten el paralelismo a mayor o menor nivel de hibridación, o incluso utilizar el algoritmo secuencial, aunque con algún retardo adicional producido por las instrucciones de control necesarias para el paralelismo utilizado. En la tabla 5.1 resumimos el total de rutinas, aclarando que para CUDA son necesarios g hilos OpenMP, pero si $h = 0$ no consideramos que la rutina esté aplicando OpenMP[1].

$p \setminus h+g$	$1+0$	$h+0$	$0+g$	$h+g$
1	SEQ	OMP	CUDA	OMP+CUDA
p	MPI	MPI+OMP	MPI+CUDA	MPI+OMP+CUDA

Tabla 5.1: Rutinas disponibles con las diferentes configuraciones de la implementación híbrida

5.2. Evaluación

En el apartado anterior se ha mostrado cómo se ha llevado a cabo la aplicación del paralelismo híbrido a las funciones de Green bidimensionales utilizando los modelos de programación paralela MPI, OpenMP y CUDA de manera conjunta. Hecho esto, pasamos a analizar la mejora que obtenemos con esta rutina para distintos valores de los parámetros de configuración en diferentes sistemas jerárquicos. Estos sistemas son *hipatia*, el cual ya hemos utilizado en el capítulo anterior, y el par *marte/mercurio*. En el primero, al no disponer de hardware gráfico, solamente se evaluarán las combinaciones que no utilicen CUDA.

5.2.1. Comparación de rendimiento entre `g++` e `icpc`

Hasta ahora, todos los sistemas utilizados para evaluar las rutinas disponían de uno o varios procesadores multinúcleo Intel. Debido a que los procesadores de *marte* y *mercurio* son AMD (visto en el apartado 2.2.3), vamos a realizar una comparativa del rendimiento entre los compiladores C++ de GNU (`g++`) e Intel (`icpc`) antes de comenzar con la evaluación de la rutina híbrida. El objetivo de este experimento es comprobar qué compilador de los dos anteriores se comporta mejor en este tipo de procesador. Si, al igual que se comprobó en *luna* [21], el compilador de Intel es, con diferencia, mejor que el de GNU, descartaremos por el momento solicitar la instalación de una alternativa específica para AMD en este sistema, como podría ser el compilador `openCC` [7].

Para realizar esta prueba ejecutaremos la implementación secuencial para diferentes valores de puntos e imágenes. En la figura 5.2 se muestra el speedup que se alcanza al utilizar el compilador de Intel en lugar del de GNU en *marte*, una máquina que no está formada por procesadores Intel, como ya hemos dicho.

[1] Esto lo hacemos para diferenciar un hilo OpenMP que actúa sin anidar paralelismo frente a otro que sí lo hace al realizar llamadas a un kernel de CUDA para que se ejecute en la GPU.

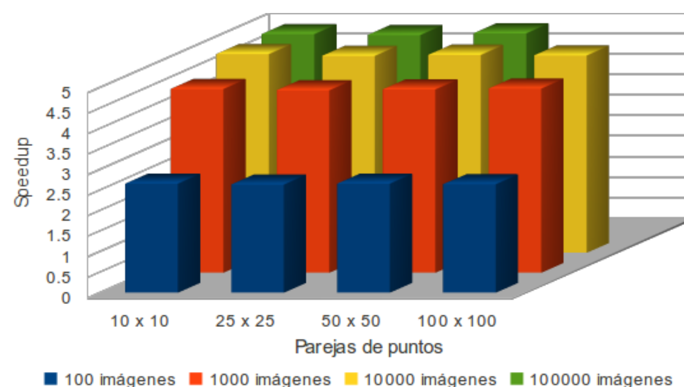

Figura 5.2: Speedup alcanzado en las funciones de Green bidimensionales al utilizar `icpc` en puesto de `g++` (*marte*)

Se observa claramente que si utilizamos el compilador de Intel en vez del de GNU se mejora el rendimiento de la rutina más de 2,5 veces, llegando casi a alcanzar speedups de 5 al ir aumentanado el número de imágenes sobre las que se calculan las funciones de Green. Como es una mejora más que aceptable, pese a que se trata un procesador AMD, utilizaremos el compilador de Intel de aquí en adelante para compilar las rutinas tanto en *marte* como en *mercurio*, ya que los procesadores son iguales.

5.2.2. Combinando MPI y OpenMP

Primero vamos a estudiar el primer nivel de hibridación, que corresponde a la combinación de MPI y OpenMP en la ejecución de la rutina, sin emplear CUDA, es decir, manteniendo el parámetro de configuración de la rutina $g = 0$ (ver columnas 2 y 3 en la tabla 5.1).

Utilizando un único nodo

Comenzaremos utilizando un único nodo, para que las comunicaciones entre procesos MPI no sean un factor muy influyente y así podamos ver que combinaciones de $p \times h$ son preferibles con poca sobrecarga de comunicaciones. En la figura 5.3 se muestran los speedups obtenidos en uno de los nodos de 8 cores de *hipatia* para varias configuraciones de p procesos MPI y h hilos OpenMP ($p \times h$ en la leyenda).

Como se puede ver, todas las configuraciones de procesos e hilos se comportan de forma similar para el mismo tamaño de problema, experimentando únicamente un ligero descenso del rendimiento al emplear un número mayor de procesos MPI. Algo similar ocurre en *marte*, como podemos comprobar en la figura 5.4.

Sin embargo, en *marte* solamente se deteriora el rendimiento al utilizar más procesos MPI en algunos casos e incluso de forma menos notable que en *hipatia*. Además, observamos que, a pesar de disponer de 6 cores, los speedups no llegan ni a 4. Una de las posibles causas de esto podría ser que el subsistema de memoria esté suponiendo un cuello de botella al aumentar el número de procesos/hilos implicados en el cálculo, ya que en *hipatia* el speedup sí que se acerca al ideal. Este cuello de botella también explicaría por qué en *marte* no se distingue apenas el descenso del rendimiento con un mayor número de procesos MPI.

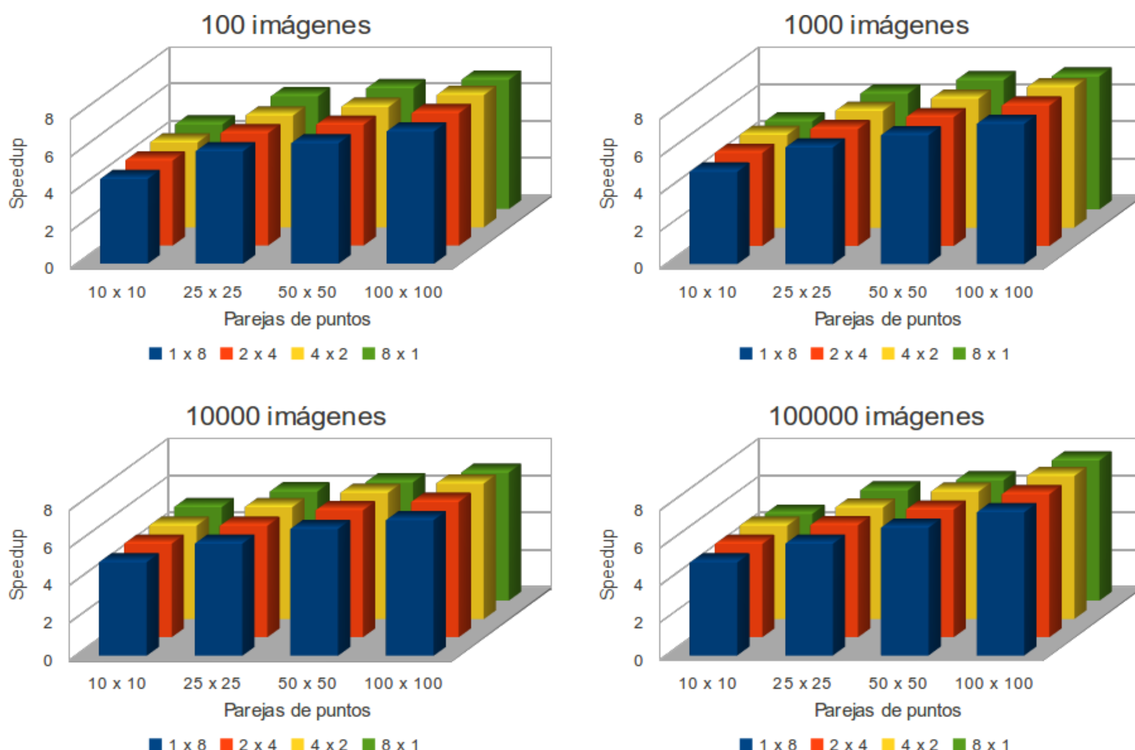

Figura 5.3: Speedups de la rutina híbrida MPI+OpenMP en un nodo (*hipatia*)

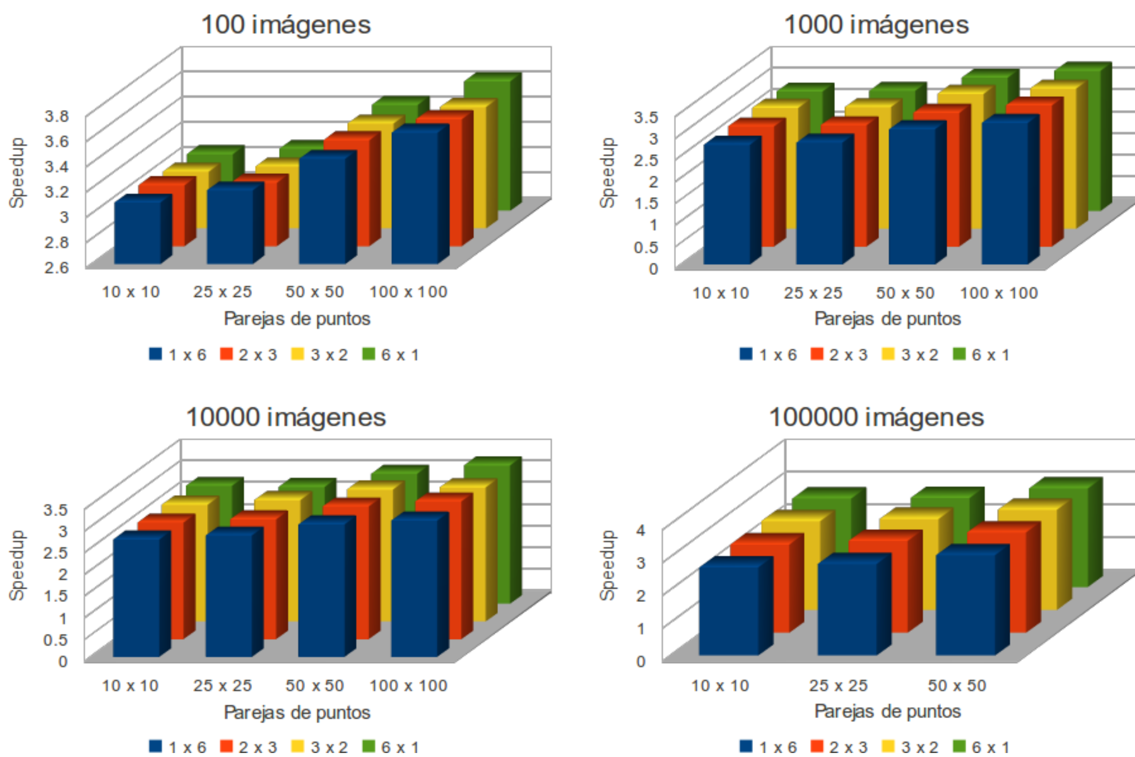

Figura 5.4: Speedups de la rutina híbrida MPI+OpenMP en un nodo (*marte*)

Utilizando dos nodos

A continuación vamos a ver el comportamiento de la rutina al utilizar dos nodos diferentes, para que la mitad de los procesos MPI estén ejecutándose en un nodo diferente al del proceso que recibe el resultado de la operación `MPI_Gather()`. De este modo,

se espera que haya una mayor sobrecarga en las comunicaciones con respecto al caso anterior. En la figura 5.5 se muestra el speedup alcanzado en *hipatia* utilizando el doble de procesos MPI que en el apartado anterior, ejecutando la mitad de ellos en uno de los nodos de 8 cores y la otra mitad en otro. Se puede observar una clara similitud con la figura 5.3, pero destacando que el speedup ideal queda algo más alejado en términos relativos que utilizando un nodo, al estar implicados más procesos en el cálculo.

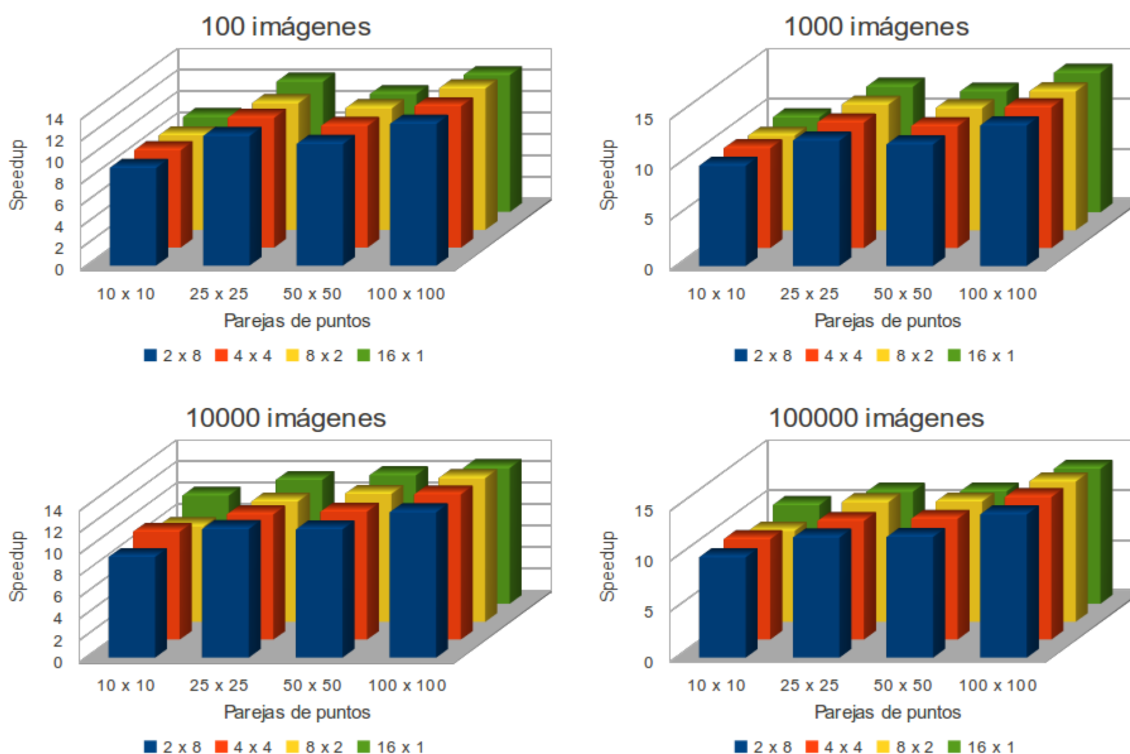

Figura 5.5: Speedups de la rutina híbrida MPI+OpenMP en dos nodos (*hipatia*)

En cuanto a *marte* y *mercurio*, podemos ver en la figura 5.6 que la rutina se comporta prácticamente igual que en *hipatia*, manteniendo la baja eficiencia que se obtenía al utilizar solamente *marte* como nodo único.

Como conclusión de este apartado, podemos decir que, en términos generales y manteniendo constante el producto ph, es preferible utilizar una relación baja de los parámetros $\frac{p}{h}$ frente a una alta.

5.2.3. Combinando MPI, OpenMP y CUDA

Una vez que se ha concluido el estudio del primer nivel de hibridación de la rutina, pasamos a evaluar la misma añadiendo en la ejecución la intervención de una GPU bajo el modelo de programación de CUDA (ver columnas 4 y 5 en la tabla 5.1). Dado que se ha comprobado que un menor número de procesos MPI es lo ideal en términos generales, realizaremos un estudio en *marte/mercurio* manteniendo fijo $p = 2$ —de forma que haya un proceso en cada nodo— y variando el número de hilos OpenMP generados por cada proceso MPI, obteniendo así una evolución del speedup en función del número de cores utilizados en cada nodo.

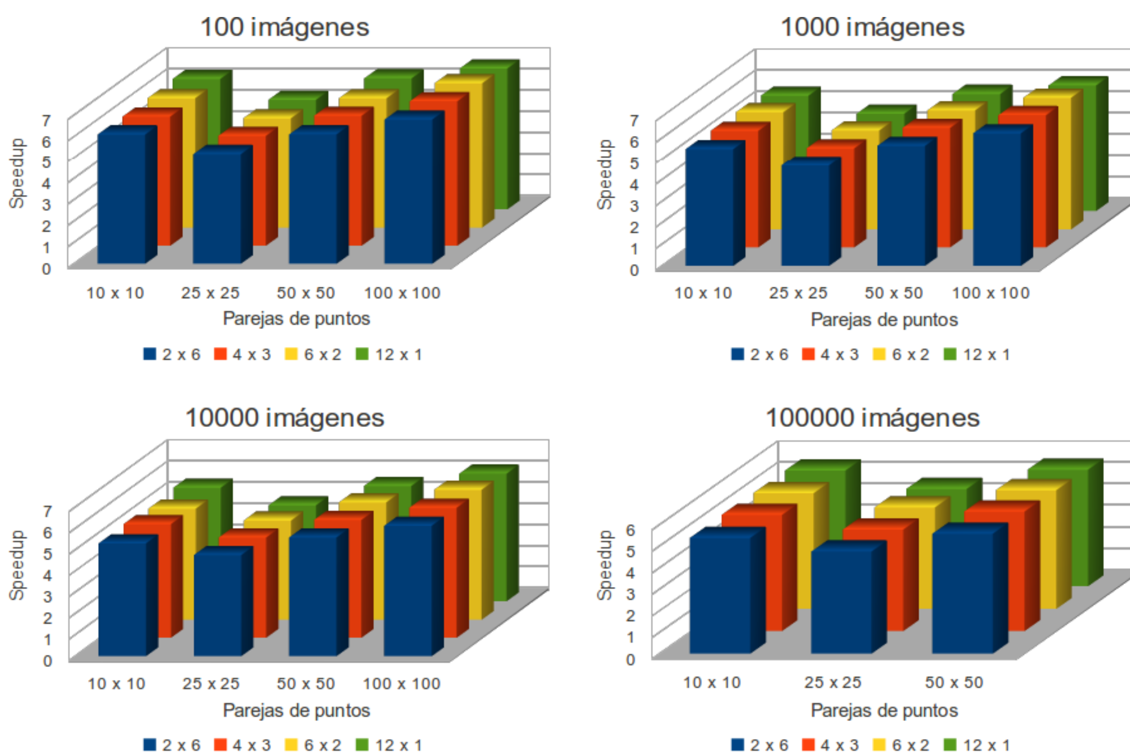

Figura 5.6: Speedups de la rutina híbrida MPI+OpenMP en dos nodos (*marte/mercurio*)

Utilizando un *kernel* por nodo

Primero estudiaremos la evolución del speedup añadiendo en cada nodo un hilo OpenMP extra que se encargue de realizar invocaciones al kernel para su ejecución en la GPU ($g = 1$), de forma que la rutina híbrida se ejecute empleando todos los paradigmas de programación paralela de forma simultánea. En la figura 5.7 se muestra una gráfica que recoge esta evolución.

Podemos ver como para 100 imágenes aumentar el número de hilos OpenMP hace que se incremente el speedup, especialmente para 100×100 puntos. Con 1000 imágenes vemos como se produce una cierta oscilación para 100×100 puntos entre 11 y 14 de speedup, pero no aumenta el rendimiento de forma significativa. Además, para 10×10 puntos el rendimiento cae al emplear un valor de $h > 0$. Esto es lo que ocurre también al calcular 10000 y 100000 imágenes en cada función de Green, siendo menor esta caída con un número mayor de puntos al poder extraer mayor paralelismo de grano grueso.

El que sea preferible utilizar solamente un hilo OpenMP dedicado a CUDA con un número alto de imágenes se debe a que en este sistema la gestión de varios hilos es costosa o existe un cuello de botella como puede ser el subsistema de memoria, tal y como ya se indicó en la sección 5.2.2. En cambio, para un número bajo de imágenes, esto no ocurre debido a que no hay suficiente carga computacional como para aprovechar las altas capacidades de cómputo que ofrece CUDA, algo que sí se aprovecha al aumentar el número de imágenes, que es donde se extrae el paralelismo de grano fino en la GPU, como se puede observar en la gráfica.

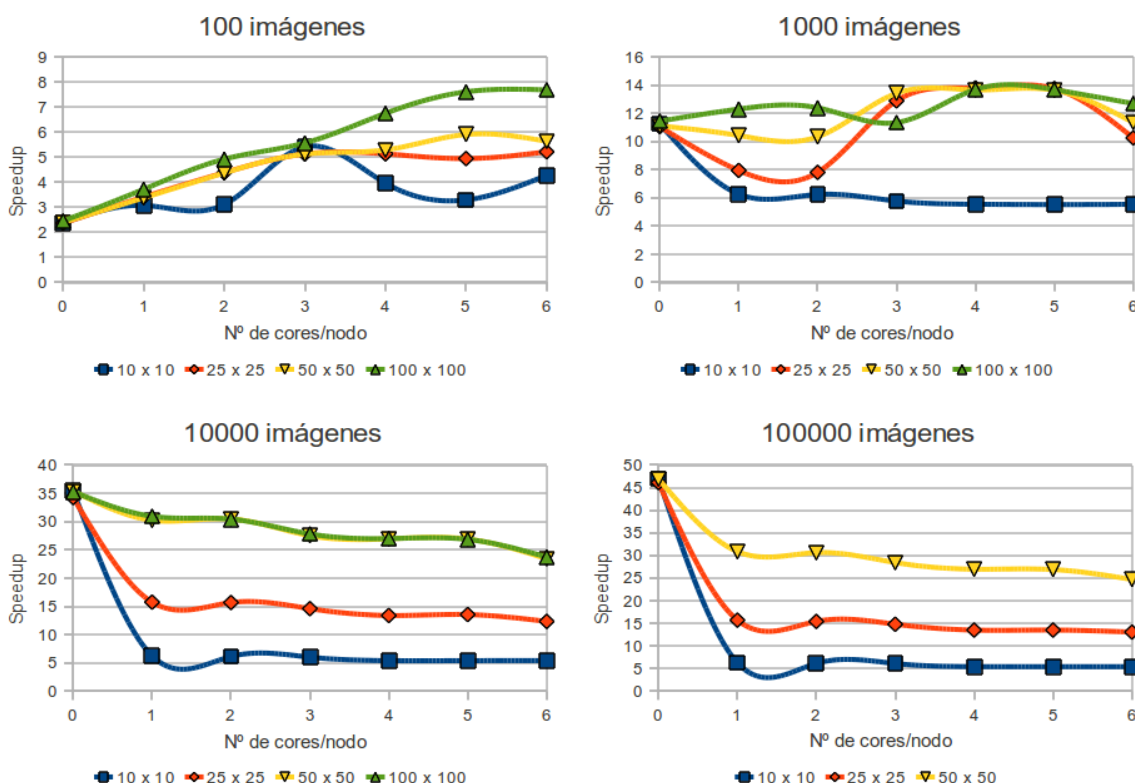

Figura 5.7: Evolución del speedup de la rutina híbrida MPI+OpenMP+CUDA utilizando un *kernel* por nodo (*marte/mercurio*)

Utilizando dos *kernels* paralelos por nodo

Como tanto *marte* como *mercurio* disponen de 2 GPUs cada uno, pasamos a repetir el mismo estudio realizado en el apartado anterior, pero invocando esta vez a dos kernels en paralelo ($g = 2$). La gráfica de la figura 5.8 muestra esta evolución, la cual es similar a la que se ha obtenido al utilizar una GPU solamente (ver figura 5.7), pero con un extra de rendimiento al emplear una GPU adicional cuando el número de imágenes a calcular es menor o igual a 1000.

De este aumento de rendimiento para pocas imágenes podemos extraer la siguiente conclusión: el cómputo realizado por el kernel realmente se reparte entre las dos GPUs disponibles, aunque solo se dedique un hilo para realizar las invocaciones al kernel. Esto justificaría por qué con un número de imágenes superior a 10000 el rendimiento obtenido es prácticamente el mismo con 1 hilo OpenMP dedicado a CUDA que con 2, pues con un valor tan alto de imágenes las 2 GPUs ya disponen de suficiente carga computacional como para mejorar aún más el rendimiento invocando a otro kernel. Sin embargo, para un número de imágenes igual o inferior a 1000, como hemos visto, la GPU parece estar algo ociosa y, por ello, sí que se mejora el rendimiento al añadir otro hilo OpenMP dedicado a CUDA, incluso puede que añadiendo alguno más siga aumentando el rendimiento, como comprobaremos a continuación.

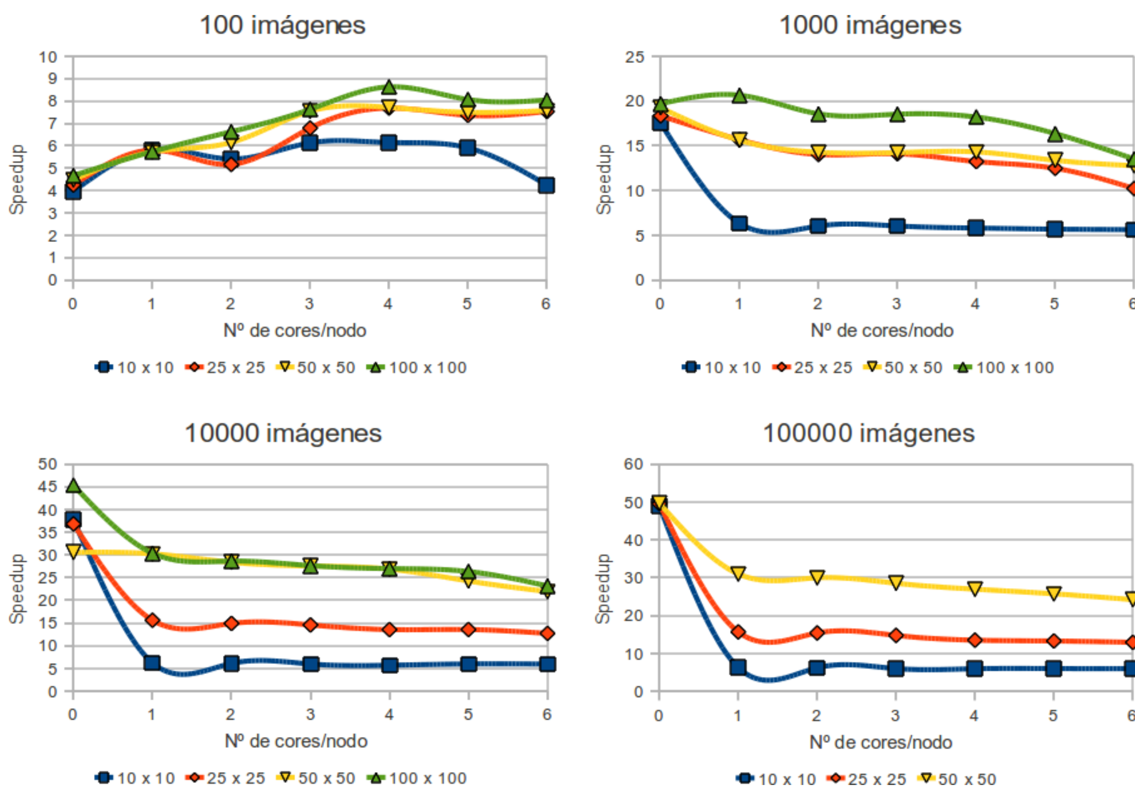

Figura 5.8: Evolución del speedup de la rutina híbrida MPI+OpenMP+CUDA utilizando tres *kernels* paralelos por nodo (*marte/mercurio*)

Utilizando tres *kernels* paralelos por nodo

En la figura 5.9 se puede ver como la predicción realizada en el punto anterior era certera, pues con $g = 3$ sigue ocurriendo algo similar a lo que ocurría con dos kernels en paralelo. Para un número bajo de imágenes aumenta el rendimiento al estar ligeramente ociosa la GPU, mientras que para valores altos añadir mayor carga computacional no supone ninguna ventaja, al estar ya ocupada la GPU.

Resumen de resultados

Como hemos podido comprobar, cuando la rutina híbrida no hace uso de ninguna GPU se puede asegurar, si obviamos las pequeñas fluctuaciones que se producen, que el mejor rendimiento se obtiene al utilizar $p = n$ y $h = c$, siendo n el número de nodos que componen el sistema y c el número de cores de cada nodo. Sin embargo, al contrario de lo que se podría pensar, al añadir g hilos OpenMP extra para que realicen invocaciones a un kernel que se ejecutará en la GPU, el valor óptimo para h no es c, sino que varía en función del tamaño del problema y de las características del sistema. La tabla 5.2 muestra las configuraciones preferidas en formato $p \times (h + g)$ en función de los números de imágenes y de puntos a calcular, donde se observa que para un número de imágenes suficientemente alto predomina la configuración $(p, h, g) = (2, 0, 2)$, aunque el valor g no afecta en gran medida al rendimiento para estos tamaños del problema al estar la GPU suficientemente ocupada, como ya hemos mencionado.

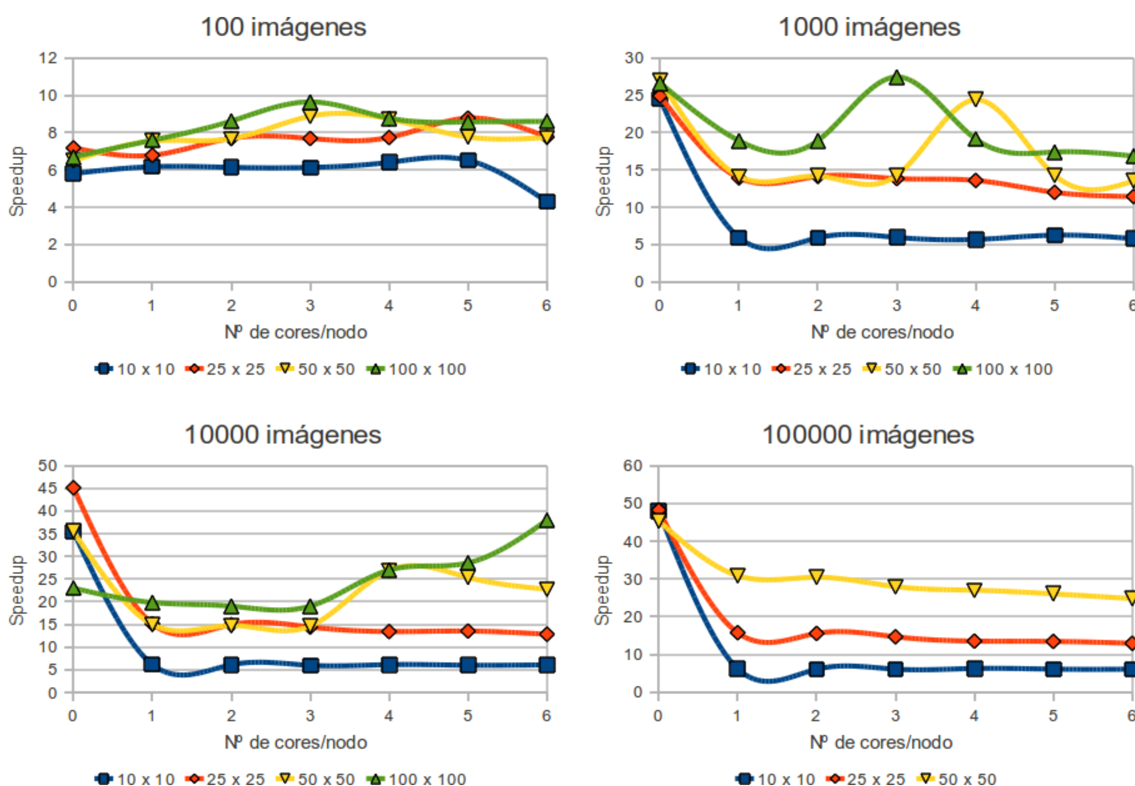

Figura 5.9: Evolución del speedup de la rutina híbrida MPI+OpenMP+CUDA utilizando tres *kernels* paralelos por nodo (*marte/mercurio*)

$x \times y \setminus nm$	100	1000	10000	100000
10×10	$2 \times (5+3)$	$2 \times (0+3)$	$2 \times (0+2)$	$2 \times (0+2)$
25×25	$2 \times (5+3)$	$2 \times (0+3)$	$2 \times (0+3)$	$2 \times (0+2)$
50×50	$2 \times (3+3)$	$2 \times (0+3)$	$2 \times (0+3)$	$2 \times (0+2)$
100×100	$2 \times (3+3)$	$2 \times (3+3)$	$2 \times (0+2)$	$2 \times (0+2)$

Tabla 5.2: Configuraciones preferidas de la rutina híbrida MPI+OpenMP+CUDA (*marte/mercurio*)

Sin embargo, para un número de imágenes menor o igual a 1000 sí que es preferible emplear 3 kernels en paralelo, ya que la GPU está parcialmente ociosa.

Debido a que los valores óptimos de los parámetros de configuración de la rutina, los cuales nos indican el número de elementos de procesamiento que se deben emplear en el cálculo paralelo, no son fijos, se resalta de nuevo la necesidad de un mecanismo de autotuning para la rutina que elija, en tiempo de ejecución, los valores de los parámetros p, h y g para una ejecución óptima en función del tamaño del problema y de las características del sistema. En el capítulo 6 profundizaremos en el diseño de dicho mecanismo de autooptimización de las rutinas.

Capítulo 6

Autooptimización

A lo largo del presente proyecto hemos podido observar el comportamiento de un conjunto de rutinas que aplican diversos modelos de paralelismo, a distintos niveles y con diferentes combinaciones de estos dando lugar a un paralelismo híbrido. Todas ellas son utilizadas para resolver el mismo problema, pero… ¿cuál de ellas es la que lo resuelve antes?

Como también hemos podido comprobar hasta ahora, la respuesta a la pregunta anterior no es única; no existe una rutina que sea más rápida que las demás, o al menos no en todos los casos. El comportamiento de cada una de las rutinas está determinado por una serie de factores:

- Los valores de los *parámetros de entrada* al problema.
- Las *características del sistema* donde se está ejecutando la rutina.
- El *número de elementos de procesamiento* implicados en el cálculo *paralelo*.

La cuestión que se plantea ahora es, en función del sistema en el que estamos trabajando y las entradas del problema, qué rutina de todas las disponibles es mejor y cuántos elementos de procesamiento deben participar en el cálculo paralelo. Para abordar este problema está el presente capítulo, en el que se diseñará un mecanismo de *autooptimización* que, tras un proceso previo de instalación de las rutinas en un sistema concreto, permita escoger la más rápida en tiempo de ejecución. Todo ello se realizará atendiendo a los factores que hemos comentado anteriormente.

La metodología de autooptimización será aplicada a las diferentes implementaciones disponibles de las rutinas que calculan las funciones de Green bidimensionales de la guía rectangular, dejando a un lado las unidimensionales debido a su bajo coste computacional. El trabajo se centrará especialmente en la rutina híbrida del capítulo 5, ya que, con diferentes valores de los parámetros de configuración p, h y g, representa a casi todas las rutinas desarrolladas (ver tabla 5.1), exceptuando las implementaciones que hacen uso de paralelismo de grano fino con OpenMP y MPI (ver secciones 3.2.2 y 3.2.4), que también serán tenidas en cuenta en el estudio.

6.1. Metodología

Una metodología general de autooptimización consta principalmente de tres fases: *diseño*, *instalación* y *ejecución*; siendo de vital importancia la fase de instalación, ya que las técnicas a aplicar en la fase previa de diseño y la fase posterior de ejecución dependerán del método de instalación elegido. Tenemos las dos siguientes opciones para realizar una instalación de la rutina:

Ejecuciones seleccionadas: Realizar una serie de ejecuciones seleccionadas, obteniendo tiempos de ejecución para diferentes configuraciones de la rutina y parámetros de entrada al problema. Para evitar una instalación exhaustiva —es decir, probar todas las configuraciones— se puede optar por dirigir las ejecuciones para no hacerlas todas. Finalmente, se obtendrá una tabla con la rutina preferida y su configuración óptima para los parámetros de entrada seleccionados.

Estimación de constantes del modelo: Partiendo de un modelo teórico del tiempo de ejecución de la rutina, obtenido en la fase de diseño de esta, se realiza una serie de ejecuciones que nos permitan estimar las constantes que aparecen en el modelo, que dependerán del problema en cuestión y del sistema concreto donde se ejecute.

La información obtenida en la fase de instalación, sea cual sea el procedimiento escogido, será utilizada en la fase de ejecución para seleccionar la rutina preferida y su configuración óptima. En el primer caso, se podrá estimar en función de la cercanía de los parámetros del problema a los de la tabla de preferencias; y en el segundo, se podrá determinar a partir del modelo, gracias a la estimación previa de las constantes.

En este capítulo vamos a ilustrar la aplicación de una metodología de autooptimización basada en la realización de varias ejecuciones para diferentes tamaños del problema y que luego elija en base a esos resultados. Para ello, utilizaremos algunos de los resultados previos del proyecto.

6.2. Instalación de la rutina

La instalación de la rutina con autooptimización consistirá, como ya hemos dicho, en encontrar la rutina y configuración óptimas para algunos tamaños de problema, a los cuales llamaremos *conjunto de instalación*. Por ejemplo, supongamos que dicho conjunto lo forman las diferentes combinaciones de $x \times y = \{10 \times 10, 50 \times 50\}$ y $nm = \{100, 10000\}$, para los cuales se obtendrían las tablas 6.1 y 6.2 que nos muestran la rutina y configuración óptimas para cada problema del conjunto de instalación junto con el speedup obtenido en *hipatia* y *marte/mercurio*, respectivamente.

$x \times y \setminus nm$	100	10000
10×10	MPI+OMP/2×8: 9,129	MPIFG/16: 15,917
50×50	MPI+OMP/4×4: 11,366	MPIFG/16: 15,058

Tabla 6.1: Rutinas y configuraciones preferidas para el conjunto de instalación y su speedup asociado (*hipatia*)

$x \times y \setminus nm$	100	10000
10×10	MPI+OMP+CUDA/$2 \times (5+3)$: 6,516	MPI+OMP+CUDA/$2 \times (0+2)$: 37,746
50×50	MPI+OMP+CUDA/$2 \times (3+3)$: 8,880	MPI+OMP+CUDA/$2 \times (0+3)$: 35,489

Tabla 6.2: Rutinas y configuraciones preferidas para el conjunto de instalación y su speedup asociado (*marte/mercurio*)

Esta información será usada posteriormente al ejecutar la rutina para decidir qué implementación y configuración se empleará para resolver el problema en función de parámetros de entrada cualesquiera.

6.3. Elección de la implementación óptima

Una vez instalada la rutina, es decir, que disponemos de cierta información sobre la ejecución de ésta en el sistema concreto, el siguiente paso será elegir, a partir de la información anterior, la implementación y configuración que se utilizarán para resolver un problema dado. Para ello, utilizaremos la función cuyo pseudocódigo se muestra en el código 6.1.

Código 6.1: Pseudocódigo de la función que elige la implementación y configuración

```
1  autotuning(x, y, n, m) {
2      speedupMax = 0;
3      para cada problema p en conjuntoInstalacion {
4          if ( p es vecino de problema(x, y, n, m) ) {
5              if (p.speedup > speedupMax) {
6                  speedupMax = p.speedup;
7                  preferido = p;
8              }
9          }
10     }
11
12     return (preferido.rutina, preferido.configuracion);
13 }
```

La función `autotuning()` simplemente buscará en el conjunto de problemas de instalación a los *vecinos* del problema dado por los parámetros de entrada `x`, `y`, `n` y `m`; quedándose como `preferido` el que mayor speedup consiguiera en la fase de instalación. Será la rutina y configuración de dicho problema las que se utilizarán para realizar el cálculo de las funciones de Green del problema actual.

Un problema del conjunto de instalación se considera vecino atendiendo a la figura 6.1, donde los círculos negros representan al conjunto de instalación en una matriz bidimensional indexada por puntos e imágenes. Cualquier otro problema (círculo blanco), es decir, con diferentes valores de puntos e imágenes a los del conjunto de instalación, tiene como vecinos a los círculos negros a los que apuntan con las flechas. Como se puede observar fácilmente, se obtienen a partir de la posición relativa que ocuparían los círculos blancos en la matriz. Por otro lado, si el problema a resolver coincide con alguno de los del conjunto de instalación, este será el único vecino y, por tanto, el que proporcione la implementación y configuración óptimas para su resolución.

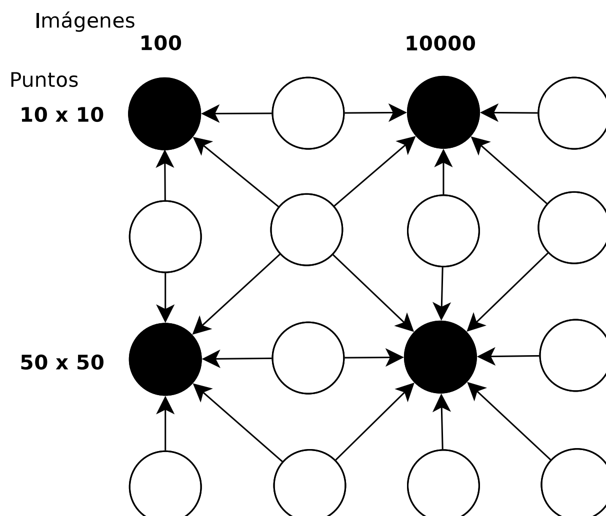

Figura 6.1: Representación gráfica de la vecindad entre problemas

Si aplicamos el algoritmo a los tamaños de problema que surgen de las diferentes combinaciones de $x \times y = \{25 \times 25, 100 \times 100\}$ y $nm = \{1000, 100000\}$, en adelante *conjunto de validación*, obtenemos las tablas 6.3 y 6.4 con las rutinas y configuraciones preferidas para *hipatia* y *marte/mercurio*, respectivamente.

$x \times y \setminus nm$	1000	100000
25×25	MPIFG/16	MPIFG/16
100×100	MPIFG/16	MPIFG/16

Tabla 6.3: Rutinas y configuraciones preferidas para el conjunto de validación aplicando autooptimización (*hipatia*)

$x \times y \setminus nm$	1000	100000
25×25	MPI+OMP+CUDA/$2 \times (0+2)$	MPI+OMP+CUDA/$2 \times (0+2)$
100×100	MPI+OMP+CUDA/$2 \times (0+3)$	MPI+OMP+CUDA/$2 \times (0+3)$

Tabla 6.4: Rutinas y configuraciones preferidas para el conjunto de validación aplicando autooptimización (*marte/mercurio*)

6.4. Contraste de resultados

Una vez que hemos visto el funcionamiento la instalación y la ejecución de la rutina con autooptimización —tomando como ejemplo los resultados obtenidos en los capítulos previos, vamos a pasar a analizar los resultados obtenidos con este mecanismo, con el objetivo de evaluar la precisión del mismo.

En las tablas 6.5 y 6.6 se muestran los tiempos de ejecución de la rutina —óptimos y con autooptimización— para el conjunto de validación en *hipatia* y *marte/mercurio*, respectivamente.

En general, podríamos decir que el mecanismo de autooptimización presentado obtiene buenos resultados, pues aunque a en algunos casos se obtengan tiempos de ejecución

$x \times y$	25×25	25×25	100×100	100×100
nm	1000	100000	1000	100000
Autooptimización	0,245	9,614	4,477	160,766
Óptimo	0,151	9,614	2,116	160,766
Diferencia absoluta	0,094	0	2,361	0
Diferencia relativa	62,25 %	0 %	111,58 %	0 %

Tabla 6.5: Comparación entre el tiempo de ejecución óptimo y el obtenido con autooptimización para el conjunto de validación (*hipatia*)

$x \times y$	25×25	25×25	100×100	100×100
nm	1000	100000	1000	100000
Autooptimización	0,155	5,012	1,706	87,814
Óptimo	0,114	5,012	1,656	79,453
Diferencia absoluta	0,041	0	0,050	8,361
Diferencia relativa	35,96 %	0 %	3,02 %	10,52 %

Tabla 6.6: Comparación entre el tiempo de ejecución óptimo y el obtenido con autooptimización para el conjunto de validación (*marte/mercurio*)

un 62,25 % o un 111,58 % mayores de lo que se obtendría haber obtenido con la rutina óptima, suele ser para problemas pequeños y en estos casos el tiempo adicional no se eleva más allá de unos segundos. Si bien esto puede suponer un problema en el caso de que queramos repetir múltiples ejecuciones para tamaños pequeños del problema, debemos tener en cuenta también que el método aplicado es muy simple si comparamos con otros que podrían ser más precisos como, por ejemplo, utilizar un conjunto de instalación mayor u otro algoritmo de selección de la rutina preferida, o uno que se base en el modelo teórico del tiempo de ejecución. Para ser utilizado con este último, en el apéndice C se expone un posible remodelado del tiempo teórico de ejecución de las rutinas[1] que tiene en cuenta factores como el reparto desigual de las computaciones y los tiempos de gestión y de comunicaciones entre los diferentes hilos y/o procesos.

[1]Primera aproximación al modelo teórico visto en las secciones 3.2 y 5.1.

Capítulo 7

Conclusiones y vías futuras

7.1. Conclusiones

A lo largo del presente proyecto se ha continuado con el estudio realizado previamente sobre un conjunto de rutinas paralelas de cálculo de funciones de Green, tanto unidimensionales como bidimensionales. Para ello, se ha contado con sistemas computacionales de diferentes características: máquinas de memoria compartida, de memoria distribuida, unidades de procesamiento gráfico y diferentes combinaciones de las anteriores.

El trabajo comienza relizando un análisis de las rutinas previas, modelando el tiempo teórico de ejecución de forma aproximada, con objeto de identificar nuevas oportunidades de paralelismo así como disponer de un punto de partida para la futura implantación de un método de autotuning a partir del modelo. Tras ello, pasamos a evaluar las rutinas en otras máquinas diferentes a donde fueron desarrolladas.

Para el problema unidimensional, los resultados obtenidos en *geatpc2* son muy similares a los obtenidos en *luna*, donde para tamaños pequeños del problema se observa preferencia por la implementación OpenMP de grano grueso y la implementación que combina OpenMP con CUDA para los más grandes. Además, debido a que CUDA emplea en los cálculos números reales de simple precisión (`float`) y el resto de implementaciones utilizan doble precisión (`double`) se plantea un pequeño estudio transversal en el que se compara el rendimiento de un tipo frente a otro en las implementaciones secuencial y OpenMP, en el cual se comprueba que la utilización de `float` no mejora más de un 60 % el rendimiento, y solamente para problemas grandes, probablemente debido a un mejor uso de la jerarquía de memoria que reduzca el número de fallos de cache.

Con respecto al caso bidimensional, en el que solo disponemos por el momento de implementaciones de grano fino, la preferencia de rutina es similar entre *geatpc2* y *luna*, en los que para problemas pequeños se prefiere OpenMP y para grandes el favorito es CUDA, aunque *luna* se empieza a decantar por CUDA con un menor número de imágenes debido a que dispone de menos cores y de una GPU más potente que *geatpc2*. En cuanto a *hipatia*, OpenMP es mejor que MPI para un número bajo de imágenes debido a que el último introduce mucha sobrecarga por mensajes frente a poca computación y, para valores altos de imágenes, se prefiere MPI al emplear nodos de mayor rendimiento que los que se pueden utilizar con el mismo número de hilos OpenMP, 16 en este caso.

También debemos añadir que, en las evaluaciones comentadas en los dos párrafos anteriores, el número óptimo de elementos de procesamiento (nodos, cores, ...) empleados para la resolución no es siempre el mismo, de ahí que se haya diseñado un mecanismo de autooptimización, el cual comentaremos más adelante.

Finalizado el estudio anterior y tras una primera experiencia con la combinación de dos modelos de programación paralela, nos planteamos ir más allá e implementar una rutina para resolver el problema de mayor coste (el bidimensional) que fuera híbrida para aprovechar de forma simultánea las ventajas de los tres tipos de entornos computacionales vistos hasta el momento al combinar MPI, OpenMP y CUDA. La idea principal de esta implementación es aplicar paralelismo a nivel de varias funciones de Green con MPI y OpenMP; y utilizar CUDA para acelerar el cálculo de una única función de Green añadiendo uno o varios hilos OpenMP extra para ello.

Hasta ahora, todos los sistemas empleados han sido *Intel* y el compilador utilizado `icpc`, pero para evaluar esta implementación híbrida se han usado *marte* y *mercurio*, ambos sistemas con procesadores AMD, por lo que surge una nueva cuestión antes de pasar a evaluar la rutina desarrollada: utilizar `g++`, `icpc` u otro compilador específico como `openCC`. Tras un pequeño estudio en el que se obtienen speedups de 2,5 a 5 al utilizar `icpc` en puesto de `g++` se descarta en principio utilizar otro compilador que no sea `icpc` al ser una mejora más que aceptable.

La evaluación de la rutina híbrida comienza con el primer nivel de hibridación resultante de combinar únicamente MPI y OpenMP, para el cual se concluye que, en términos generales, manteniendo constante el producto ph es preferible utilizar una relación baja de los parámetros $\frac{p}{h}$, lo que normalmente significa $p = n$ y $h = c$, siendo n el número de nodos del sistema y c el número de cores de cada nodo.

Al añadir la intervención de la GPU en la rutina, los resultados anteriores difieren significativamente, pues para valores altos de imágenes la mejor opción de la rutina es utilizar únicamente 2 o 3 hilos dedicados al cálculo con CUDA y no utilizar el resto de los cores del nodo. Esto se debe a que en el par *marte/mercurio* existe algún cuello de botella que al utilizar varios cores penaliza el rendimiento. Sin embargo, para pocas imágenes sí merece la pena apoyarse en algunos de los cores restantes ya que CUDA no puede extraer suficiente paralelismo. Como vemos, los resultados son variables, justificando así de nuevo la implantación de un mecanismo de autooptimización.

Por último, se ha ilustrado la aplicación de un sencillo mecanismo de autooptimización basado en información empírica almacenada tras un proceso de instalación previo de las rutinas en el sistema donde se van a utilizar, obteniendo resultados satisfactorios teniendo en cuenta la simpleza del método aplicado. Se han obtenido predicciones exactas de la rutina óptima y algunos tiempos de no más de un 36 % mayores al óptimo en *marte/mercurio* y pese a que se han obtenido tiempos hasta un 112 % más lentos en *hipatia*, no son más que unos segundos de diferencia con la rutina óptima.

En resumen, hemos evaluado las rutinas de partida en diferentes sistemas con resultados similares a los obtenidos anteriormente y, por tanto, satisfactorios; se ha desarrollado una rutina que hace uso de varios modelos de programación paralela de manera conjunta con éxito, aprovechando las características heterogéneas y jerárquicas de los sistemas computacionales actuales; y hemos finalizado con la aplicación de un método simple de

autotuning para elegir en tiempo de ejecución una rutina (y su configuración de ejecución) para resolver un problema dado que no se aleja demasiado de la óptima.

7.2. Vías futuras

A lo largo de este proyecto hemos cubierto parcialmente algunas de las líneas que quedaron abiertas en [21]:

- Evaluar el comportamiento de las rutinas paralelas en sistemas computacionales de diferentes características, como lo son *geatpc2*, *hipatia* y *marte/mercurio*.
- Diseñar y evaluar rutinas que saquen partido de las características híbridas presentes en los sistemas actuales.
- Utilizar un mecanismo de autooptimización que, en función de los parámetros de entrada al problema y del sistema donde se resuelva, escoja la rutina que lo resuelva más rápido.

Como bien se ha dicho, han sido cubiertas solamente algunas, y de forma parcial, por lo que se proponen las siguientes vías futuras como continuación natural del trabajo realizado en este proyecto:

- Ampliar el mismo estudio realizado a las funciones de Green tridimensionales.
- Preparar y evaluar las rutinas con diferentes configuraciones de ejecución sobre sistemas heterogéneos en el que existan nodos compuestos por CPUs y GPUs de diferentes características entre sí.
- Utilización de las rutinas desarrolladas en programas de electromagnetismo donde utilicen funciones de Green [22, 24].
- Profundizar en el desarrollo de otros mecanismos de autooptimización más precisos, tanto basados en información empírica de instalación como con modelado de rutinas y estimación de constantes del sistema. Este último incluiría un refinamiento más exhaustivo del modelo teórico, con utilización de modelos como los que se muestran en el apéndice C.
- Experimentar las posibilidades que nos ofrecen otros estándares de computación paralela, como el estándar abierto *OpenCL* [3], con el que es posible codificar tanto aplicaciones para multicores como para dispositivos de aceleración gráfica a través del mismo framework de programación.
- Aprovechar otros niveles de paralelismo que las máquinas actuales nos ofrecen, como pueden ser las extensiones multimedia de los procesadores Intel *SSE* (*Streaming SIMD Extensions*), y combinar dicho paralelismo con las implementaciones disponibles.

Apéndice A

Instalación del compilador de Intel

En este apéndice se proporciona un script junto con las instrucciones para instalar el compilador de C/C++ de Intel, en su versión de 64 bits. El script que realiza la descarga, la extracción y ejecuta el instalador podemos verlo en el código A.1.

Código A.1: Script de instalación del compilador de Intel

```
1 wget http://registrationcenter-download.intel.com/akdlm/irc_nas/2475/
      l_ccompxe_intel64_2011.9.293.tgz
2 tar xzfv l_ccompxe_intel64_2011.9.293.tgz
3 sudo l_ccompxe_intel64_2011.9.293/install.sh
```

Seguimos cada uno de los pasos dejando las opciones por defecto y, cuando llegue el momento de introducir la clave de activación, introduciremos la siguiente, obtenida tras registrarnos en la web de Intel: `N5D5-PFF8X3JD`. Con estas sencillas instrucciones, finaliza la instalación del compilador.

Antes de empezar a trabajar, se necesita un paso previo de post-instalación, que consiste en añadir la siguiente línea al fichero `~/.bashrc`:

```
source /opt/intel/bin/compilervars.sh intel64
```

Hecho esto, ya podemos utilizar el compilador de Intel para nuestros programas escritos en C/C++.

Apéndice B

Instalación del entorno CUDA

El objetivo de este apéndice es servir como una guía rápida para la instalación del entorno de programación CUDA en un sistema Linux. La instalación del entorno se resume con la instalación de los siguientes items: el *Developer Driver*, el *CUDA Toolkit* y el *GPU Computing SDK*. Concretemante, se instalará la versión 4.1 del entorno.

B.1. Developer Driver

En primer lugar, instalaremos el driver de CUDA para la GPU NVIDIA, el cual nos permitirá comunicarnos con la GPU. Nos dirigimos a la web de descargas del toolkit [2], donde seleccionamos, en la sección *Developer Drivers Downloads*, la versión adecuada del driver y la descargamos.

Una vez descargado el fichero, lo ejecutamos como usuario *root* desde la línea de comandos. En nuestro caso, como utilizamos *OpenSUSE 11.0* de 64 bits, sería el fichero:

```
./NVIDIA-Linux-x86_64-285.05.33.run
```

Seguimos la secuencia de instalación por defecto y el driver quedará instalado.

B.2. CUDA Toolkit

En primer lugar, se debe realizar la instalación del toolkit de CUDA. Este contiene todas las herramientas necesarias para empezar a programar con CUDA; además de herramientas de depuración, librerías y documentación.

Para instalarlo, debemos dirigirnos previamente a la web anterior, elegir la versión adecuada de la sección *CUDA Toolkit Downloads* y descargarla.

Una vez descargado el archivo, desde línea de comandos, y como usuario *root* ejecutamos el fichero descargado. En nuestro caso, ejecutaríamos la orden:

```
./cudatoolkit_4.1.28_linux_64_suse11.2.run
```

Seguimos la instalación dejando todas las opciones por defecto y la instalación del toolkit se dará por finalizada.

B.3. GPU Computing SDK

Tras instalar el toolkit, procedemos a instalar el SDK, el cual contiene una serie de ejemplos útiles para comenzar a utilizar CUDA, junto con unas librerías útiles en adición a las básicas proporcionadas en el toolkit.

El procedimiento seguido para instalarlo es similar al visto anteriormente, descargamos el SDK de la web anterior, dirigiéndonos a la sección *GPU Computing SDK Downloads* y seleccionando la versión para Linux.

Una vez descargado el fichero, se procederá a la instalación del SDK en el directorio *home* del usuario actual. Para ello, se introducirá por línea de comandos la siguiente orden para ejecutar el fichero descargado:

```
./gpucomputingsdk_4.1.28_linux.run
```

Al igual que antes, dejamos marcadas las opciones por defecto en la instalación cuando se nos pregunte y el SDK quedará instalado.

B.4. Post-instalación

B.4.1. Variables de entorno

Una vez finalizada la instalación, es necesario actualizar dos variables de entorno siempre que queramos utilizar CUDA. Para ello, modificaremos el fichero `.bashrc` del directorio `home` de modo que se automatice este proceso. Concretamente, se añadirán las líneas:

```
export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:/usr/local/cuda/lib:~/
    NVIDIA_GPU_Computing_SDK/shared/lib:$LD_LIBRARY_PATH
```

Cabe destacar que el directorio `/usr/local/cuda/lib64` solamente será necesario incluirlo en sistemas de 64 bits.

B.4.2. Librerías del SDK

Además de lo anterior, tras la instalación del SDK, debemos dirigirnos al directorio donde se ha instalado este. Si se ha dejado la opción por defecto, será `~/NVIDIA_GPU_Computing_SDK`. Desde allí, ejecutamos la orden `make` para compilar las librerías del SDK.

Cuando termine la ejecución, en el subdirectorio `C/src` econtramos el código fuente de los ejemplos en C del SDK, los cuales podemos compilar y ejecutar como primera toma de contacto con el entorno. Además de ejemplos en C/C++, también se encuentran ejemplos para *OpenCL* y *DirectCompute*.

Hecho esto, se da por finalizada la instalación del entorno de programación CUDA para GPGPU.

Apéndice C

Remodelado de las funciones de Green bidimensionales

Antes de pasar a aplicar una metodología de autooptimización basada en el modelo teórico del tiempo de ejecución, revisaremos de nuevo dicho modelo para añadir factores que no se tenían en cuenta, tales como la gestión de procesos e hilos o la sobrecarga introducida por CUDA con cada invocación al kernel, además de finalizar con el modelado de las operaciones de reducción (R_{OMP} y R_{MPI}) y `MPI_Gather()` (G_{MPI}). El objetivo de este *remodelado* es que podamos obtener un ajuste más fino al aplicar la autooptimización a la rutina.

El modelo teórico de la rutina secuencial de la ecuación 3.6 se considera definitivo, por lo que pasamos directamente a las rutinas paralelas.

C.1. Paralelismo de grano fino con OpenMP

Comenzaremos con la rutina que emplea paralelismo de grano fino con OpenMP. Para ello, tenemos que añadir a la fórmula 3.7 los modelos de tiempo de gestión de los hilos OpenMP ($T_G(h)$) y de la operación de reducción ($R_{OMP}(h)$) desarrollados, dando como resultado la ecuación siguiente:

$$\begin{aligned} t(x,y,nmod,n,m,h) &= xy\left(\left\lceil\frac{nmod}{h}\right\rceil S_1 + \left\lceil\frac{(2m+1)}{h}\right\rceil (2n+1)S_2 + R_{OMP}(h) + T_G(h)\right) \\ &= xy\left(\left\lceil\frac{nmod}{h}\right\rceil S_1 + \left\lceil\frac{(2m+1)}{h}\right\rceil (2n+1)S_2 + \log_2 hR_1\right. \\ &\quad \left. + (h-1)R_2 + hT_G\right) \end{aligned} \tag{C.1}$$

donde modelamos $R_{OMP}(h)$ en función de dos posibles formas de realizar la reducción —paralelo o secuencial—, ya que desconocemos como realiza OpenMP internamente la reducción. La estimación de constantes decidirá el peso de cada una de ellas. En cuanto al tiempo de gestión de hilos, supondremos que depende linealmente de h. Tras finalizar el desarrollo de cada término, nos quedan las constantes siguientes a estimar: S_1, S_2, R_1, R_2 y T_G.

C.2. Paralelismo de grano fino con MPI

Siguiendo la misma idea que en el apartado anterior, modelaremos a continuación los tiempos de gestión de procesos en MPI $T_G(p)$ y de la operación de reducción $R_{MPI}(p)$ en la rutina que aplica paralelismo MPI de grano fino, obteniendo la siguiente ecuación:

$$\begin{aligned} t(x,y,nmod,n,m,p) &= xy\left(nmod \cdot S_1 + \left\lceil \frac{(2m+1)}{p} \right\rceil (2n+1)S_2 + R_{MPI}(p)\right) + T_G(p) \\ &= xy\left(nmod \cdot S_1 + \left\lceil \frac{(2m+1)}{p} \right\rceil (2n+1)S_2 + T_I + (p-1)\right. \\ &\quad \left. \cdot (T_E + T_R)\right) + pT_G \\ &= xy\left(nmod \cdot S_1 + \left\lceil \frac{(2m+1)}{p} \right\rceil (2n+1)S_2 + T_I + (p-1) \cdot T_{E+R}\right) \\ &\quad + pT_G \end{aligned} \tag{C.2}$$

Al trabajar con MPI, en la operación $R_{MPI}(p)$ intervendrán comunicaciones, las cuales dividimos en dos fases: *inicio* de las comunicaciones (T_I), que consideraremos que se realiza en paralelo en todos los procesos; y el tiempo de *envío* de la parte de la función de Green calculada por cada proceso (T_E) hacia el encargado la reducción de los datos, que tardará T_R segundos en acumular cada parte recibida. Finalmente, tras unir en la misma constante los tiempos T_E y T_R al estar multiplicados por el mismo término, nos quedan las constantes siguientes a estimar: S_1, S_2, T_I, T_{E+R} y T_G.

C.3. Paralelismo de grano fino con CUDA

Antes de refinar el modelo teórico de la rutina híbrida debemos hacer lo propio con la implementación CUDA, para poder desarrollar el speedup que obtenemos con una GPU. A la ecuación 3.8 hemos de añadir el tiempo de creación/destrucción del kernel (T_K) junto con el referente a movimientos de datos entre CPU y GPU ($T_M(m,n)$), como se puede ver en la siguiente fórmula:

$$\begin{aligned} t(x,y,nmod,n,m) &= xy\left(nmod \cdot S_1 + (2m+1)(2n+1)\left(S_{2(GPU)} + R\right) + T_M(m,n) + T_K\right) \\ &= xy\left(nmod \cdot S_1 + (2m+1)(2n+1)\left(S_{2(GPU)} + R\right) + \sum_{-m}^{m}\sum_{-n}^{n} T_M + T_K\right) \\ &= xy\left(nmod \cdot S_1 + (2m+1)(2n+1)\left(S_{2(GPU)} + R + T_M\right) + T_K\right) \\ &= xy\left(nmod \cdot S_1 + (2m+1)(2n+1)T_{M+S_{2(GPU)}+R} + T_K\right) \end{aligned} \tag{C.3}$$

que tras simplificar y unificar constantes nos quedan las siguientes a estimar: S_1, $T_{M+S_{2(GPU)}+R}$ y T_K.

Para finalizar, simplificamos el speedup que se alcanza utilizando una GPU respecto a la CPU para un tamaño de problema dado:

$$\begin{aligned} S_{GPU/CPU}(x,y,nmod,n,m) &= \frac{xy\left(nmod \cdot S_1 + (2m+1)(2n+1)S_2\right)}{xy\left(nmod \cdot S_1 + (2m+1)(2n+1)T_{M+S_{2(GPU)}+R} + T_K\right)} \\ &= \frac{nmod \cdot S_1 + (2m+1)(2n+1)S_2}{nmod \cdot S_1 + (2m+1)(2n+1)T_{M+S_{2(GPU)}+R} + T_K} \end{aligned} \tag{C.4}$$

C.4. Paralelismo híbrido combinando MPI, OpenMP y CUDA

Al igual que con las anteriores rutinas, añadiremos al modelo teórico del tiempo de ejecución de la implementación híbrida los tiempos correspondientes a gestión de procesos/hilos $T_G(p,h,g)$ y al paso de mensajes que realiza la operación $G_{MPI}(x,y,p)$. La siguiente fórmula muestra el desarrollo de estos nuevos factores:

$$\begin{aligned} t(x,y,nmod,n,m,p,h,g) &= \left\lceil \frac{\left\lceil \frac{x}{p} \right\rceil}{h + gS_{GPU/CPU}(x,y,nmod,n,m)} \right\rceil y(nmod \cdot S_1 + (2m+1) \\ &\cdot (2n+1)S_2) + G_{MPI}(x,y,p) + T_G(p,h,g) \\ &= \left\lceil \frac{\left\lceil \frac{x}{p} \right\rceil}{h + gS_{GPU/CPU}(x,y,nmod,n,m)} \right\rceil y(nmod \cdot S_1 + (2m+1) \\ &\cdot (2n+1)S_2) + T_I + (p-1) \left\lceil \frac{x}{p} \right\rceil yT_E + pT_{G(MPI)} \\ &+ (h+g)T_{G(OMP)} \end{aligned} \tag{C.5}$$

El tiempo de comunicaciones incluirá un tiempo de inicio de comunicaciones (T_I) y el tiempo necesario para enviar una función de Green al maestro (T_E). El total de funciones de Green a enviar desde cada proceso serán $\left\lceil \frac{x}{p} \right\rceil y$. El tiempo de gestión se descompone en procesos MPI ($T_{G(MPI)}$) e hilos OpenMP ($T_{G(OMP)}$). Finalmente, tendremos que estimar las constantes siguientes: S_1, S_2, T_I, T_E, $T_{G(MPI)}$ y $T_{G(OMP)}$.

Bibliografía

[1] «ComplexHPC.org». Último acceso: Junio 2012.
`http://complexhpc.org/`

[2] «CUDA Toolkit 4.1». Último acceso: Febrero 2012.
`http://developer.nvidia.com/cuda-toolkit-41`

[3] «OpenCL - The open standard for parallel programming of heterogeneous systems». Último acceso: Septiembre 2012.
`http://www.khronos.org/opencl/`

[4] «OpenMP.org » About the OpenMP ARB and OpenMP.org». Último acceso: Mayo 2012.
`http://openmp.org/wp/about-openmp/`

[5] «Portable Hardware Locality (hwloc)». Último acceso: Mayo 2012.
`http://www.open-mpi.org/projects/hwloc/`

[6] «SEDIC: Recursos: Servicios de Cálculo Científico». Último acceso: Mayo 2012.
`http://www.upct.es/sait/sedic/servcalc.html`

[7] «x86 Open64 Compiler Suite | AMD Developer Central». Último acceso: Junio 2012.
`http://developer.amd.com/tools/open64/Pages/default.aspx`

[8] *NVIDIA CUDA C Programming Guide, Version 4.2*. NVIDIA, 2012.
`http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`

[9] CÁMARA, J.; CUENCA, J.; GIMÉNEZ, D. and VIDAL, A. M.: «Empirical Autotuning of Two-Level Parallel Linear Algebra Routines on Large cc-NUMA Systems». In: *The 10th IEEE International Symposium on Parallel and Distributed Processing with Applications*, Leganés, Madrid, 2012.

[10] CAPOLINO, F.; WILTON, D. R. and JOHNSON, W. A.: «Efficient Computation of the 2-D Green's Function for 1-D Periodic Structures Using the Ewald Method». *IEEE Transactions on Antennas and Propagation*, 2005, **53(9)**, pp. 2977–2984.

[11] CUENCA, J.; GIMÉNEZ, D. and GARCÍA, L. P.: «Improving linear algebra computation on NUMA platforms through auto-tuned nested parallelism». In: *20th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, PDP 2012*, Garching, Germany, 2012.

[12] CUTILLAS-LOZANO, L-G.; CUTILLAS-LOZANO, J-M. and GIMÉNEZ, D.: «Modeling shared-memory metaheuristic schemes for electricity consumption». In: *9th International Symposium on Distributed Computing and Artificial Intelligence*, University of Salamanca (Spain), 2012.

[13] FRANCO, J.; BERNABÉ, G.; FERNÁNDEZ, J. and UJALDÓN, M.: «Parallel 3D fast wavelet transform on manycore GPUs and multicore CPUs». In: *International Conference on Computational Science*, Amsterdam, pp. 1101–1110, 2010.

[14] FRIGO, M. and JOHNSON, S. G.: «FFTW: An Adaptive Software Architecture for the FFT». In: *ICASSP conference*, Seattle, Washington, USA, pp. 1381–1384, 1998.

[15] GARLAND, M.; LE GRAND, S.; NICKOLLS, J.; ANDERSON, J.; HARDWICK, J.; MORTON, S.; PHILLIPS, E.; ZHANG, Y. and VOLKOV, V.: «Parallel Computing Experiences with CUDA». *IEEE Micro*, 2008, **28(4)**, pp. 13–27.

[16] HIMAWAN, B. and VACHHARAJANI, M.: «Deconstructing Hardware Usage for General Purpose Computation on GPUs». In: *5th Annual Workshop on Duplicating Deconstructing and Debunking*, Boston, MA, USA, 2006.

[17] KIRK, D. B. and HWU, W-M. W.: «B.2 Memory Coalescing Variations». In: *Programming Massively Parallel Processors: A Hands-on Approach*, pp. 246–250. Elsevier Inc., 2010.

[18] LINDHOLM, E.; NICKOLLS, J.; OBERMAN, S. and MONTRYN, J.: «NVIDIA Tesla: A Unified Graphics and Computing Architecture». *IEEE Micro*, 2008, **28(2)**, pp. 39–55.

[19] PARK, M-J. and NAM, S.: «Rapid Summation of the Green's Function for the Rectangular Waveguide». *IEEE Transactions on Microwave Theory and Techniques*, 1998, **46(12)**, pp. 2164–2166.

[20] PARK, M-J.; PARK, J. and NAM, S.: «Efficient Calculation of the Green's Function for the Rectangular Cavity». *IEEE Microwave and Guided Wave Letters*, 1998, **8(3)**, pp. 124–126.

[21] PÉREZ, C.: *Análisis, desarrollo y optimización de rutinas paralelas de cálculo de funciones de Green*. Proyecto fin de carrera, Facultad de Informática (Universidad de Murcia), 2011. `http://www.um.es/pcgum/PFCs_y_TMs/2011memoriaPFC_Green.pdf`

[22] PÉREZ, F. J.: *Investigación En Técnicas Numéricas Aplicadas Al Estudio De Nuevos Dispositivos Para Sistemas De Comunicaciones*. Ph.D. thesis, Escuela Técnica Superior de Ingeniería de Telecomunicación (Universidad Politécnica de Cartagena), 2009.

[23] PÉREZ-ALCARAZ, C.; GIMÉNEZ, D.; ÁLVAREZ-MELCÓN, A. and QUESADA, F. D.: «Parallelizing the computation of Green functions for computational electromagnetism problems». In: *The 13th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2012)*, Shanghai, China, 2012.

[24] PICÓN, J. G.: *Optimización y Paralelización del Software para el Cálculo de Campos Electromagnéticos MEATSS*. Proyecto fin de carrera, Escuela Técnica Superior de Ingenieros de Telecomunicación (Universidad Politécnica de Valencia), 2012.

[25] RAMIRO, C.; LÓPEZ-ESPÍN, J. J.; GIMÉNEZ, D. and VIDAL, A. M.: «Parallel Two-Stage Least Squares algorithms for Simultaneous Equations Models on GPU». *Technical Report*, Polytechnic University of Valencia, 2012.

[26] ——: «Two-Stage Least Squares algorithms with QR decomposition for Simultaneous Equations Models on heterogeneous multicore and multi-GPU systems». In: *International Conference on Computational Science*, Omaha, Nebraska, pp. 1–4, 2012.

[27] WHALEY, R. C.; PETITET, A. and DONGARRA, J.: «Automated empirical optimizations of software and the ATLAS project». *Parallel Computing*, 2001, **27(1-2)**, pp. 3–35.