Konputagailuen Arkitektura eta Teknologia Saila
Departamento de Arquitectura y Tecnología de Computadores

Universidad del País Vasco
Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Resolución eficiente de la ecuación de Poisson en un clúster de GPU

Memoria que para optar al grado de Doctor en Informática presenta:
Jose Luis Jodra Luque

Dirigida por:
Dr. Ibai Gurrutxaga Goikoetxea
Dr. Javier Muguerza Rivero

Donostia, 2016

Konputagailuen Arkitektura eta Teknologia Saila
Departamento de Arquitectura y Tecnología de Computadores

Universidad del País Vasco
Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Resolución eficiente de la ecuación de Poisson en un clúster de GPU

Memoria que para optar al grado de Doctor en Informática presenta:

Jose Luis Jodra Luque

Dirigida por:

Dr. Ibai Gurrutxaga Goikoetxea
Dr. Javier Muguerza Rivero

Donostia, 2016

Este trabajo ha sido parcialmente subvencionado por la Universidad del País Vasco UPV/EHU (UFI11/45); por el Departamento de Educación, Universidades e Investigación (IT-395-10); y por el Ministerio de Economía y Competitividad del Gobierno de España cofinanciado por el Fondo Europeo de Desarrollo Regional – FEDER (TIN2014-52665-C2-1-R).

A Cris, Izaro, Ibon,
Aita y Ama

Agradecimientos

Debo de agradecer de manera especial y sincera a Ibai y a Javi por haberme permitido realizar esta tesis doctoral bajo su dirección. Muchas gracias por vuestra actitud, siempre dispuestos a ayudar y a trabajar. Gracias por tirarme de las orejas cuando era necesario y por animarme en los momentos en los que quería tirar la toalla. En definitiva, muchas gracias por todo. Tengo una deuda con vosotros que no sé si algún día podré saldar.

Quiero agradecer también a todos los miembros del grupo de investigación ALDAPA, que me han adoptado *estando ya un poco crecidito*. Desde el principio me habéis tratado como si fuera uno más.

No puedo olvidarme de mi anterior grupo de investigación, NQaS. Desde la distancia guardo un recuerdo muy bonito de todos vosotros, de los que siguen y de los que ya no siguen, de los que están y de los que ya no están. A todos vosotros gracias, siempre seréis mis *patitos*.

En cuanto a mis padres, no hay espacio suficiente para expresar mi agradecimiento. Gracias por ser unos padres excelentes. Aita, te echo de menos.

Gracias Cris por ser como eres, por poseer todas las cualidades que yo buscaba en una pareja y por hacerme feliz. Gracias por quererme tal y como soy –con mis pocas virtudes y muchos defectos– pero sobre todo, gracias por darme dos hijos maravillosos; espero ser digno de tan preciado tesoro.

Gracias Izaro e Ibon, con vosotros mi vida ha alcanzado una nueva dimensión en la que cada día me sorprendéis de una manera inimaginable. Espero poder devolveros todo el tiempo que esta tesis *nos ha robado*. Os quiero mucho y me hacéis muy feliz, TQMIUM.

Resumen

Resumen

Este trabajo de investigación se enmarca en el contexto de la computación de alto rendimiento (High Performance Computing, HPC) y, más en concreto, en la computación paralela utilizando computación de propósito general en GPU (General-Purpose computing on GPU, GPGPU). Aunque el trabajo surge en el contexto de la física computacional, las aportaciones de esta tesis son aplicables a múltiples ámbitos de la vida real: electrostática, ingeniería mecánica, etc.

El objetivo principal de este trabajo se ha centrado en ***la resolución eficiente de la ecuación de Poisson en un clúster de unidades de procesamiento gráfico (Graphics Processing Units, GPU)*** y se han aplicado diversas técnicas con el fin de optimizar los programas implementados: técnicas de segmentación software, optimización del acceso a memoria, sincronización, estimación de parámetros óptimos de las librerías de cómputo, etc.

La ecuación de Poisson está presente en multitud de cálculos científicos y su resolución suele ser una de las fases que más coste computacional conlleva, por lo que su optimización es crítica para cualquier implementación que haga uso de ella. Existe una gran cantidad de métodos teóricos que permiten una solución eficiente, siendo uno de ellos la transformada rápida de Fourier (Fast Fourier Transform, FFT), ampliamente usada en la solución de diversos problemas de cómputo. En este trabajo de investigación se ha implementado un algoritmo que resuelve eficientemente la ecuación de Poisson mediante FFT en un clúster de GPU.

Para llevar a cabo esta implementación, se ha hecho uso de una librería específica que ofrece NVIDIA especializada en la resolución de la FFT en GPU, conocida como transformada rápida de Fourier de CUDA (CUDA Fast Fourier Transform, cuFFT). El uso de esta librería ha implicado la realización de un análisis de los diferentes parámetros y modos de funcionamiento disponibles, de cara a identificar cuál es la configuración más adecuada para conseguir cálculos de FFT óptimos. En este análisis se ha estudiado el comportamiento de la librería en cuanto a rendimiento y uso de memoria para el cálculo de la FFT en diferentes escenarios y con diversas configuraciones.

La resolución de la ecuación de Poisson en un clúster de GPU implica la comunicación de datos entre diferentes GPU de cara a obtener el resultado final. En esta comunicación los datos pueden estar distribuidos en memoria de diferentes maneras, y éstas no siempre tienen porqué ser la más adecuadas para conseguir la mayor eficiencia en la fase de comunicación. Para realizar la comunicación de manera eficiente es importante disponer de un mecanismo que permita modificar la distribución de los datos en la memoria de la GPU. Por este motivo, otra de las aportaciones del presente trabajo de investigación ha consistido en definir una librería para realizar transposiciones 3D en GPU de manera eficiente y que será utilizada para optimizar la comunicación interproceso (MPI).

Los resultados obtenidos con la implementación realizada en este trabajo para resolver la ecuación de Poisson, muestran una buena escalabilidad, por lo menos, hasta 16 GPU –máximo número de GPU disponibles en la arquitectura utilizada. Aunque el elemento que más penaliza el tiempo de ejecución corresponde a las comunicaciones por red del clúster, la implementación en GPU es hasta 3,4 veces más rápida que la implementación en un clúster de CPU.

Palabras clave: HPC, MPI, GPGPU, FFT, ecuación de Poisson, CUDA, transposiciones.

Índice general

Índice de figuras

Índice de tablas

Lista de algoritmos

Parte I

INTRODUCCIÓN

Capítulo 1

Introducción

Índice

Las unidades de procesamiento gráfico (Graphics Processing Units, GPU) son aceleradores gráficos que en su origen fueron creados para acelerar los procesos gráficos. Sin embargo, ésta no es su única función ya que en los últimos años están siendo extensamente utilizadas en sistemas de cómputo paralelo con el objetivo de ejecutar programas de propósito general, frecuentemente de carácter científico.

Esta práctica se conoce como computación de propósito general en GPU (General-Purpose computing on GPU, GPGPU) y nadie puede poner en duda que esta nueva tendencia está suponiendo una revolución en el cálculo científico paralelo. Basta sólo con ver la última clasificación de computadores de top500 publicada, donde a noviembre de 2015, 73 de esas máquinas están provistas de aceleradores gráficos en su interior [1].

Como se ha mencionado, una de las áreas donde más intensamente se está aprovechando las ventajas que ofrecen este tipo de aceleradores es la computación científica. Si nos fijamos en publicaciones de los últimos años, se puede ver que una gran cantidad de trabajos han sido portados de manera exitosa desde código para unidad de procesamiento central (Central Processing

Unit, CPU) a GPU, obteniendo importantes mejoras en el rendimiento [2–4].

1.1. Motivación

Dentro de la computación científica, la solución de la ecuación de Poisson es frecuentemente utilizada para resolver problemas de muy diversa índole. Por ejemplo, su resolución es necesaria en procesos relativos a la electrostática, ingeniería mecánica y física teórica donde, en la mayoría de los casos, es el proceso que más demanda de cómputo requiere [5].

La ecuación de Poisson es una ecuación diferencial parcial elíptica y una manera de resolver computacionalmente dicha ecuación es mediante la transformada discreta de Fourier (Discrete Fourier Transform, DFT) que puede ser calculada eficientemente mediante la transformada rápida de Fourier (Fast Fourier Transform, FFT). Existe una vasta literatura científica sobre FFT y su eficiente resolución en CPU [6–9], y, más recientemente, se están presentando diversas implementaciones de su resolución sobre una y múltiples GPU [10–29]. Sin embargo, todos estos trabajos únicamente se han centrado en calcular eficientemente la FFT, y no se han aplicado directamente en la resolución de la ecuación de Poisson.

En la bibliografía se encuentran múltiples propuestas para resolver la ecuación de Poisson, tanto en CPU [5] como en GPU. Algunas de las soluciones para resolver la ecuación en GPU se limitan a un único nodo [30–35]. Y otros trabajos resuelven la ecuación de Poisson en múltiples nodos con múltiples GPU, aunque en vez de resolver la ecuación mediante FFT utilizan otro método (Jacobi) con el que no se consigue una solución eficiente [36, 37].

Si nos centramos en los trabajos para resolver la ecuación de Poisson basadas en FFT en GPU, tenemos a autores de referencia como Wu y Jaja, quienes han publicado una serie de trabajos en este área [16, 38–40]. Sin embargo, ninguna de estas soluciones es aplicable a un clúster de GPU.

En cuanto a trabajos específicos sobre el cálculo de la FFT en un clúster de GPU, la mayoría se centran en optimizar las primitivas de la comunicación inter-nodo en lugar de utilizar el estándar de interfaz de paso de mensajes (Message Passing Interface, MPI) [10–15]. Estas primitivas permiten optimizaciones a muy bajo nivel, lo que permite reducir la latencia de comunicación a

costa de perder la portabilidad del código obtenido. Los resultados obtenidos en estos trabajos son bastante prometedores, y esto ha motivado la creación de una implementación que sea capaz de calcular eficientemente la ecuación de Poisson mediante el cálculo de la FFT en un clúster de GPU, utilizando técnicas más estándar que permitan una implementación que pueda ser portada a diferentes arquitecturas sin perder eficiencia en el cálculo.

En el siguiente apartado se van a presentar los objetivos que se han definido para la elaboración de esta tesis.

1.2. Objetivos

El objetivo principal del presente trabajo de investigación consiste en **resolver eficientemente la ecuación de Poisson mediante FFT en un clúster de GPU**. De cara a conseguir una implementación eficiente, una de las optimizaciones más importantes ha consistido en aplicar técnicas de segmentación con el fin de solapar cálculo y comunicación en diversas partes de la resolución.

Se ha hecho uso de una librería específica que ofrece NVIDIA especializada en la resolución de la FFT en GPU conocida como transformada rápida de Fourier de CUDA (CUDA Fast Fourier Transform, cuFFT). El uso de esta librería ha implicado la realización de un análisis de los diferentes parámetros y modos de funcionamiento disponibles, de cara a identificar cuál es la configuración más adecuada para conseguir cálculos de FFT óptimos. Este estudio ha supuesto la definición del siguiente objetivo parcial: Ajustar los parámetros que ofrece la librería cuFFT para optimizar la ejecución de las FFT mediante la **librería cuFFT** tanto en rendimiento como en consumo de memoria.

La resolución de la ecuación de Poisson en un clúster de GPU implica la comunicación de datos entre diferentes GPU de cara a obtener el resultado final. En esta comunicación los datos pueden estar distribuidos en memoria de diferentes maneras, y éstas no siempre tienen porqué ser las más adecuadas para conseguir la mayor eficiencia en la fase de comunicación. De cara a realizar la comunicación de manera eficiente, es importante disponer de un mecanismo que permita reubicar los datos en la memoria de la GPU. Por este motivo, otro de los objetivos parciales consiste en definir una librería

para realizar **transposiciones** 3D en GPU de manera eficiente y que será utilizada en la comunicación MPI interproceso.

1.3. Estructura de la memoria

El estudio realizado en este trabajo de investigación se presenta estructurado de la siguiente manera.

En una primera parte, en la que se incluye este **primer capítulo** en el que nos encontramos, se realiza una introducción al contexto en el que se desarrolla la tesis, se exponen los antecedentes que motivan la realización de la misma y se presentan los objetivos que ésta persigue. En el **capítulo 2** se presentan los conceptos y las tecnologías en los que el trabajo de investigación se ha basado, y a los que se hace continua referencia a lo largo de la memoria.

En una segunda parte, se presentan las aportaciones principales de este trabajo. En el **capítulo 3** se presenta el estudio realizado de una librería que permite resolver eficientemente el cálculo de la FFT en una GPU. Para resolver eficientemente el cálculo de la FFT en un clúster de GPU, es necesario realizar las comunicaciones entre los nodos del clúster de manera óptima. Por lo tanto, es importante disponer de un mecanismo que permita reubicar los datos en la memoria de una GPU de cara a realizar las comunicaciones entre los nodos del clúster de manera eficiente. Para la consecución de esta tarea, en el **capítulo 4** se presentan las diferentes implementaciones que se han creado para realizar transposiciones 3D de manera eficiente en una GPU.

El **capítulo 5** describe la resolución eficiente de la ecuación de Poisson en un clúster de GPU. En su discusión, se hará referencia a lo descrito en los capítulos 3 y 4, y se aplicarán técnicas de segmentación de cálculo con el fin de solapar cálculo y comunicación en diversas partes de la resolución.

Por último, en una tercera parte, la tesis concluye con el **capítulo 6**, que recoge las conclusiones y líneas futuras. En este capítulo se resumen las aportaciones realizadas en la tesis y sus publicaciones asociadas, y se presentan futuras líneas de trabajo que derivan de la misma.

Capítulo 2

Contexto de la investigación

Índice

Este capítulo se divide en cuatro secciones. En la primera sección inicialmente se realiza una breve introducción sobre las características generales de la GPGPU, a continuación se enumeran los principales fabricantes de

GPU y, por último, se incluye un resumen de los modelos de programación que se utilizan con estas tecnologías. En la segunda sección, en primer lugar se presenta la estructura general de una GPU. Seguidamente, se expone la jerarquía de memoria utilizada y cuál es su funcionamiento. Para finalizar esta sección, se recogen las características concretas del hardware de las GPU de NVIDIA, fabricante seleccionado para desarrollar el presente trabajo de investigación. En la tercera sección, se va a presentar el ámbito de aplicación de la programación paralela en este trabajo, incluyendo FFT, transposiciones y la ecuación de Poisson. En la última sección se va a hacer un análisis en torno al estado del arte actual.

2.1. Introducción

En el desarrollo del presente trabajo de investigación se va a hacer uso del concepto GPGPU. El modelo empleado para esta tecnología se basa en el uso combinado de CPU y GPU en un sistema de co-procesamiento heterogéneo. La GPU actúa como un co-procesador que en conjunción con la CPU, puede ayudar a acelerar los cálculos gracias a su enorme potencia de procesamiento paralelo. En principio, cualquier algoritmo que se puede implementar en una CPU también puede ser implementado en una GPU, aunque dependiendo del problema y de las dependencias entre las tareas, lo cual condiciona el grado de paralelismo del problema a resolver, estas implementaciones no serán igual de eficientes en ambas arquitecturas. La elección de la arquitectura más adecuada para una resolución eficiente del problema a resolver vendrá dada por las características de la aplicación a paralelizar.

En la figura 2.1 se puede observar cuál es la distribución interna y la superficie dedicada a computación, memoria y lógica de control, de una arquitectura típica en una CPU y en una GPU. Mientras que en las CPU se dispone de menos unidades funcionales con cachés de mayor tamaño y una lógica de control más compleja, en las GPU se multiplican las unidades funcionales (muchos más simples que las unidades funcionales de las CPU) que son agrupadas junto a unas cachés de menor tamaño y lógicas de control más sencillas. Este esquema permite que las arquitecturas *multicore* (CPU) sean adecuadas para realizar paralelismo de tareas y de datos, pudiendo realizar operaciones complejas aunque con un número de tareas moderado; mientras que las arquitecturas *manycore* (GPU) son adecuadas para realizar paralelismo masivo de datos con operaciones más sencillas.

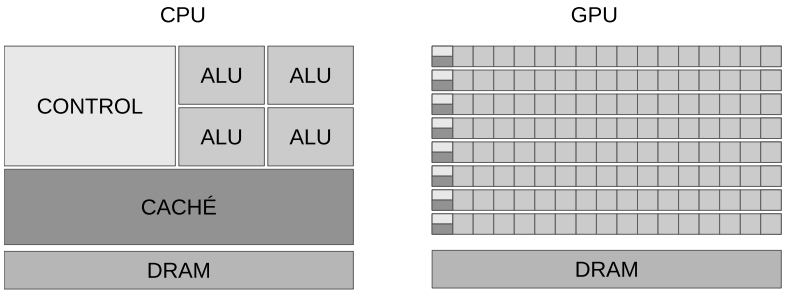

Figura 2.1: Diferencias en la arquitectura entre una CPU *multicore* y una GPU *manycore*

En general, una arquitectura tipo GPU está formada por un conjunto de multiprocesadores compuestos por una serie de procesadores orientados a la ejecución de hilos. Todos los multiprocesadores ejecutan el mismo código (que en el contexto de las GPU se denomina ***kernel***) pero sobre distintos datos. Este paradigma de programación se conoce con el nombre de "un único programa, múltiples datos" (Single Program Multiple Data, SPMD). Internamente, en cada multiprocesador la ejecución se realiza por grupos que siguen el paradigma de programación paralela de "una única instrucción, múltiples datos" (Single Instruction Multiple Data, SIMD). En este tipo de arquitecturas, al igual que en las CPU, la velocidad de ejecución está condicionada por la localidad de los datos (tanto espacial como temporal) y, por tanto, se utilizan memorias caché para acelerar el acceso a los datos. Este es el uso habitual que se le da a una GPU tradicional. No obstante, hoy en día se está avanzando en la arquitectura de la GPU y ya es posible realizar paralelismo entre tareas.

2.1.1. Fabricantes de GPU

Históricamente son varios los fabricantes de GPU que han desarrolado tecnología para GPGPU: AMD, NVIDIA, Intel e IBM. Cada uno de ellos propone su propia arquitectura, que normalmente va acompañada de soluciones software específicas para explotar al máximo las capacidades concretas de cada una de las arquitecturas. AMD y NVIDIA se centran principalmente en desarrollar dispositivos basados exclusivamente en GPU, mientras que Intel e IBM se decantan por soluciones híbridas CPU+GPU. No obstante, son dos los fabricantes que, a día de hoy, copan el mercado del desarrollo de GPU

para su uso en GPGPU: NVIDIA y AMD.

NVIDIA

La empresa NVIDIA se funda en 1993 [41], pero no es hasta 1995 cuando presenta su primer producto, la NVIDIA NV1, una tarjeta de interconexión de componentes periféricos (Peripheral Component Interconnect, PCI) comercializada bajo el nombre de Diamond Edge 3D y dotada de un núcleo de gráficos 2D/3D. En 1999 comercializa la GeForce 256, lo que a partir de ese instante se conocerá como la primera GPU. En el mismo año también se introduce la familia Quadro, una familia de GPU para gráficos de uso profesional. A finales del 2006 NVIDIA presenta una nueva arquitectura, llamada arquitectura de dispositivo de cómputo unificada (Compute Unified Device Architecture, CUDA), y una línea de hardware orientada a la GPGPU, llamada Tesla, y que será comercializada a partir de 2007. Con este movimiento NVIDIA pretende dar respuesta a las necesidades que surgen en la comunidad científica a la hora de usar las GPU para computación de propósito general [42–45]. La familia Tesla ofrece un hardware para computación de altas prestaciones sin ningún tipo de orientación a aplicaciones gráficas.

Una arquitectura genérica para una GPU de NVIDIA dispone de una gran cantidad de unidades de cómputo que se denominan núcleos CUDA o procesador de flujos (Stream Processor, SP). Estos SP están organizados en bloques llamados multiprocesador de flujos (Stream Multiprocessor, SM). A su vez, los SM están agrupados en bloques llamados clúster de procesamiento gráfico (Graphics Processing Cluster, GPC). Cada SP posee su propia unidad de cálculo de coma flotante y una unidad de cálculo de enteros. Cada bloque SM posee también un planificador, junto con una memoria caché de primer nivel y unidades de acceso y filtrado de texturas propias. Estos bloques pueden compartir información con el resto de bloques por medio de un segundo nivel de memoria caché. Finalmente, se dispone de una memoria principal a la que cualquier unidad de cómputo puede acceder directamente.

AMD

La empresa AMD se funda en 1969 [46]. En los primeros años su principal labor consistía en crear microprocesadores para el computador personal

(Personal Computer, PC) de International Business Machines Corp. (IBM). Pasado un tiempo, esta tecnología también fue utilizada por el resto de fabricantes de PC. Su carrera en el desarrollo de GPU no comenzó hasta que en 2006 adquirió la empresa ATI, dedicada a la creación de GPU desde 1985 [47]. A partir de este instante AMD también se une a la carrera para crear hardware orientado a la GPGPU.

La unidad básica de cómputo de una arquitectura GPU tipo AMD es la unidad SIMD. Esta unidad está compuesta por 16 unidades aritmético-lógicas (Arithmetic Logic Units, ALU) y por registros. Varias unidades SIMD forman una unidad de cómputo (Compute Unit, CU). Estas CU, aparte de estar formadas por unidades SIMD, dispondrán de: unidades específicas encargadas de la lógica y el control de saltos, la memoria compartida, el planificador, una unidad escalar –para realizar el trabajo que las unidades aritmético-lógicas (Arithmetic Logic Units, ALU) simples no puedan hacer–, unidades para el manejo de texturas y la caché de primer nivel. Estos CU pueden compartir información con el resto de bloques por medio de un segundo nivel de memoria caché. Finalmente, cualquiera CU puede acceder directamente a la memoria principal de la GPU.

2.1.2. Modelos de programación

Uno de los principales retos a los que se enfrenta un programador a la hora de trabajar con una GPU radica en la dificultad en portar aplicaciones diseñadas para su ejecución en CPU para que se ejecuten eficientemente en GPU. Por este motivo, los fabricantes de GPU han desarrollado modelos de programación, ya sean de código cerrado como CUDA –desarrollado por NVIDIA–, o de código abierto como el lenguaje de computación abierto (Open Computing Language, OpenCL) –adoptado por AMD–, que simplifican considerablemente esta tarea.

CUDA

CUDA es un entorno de programación que ofrece el fabricante NVIDIA para programar sus GPU. Este entorno proporciona: un compilador, un depurador, herramientas de análisis de rendimiento, librerías y los drivers específicos para cada GPU. Utiliza un lenguaje de programación propietario

que es una variante del lenguaje C, llamado CUDA C, aunque sus GPU también pueden ser programadas usando únicamente el lenguaje C.

En la figura 2.2, se detalla las capas de software de CUDA, que incluye: las librerías, el CUDA Runtime y el driver de CUDA.

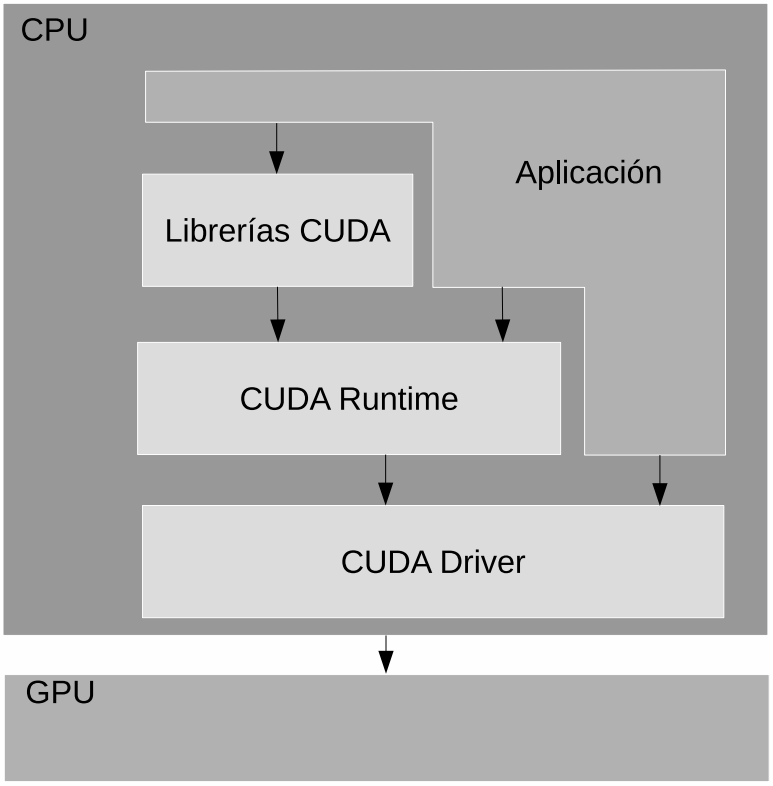

Figura 2.2: Capas de software de CUDA

Las librerías de CUDA son librerías matemáticas de alto nivel, por ejemplo: la librería cuFFT para el cálculo de la FFT en una GPU o la librería cuBLAS, que es una implementación para GPU de funciones para operaciones vectoriales y matriciales básicas (Basic Linear Algebra Subprograms, BLAS). El *Runtime* de CUDA provee una interfaz de programación de aplicaciones (Application Programming Interface, API) y un conjunto de instrucciones accesibles a través de lenguajes de alto nivel como son Fortran o C. Por último, el controlador de dispositivos de CUDA (CUDA Driver) es un controlador de hardware que está dedicado a controlar la GPU.

OpenCL

OpenCL es un entorno de trabajo para escribir programas que se ejecutan en plataformas heterogéneas que constan de: CPU, GPU, procesador digital de señales (Digital Signal Processor, DSP), Field Programmable Gate Array (FPGA) y otros procesadores o aceleradores de hardware. OpenCL especifica un lenguaje de programación multiplataforma –basado en C99– para la programación de estos dispositivos y ofrece una API para controlar la plataforma y ejecutar los programas en los diferentes dispositivos. Además, OpenCL proporciona una interfaz estándar para programación paralela basada en tareas y con paralelismo en base a datos.

OpenCL fue concebida por Apple, a pesar de que lo acabó desarrollando el grupo Khronos [48], que es el mismo que impulsó la librería de gráficos abierta (Open Graphics Library, OpenGL) y el responsable actual de su mantenimiento. Es el lenguaje de programación por el que ha apostado AMD para la programación de sus GPU, aunque también se puede utilizar con las GPU de NVIDIA.

2.2. Ámbito tecnológico

2.2.1. Unidades de cómputo

Una GPU está compuesta por unidades de cómputo cuya función principal consiste en procesar la información que recibe de la CPU. Esta información es transformada mediante las unidades de cómputo, y la información resultante es presentada por pantalla –para los casos en los que se esté trabajando con gráficos– o enviada de vuelta a la CPU –en los casos en los que se estén realizando cálculos matemáticos. En concreto, esta tarea típicamente consiste en recibir la representación de una escena tridimensional como entrada y generar una imagen en 2D como salida. No obstante, debido al creciente uso de las GPU para su uso como GPGPU, los datos de entrada pueden convertirse en matrices de datos sobre los cuales se realizan una serie de cálculos matemáticos antes de ser devueltos a la CPU.

Para el estudio de las unidades de cómputo en una GPU, es útil comprender la estructura de su *pipeline* gráfico (también conocido como tubería de gráficos

o tubería de renderizado). A continuación se presenta el pipeline gráfico basado en la descripción de Kirk y Hwu [49].

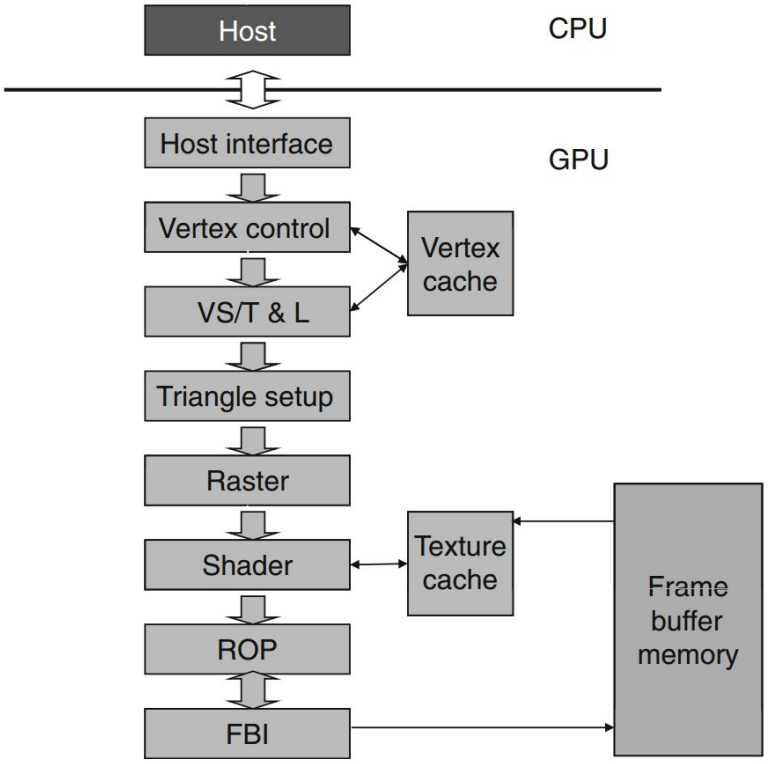

Figura 2.3: *Pipeline* gráfico de una GPU

En la figura 2.3 se puede ver cuál es el esquema que sigue un *pipeline* gráfico de una GPU. La primera etapa del pipeline es la encargada de recibir las imágenes de la CPU en algún formato manejable (normalmente, polígonos en su formato más simple –triángulos–). En la segunda etapa (*Vertex control*), los triángulos son transformados a otra posición. En la siguiente etapa, los *Vertex shaders* se encargan de transformar los vértices visibles, asignándoles diferentes propiedades: colores, texturas, etc. A continuación, la etapa *triangle setup* realiza cálculos para las aristas, interpolando colores y otros valores de los vértices. En la siguiente etapa, el *Raster* se encarga de determinar qué píxeles van a formar parte de cada triángulo, a la vez que les asigna sus propiedades más importantes: sombreado, color, textura, etc. A continuación, los *pixel shaders* se encargan de realizar el sombreado de cada pixel. En la penúltima etapa, el *Raster OPeration (ROP)* aplica las restricciones finales

sobre cada píxel, determinando qué píxeles deben ser visibles y descartando los que no. Finalmente, en la última etapa, los datos se leen y escriben en el buffer de la imagen (*Frame Buffer Interface, FBI*).

2.2.2. Memoria

La memoria de un ordenador es el dispositivo que se encarga de almacenar la información sobre la que se van a realizar cómputos. La duración de los datos en memoria depende de las características inherentes al tipo de memoria que se utilice y a la tecnología con la que se haya construido. Así, una memoria volátil es aquella que requiere energía constante para mantener la información almacenada. Por el contrario, una memoria no volátil retendrá la información almacenada incluso si no recibe corriente eléctrica constantemente.

Las características principales de una memoria son: capacidad, velocidad y coste por bit. En primer lugar, es obvio que a más capacidad mayor es el número de datos que pueden ser almacenados en el sistema. En segundo lugar, la velocidad óptima de trabajo para una memoria es aquella en la que el procesador no tiene que esperar entre cálculos a la hora de leer o escribir datos en ella. Finalmente, el coste por bit no debe de ser excesivo, tiene que ser factible construir un equipo con un coste de memoria adecuado al coste final del sistema.

Los tres factores compiten entre sí, por lo que hay que encontrar un equilibrio entre todos ellos. El objetivo es conseguir una memoria de tamaño suficiente, que disponga de una velocidad que satisfaga nuestras necesidades de rendimiento y cuyo coste no sea excesivo.

Jerarquía de memoria

La jerarquía de memoria es la organización piramidal de la memoria que tienen las computadoras en niveles. El objetivo principal consiste en alcanzar el rendimiento de la memoria más eficaz del sistema, pero a coste de las memorias más económicas (que también son las más lentas). Para la consecución de este objetivo, se hace uso de un principio basado en los comportamientos de los accesos a memoria de los programas al ser ejecutados, que se llama localidad (temporal y espacial).

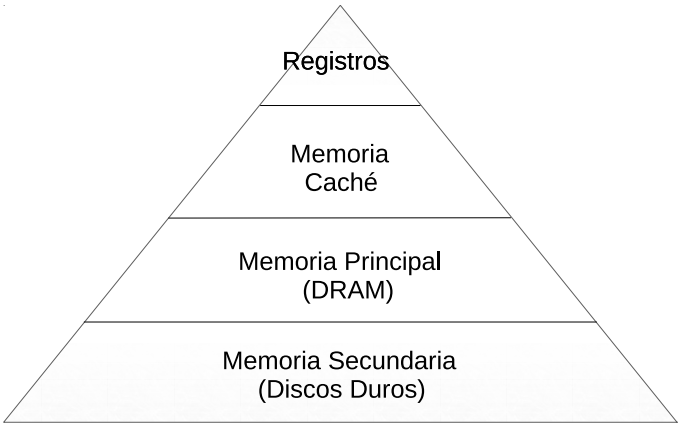

Figura 2.4: Jerarquía de memoria de una CPU

En las figuras 2.4 y 2.5, se presentan las jerarquías de memorias correspondientes a una CPU y a una GPU. En ambos casos, en la parte superior se encuentran las memorias más rápidas (de menor tiempo de acceso a los datos), pero con menor capacidad de almacenamiento (de menor tamaño) y mayor coste. Según se desciende por la pirámide la capacidad de almacenamiento aumenta a la vez que disminuyen el coste y la velocidad de acceso a los datos.

Cada una de las memorias de una GPU que aparecen en la figura 2.5, tiene unos propósitos concretos: los registros son espacios de memoria accesibles únicamente por el hilo de ejecución al que pertenecen. A la memoria compartida puede acceder todo hilo de un mismo bloque. El resto de memorias son completamente accesibles por el programa que se está ejecutando en ese momento. La diferencia entre las memorias principales radica en que la memoria constante es sólo de lectura y la memoria de texturas dispone de una caché espacial.

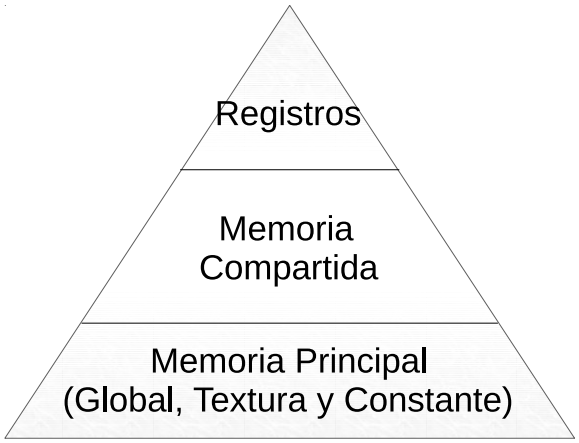

Figura 2.5: Jerarquía de memoria de una GPU

Funcionamiento

En un sistema que dispone de una o de múltiples GPU, normalmente las GPU son vistas como dispositivos externos a la CPU. La comunicación entre CPU y GPU se realiza mediante un bus de comunicaciones dedicado, conocido como interconexión de componentes periféricos exprés (Peripheral Component Interconnect Express, PCIe).

En la figura 2.6 se presenta un ejemplo del flujo de procesamiento que se utiliza en las GPU. Este flujo de procesamiento consta de 4 pasos y es habitual en la mayoría de las arquitecturas GPU, aunque las últimas arquitecturas híbridas que empiezan a aparecer no necesitan los pasos de copia (1 y 4):

1. Copiar datos desde la memoria principal a la memoria global de la GPU.
2. La CPU manda las instrucciones a ejecutar a la GPU.
3. La GPU ejecuta las instrucciones paralelamente en cada núcleo.
4. Copiar los datos de los resultados desde la memoria global de la GPU a la memoria principal de la CPU.

Es importante recalcar que, debido a su lentitud, uno de los objetivos debe consistir en minimizar la copia de datos a realizar tanto en el primer paso como en el cuarto paso.

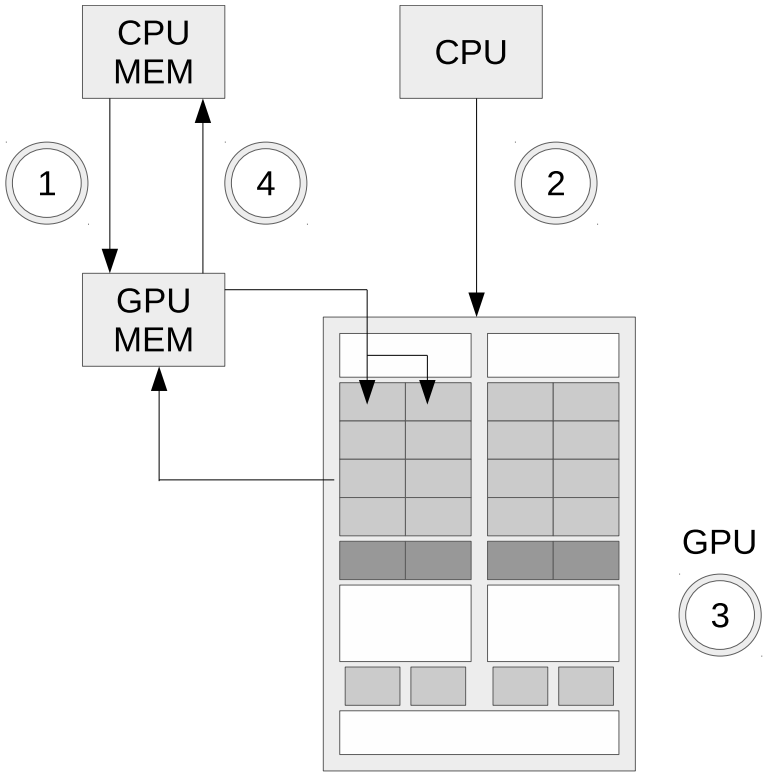

Figura 2.6: Ejemplo del flujo de procesamiento entre una CPU y una GPU

En el presente trabajo de investigación la tecnología que se va a utilizar está basada en las GPU de NVIDIA, y por lo tanto, a partir de este punto toda referencia tecnológica que se incluya corresponde a esta tecnología.

2.2.3. Tecnologías hardware

En este apartado se realiza un resumen de las tecnologías que permiten ejecutar código CUDA C. En primer lugar se listan las gamas de GPU comercializadas por NVIDIA. A continuación, se explica en qué consiste el término "capacidad de cómputo" usado por NVIDIA. Más tarde, se presentan

las diferentes versiones de la arquitectura CUDA. Finalmente, se explican los conceptos básicos de programación de la arquitectura CUDA.

Gamas GPU de NVIDIA

Dentro de la cartera de productos que comercializa NVIDIA existe un conjunto muy extenso de productos centrados en la aceleración gráfica y la computación de propósito general.

En la siguiente clasificación podemos ver cuáles son las gamas que comercializa NVIDIA y a qué mercado están orientadas:

- **GeForce**, orientada al consumidor no profesional para realizar tareas multimedia (videojuegos, edición de vídeo, fotografía digital, etc.) [50].
- **Quadro**, orientada a ofrecer soluciones profesionales para el diseño asistido por ordenador y para la creación de contenido digital [51].
- **Tegra**, orientada a dispositivos móviles [52].
- **Tesla**, orientada al cómputo de propósito general de altas prestaciones [53].

Para nuestro trabajo la gama que realmente nos interesa es la **Tesla** –dado que el cálculo científico requiere habitualmente computación de alto coste. Aunque por cuestiones económicas es habitual utilizar las gamas GeForce o Quadro en fase de desarrollo. A continuación se van a analizar las diferentes versiones de la arquitectura CUDA que han ido surgiendo a lo largo de los últimos años, haciendo hincapié en las características más importantes de cada una de ellas. En este punto hay que aclarar que la compañía NVIDIA utiliza el nombre de **Tesla** para nombrar dos conceptos diferentes dentro de su cartera de productos. Mientras que cuando hablamos de la **gama Tesla** nos estamos refiriendo al mercado al que las GPU están dirigidas (aquí se agrupan todas las GPU orientadas al cómputo de propósito general de altas prestaciones), también se utiliza la palabra **Tesla** para identificar a un tipo de **arquitectura** concreta de NVIDIA.

Compute Capability (CC)

En el ciclo de vida de toda arquitectura de GPU de NVIDIA, se hacen

pequeñas modificaciones que no afectan a la arquitectura principal, pero sí a ciertas capacidades. Con el objeto de informar cuáles son las capacidades correspondientes a una arquitectura, se utiliza el concepto de capacidad de cómputo (Compute Capability, CC).

La CC de un dispositivo se representa por dos números separados por un punto y se conoce como la versión de CC. Esta versión identifica cuáles son las características soportadas por la arquitectura de una GPU que utiliza CUDA y es usada por las aplicaciones para determinar cuáles de ellas pueden ser utilizadas en tiempo de ejecución.

El formato de la CC es X.Y, donde el número mayor de la versión corresponde a X y el número menor de la versión a Y. Los dispositivos que contengan el mismo número mayor de la versión forman parte de la misma arquitectura. Por ejemplo: los dispositivos que tienen como número mayor de la versión el número 6 forman parte de las arquitectura Pascal, los que tienen el número 5 forman parte de la arquitectura Maxwell, los que tienen el número 3 forman parte de la arquitectura Kepler, el 2 corresponde a la arquitectura Fermi y el 1 corresponde a la arquitectura Tesla. El número menor de la versión corresponde a pequeñas mejoras que se han ido introduciendo en la arquitectura. Normalmente implica una revisión de la misma, aunque en ocasiones también incluyen nuevas características.

La versión CC de una particular GPU no debe ser confundida con la versión de CUDA (por ejemplo: CUDA 6, CUDA 6.5, CUDA 7, etc.), la cual corresponde a la versión de la plataforma software de CUDA. Esta plataforma es usada por los desarrolladores de aplicaciones para crear programas capaces de ejecutarse en diferentes generaciones de arquitecturas de GPU. Aunque habitualmente la plataforma añade soporte nativo para las nuevas arquitecturas soportando la versión de la CC de las mismas, también puede incluir características de software nuevas que son independientes de la arquitectura, por ejemplo: optimizaciones en el compilador, mejoras en el *profiler* o la inclusión de nuevas características y funcionalidades en las librerías del API (CUDA Fast Fourier Transform, CUDA Basic Linear Algebra Subprograms, etc.).

En la figura 2.7 se presenta un gráfico con la evolución en el tiempo de las arquitecturas comercializadas por NVIDIA [54]. En este gráfico se presentan los nombres de las arquitecturas GPU junto con su CC. En el *eje x* se indica la fecha de lanzamiento, mientras que en el *eje y* se cuantifica la potencia de cálculo mediante la operación Single precision floating General Matrix

Multiply (SGEMM) –en Gflop/s– por vatio.

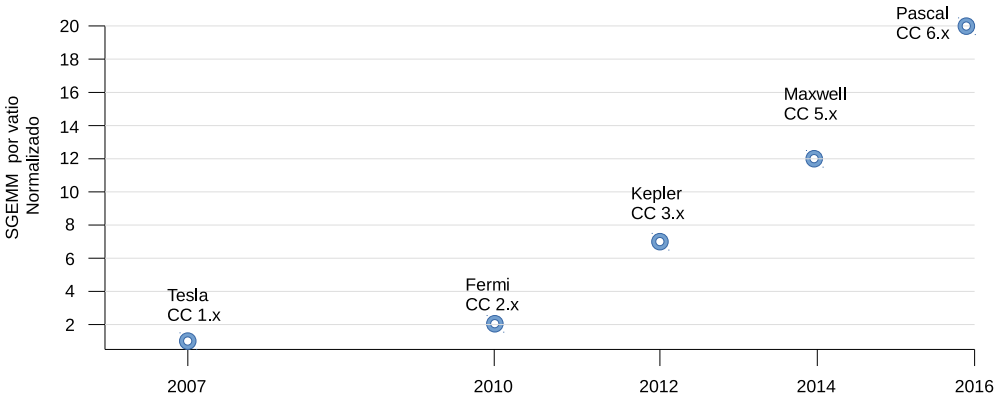

Figura 2.7: Evolución de las GPU de NVIDIA

Arquitecturas de NVIDIA

Tesla [55] –comercializada en 2007– es el nombre de la primera arquitectura de NVIDIA que implementa el modelo CUDA. El nuevo diseño de la arquitectura supone un cambio desde las unidades funcionales independientes (sombreador de píxel y de vértices) que se encontraban en las anteriores GPU a la nueva colección homogénea de los procesadores de coma flotante, llamados SP, que permiten la realización de tareas más universales. Esta arquitectura unificada consiste en un número de SP que, al contrario a lo que la antigua solución proponía –basada en procesadores vectoriales–, es escalar y por lo tanto, sólo puede operar en un único componente simultáneamente. Esto hace que los SP sean menos complejos de construir mientras se mantiene su flexibilidad. Estas unidades escalares en muchos casos se comportan de manera más eficiente que las unidades vectoriales, ya que su menor ancho de banda máximo es compensado por su gran eficiencia de cómputo y por permitir una frecuencia del reloj mucho más alta en su ejecución. Este aumento en la frecuencia es posible gracias a la simplicidad en su construcción.

En las figuras 2.8 y 2.9 se representa el esquema interno de las GPU basadas en la arquitectura Tesla. Internamente está formada por un conjunto de procesadores escalables (Scalable Processor Array, SPA), que en el caso de la tarjeta gráfica de las figuras –una GeForce 8800– dispone de 128 núcleos SP organizados en 16 SM agrupados en 8 unidades de procesamiento indepen-

dientes conocidos como clúster de procesador de texturas (Texture Processor Cluster, TPC) y con 6 módulos de memoria.

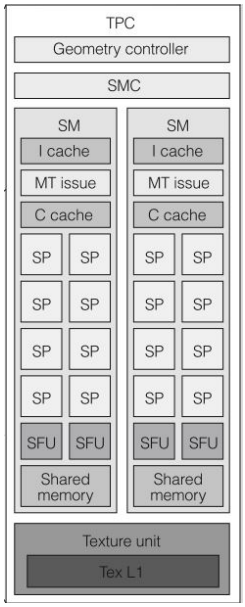

Figura 2.8: Arquitectura interna de un TPC de Tesla

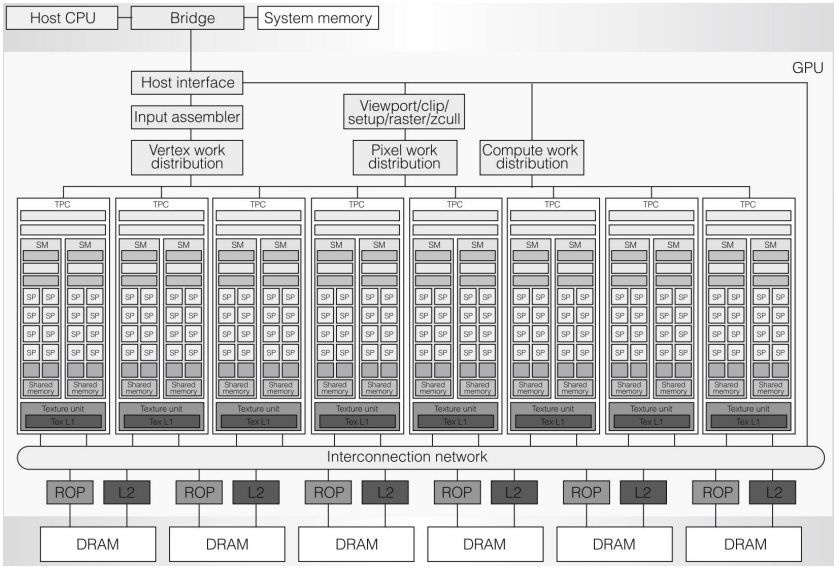

Figura 2.9: Arquitectura Tesla de NVIDIA

En la tabla 2.1 se listan las versiones de CC que se implementaron a lo largo del ciclo de vida de la arquitectura Tesla junto a las GPU que forman parte de cada versión.

Versión CC	GPU
1.0	G80
1.1	G92, G94, G96, G98, G84, G86
1.2	GT218, GT216, GT215
1.3	GT200, GT200b

Tabla 2.1: Implementaciones de la arquitectura Tesla con sus CC

Fermi [56] –comercializada en el 2010– es el nombre de la arquitectura de NVIDIA que sucede a la arquitectura Tesla.

Las GPU correspondientes a esta arquitectura están compuestas por un array escalable de GPC –equivalente al TPC de la generación anterior–, varios SM y 6 módulos de memoria.

En las figuras 2.10 y 2.11 se representa el esquema interno de la GPU GF100 basada en la arquitectura Fermi. En total dispone de 512 núcleos SP organizados en 16 SM con 32 núcleos SP cada uno. Cada SM es un multiprocesador altamente paralelizado capaz de soportar hasta 1536 hilos de ejecución de manera simultánea.

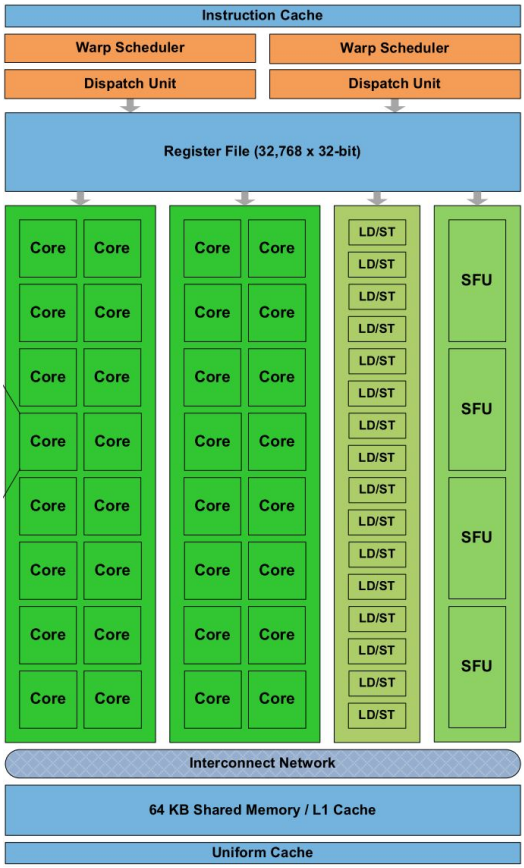

Figura 2.10: Arquitectura interna de la GPU GF100 para cada SM Fermi

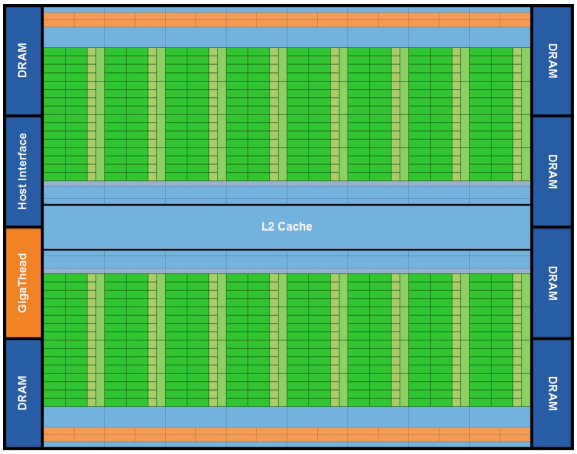

Figura 2.11: Arquitectura Fermi de la GPU GF100 con 16 SM

En la tabla 2.2 se listan las versiones de CC que se implementaron a lo largo del ciclo de vida de la arquitectura Fermi junto a las GPU que forman parte de cada versión. Las funcionalidades más importantes de esta versión son: incluye funciones atómicas para operar con números enteros y de coma flotante tanto de 32-bit como de 64-bit e intercambiar datos tanto en memoria principal como en memoria compartida, permite trabajar con números de doble precisión en coma flotante, implementa bloques de hilos en estructuras de tres dimensiones e implementa funciones para la sincronización y gestión de hilos de ejecución.

Versión CC	GPU
2.0	GF100, GF110
2.1	GF104, GF106, GF108, GF114, GF116, GF117, GF119

Tabla 2.2: Implementaciones de la arquitectura Fermi con sus CC

Kepler [57] –comercializada en el 2012– es el nombre de la arquitectura de NVIDIA que sucede a la arquitectura Fermi.

La arquitectura Kepler de NVIDIA soporta acceso directo remoto a memoria (Remote Direct Memory Access, RDMA) mediante NVIDIA GPUDirect, el cual está diseñado para mejorar el rendimiento del sistema que la alberga permitiendo el acceso directo a la memoria de la GPU por parte de dispositivos de terceros como: conectores InfiniBand (IB), tarjeta de interfaz de red (Network Interface Card, NIC) y unidades de disco de estado sólido (Solid State Disk, SSD).

Al igual que ocurre con las GPU de la arquitectura Fermi, las GPU con arquitectura Kepler están compuestas por diferentes configuraciones de GPC, varios multiprocesador de flujos de siguiente generación (next-generation stream multiprocessor, SMX) –que son una evolución de los SM de la generación anterior– y 4 controladores de memoria.

En las figuras 2.12 y 2.13 se representa el esquema interno de la GPU GTX680 basadas en la arquitectura Kepler. Consiste en 4 GPC, 8 SMX y 4

controladores de memoria. En su totalidad implementa hasta 1536 núcleos CUDA. Aunque el bloque de alto nivel que domina esta arquitectura es el GPC, el motor de la GPU corresponde a los SMX, espacio de la arquitectura donde residen las unidades de cómputo dedicadas al procesamiento gráfico.

Por razones de eficiencia energética, para la arquitectura de la GPU GTX680 se ha optado por integrar 8 unidades SMX, a diferencia de las 16 unidades SM utilizadas en las generaciones anteriores. Las unidades SMX son las responsables de conseguir una mejora en la eficiencia energética de la arquitectura Kepler respecto a generaciones anteriores, debido a que las frecuencias de reloj por núcleo han disminuido respecto a las versiones anteriores, llegando a consumir un 90 % menos que las GPU de la arquitectura Fermi.

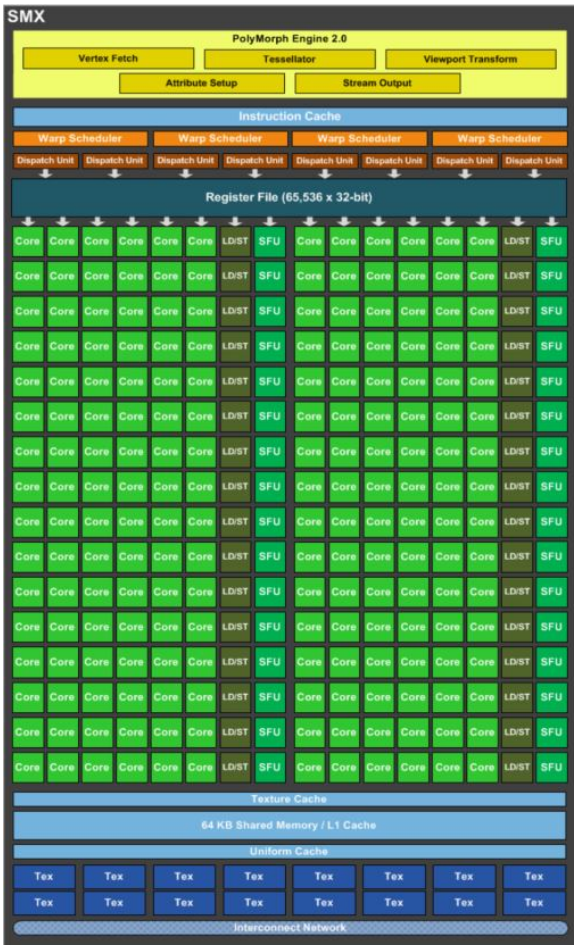

Figura 2.12: Esquema de un SMX de la GPU GTX680 de Kepler

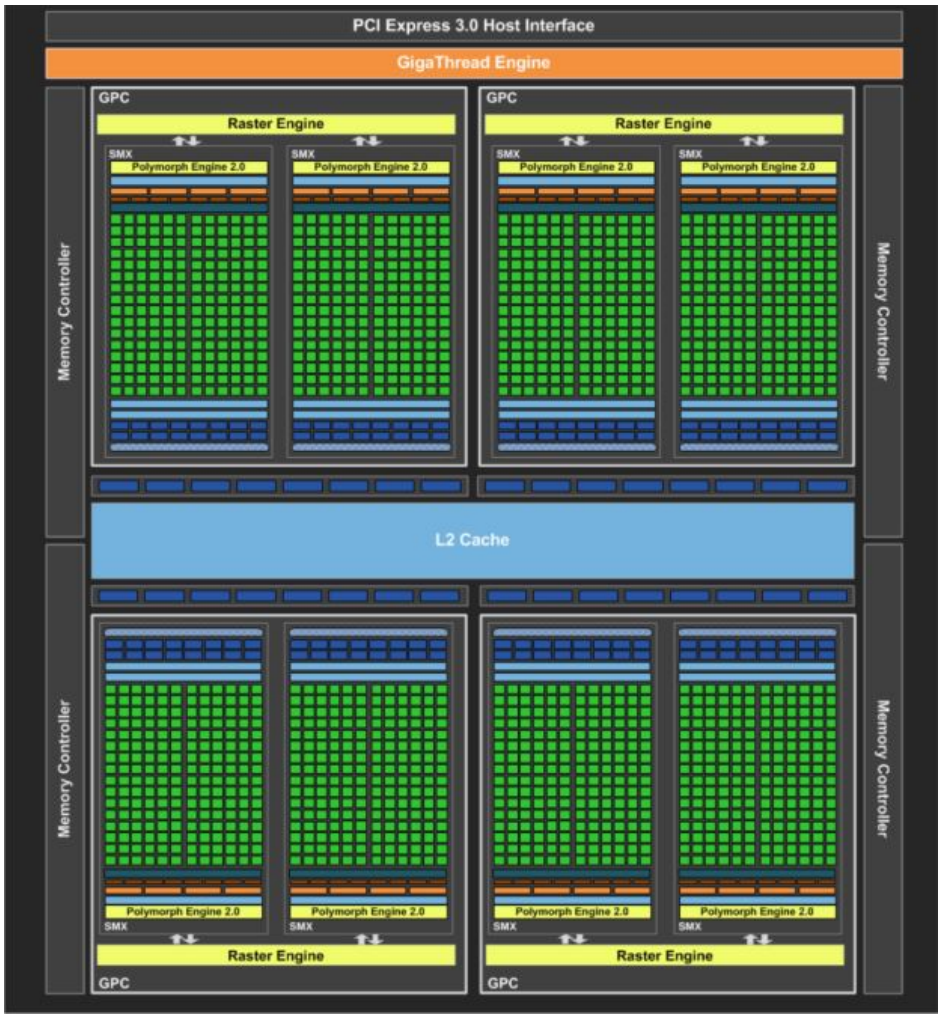

Figura 2.13: Arquitectura Kepler de la GPU GTX680

En la tabla 2.3 se listan las versiones de CC que se implementaron a lo largo del ciclo de vida de la arquitectura Kepler, junto a las GPU que forman parte de cada versión. La funcionalidad más importante en esta versión es que permite la programación de la memoria unificada. La memoria unificada es una evolución del direccionamiento virtual unificado (Unified Virtual Addressing, UVA) presente en anteriores arquitecturas, donde se permitía al programador tratar las memorias de la CPU y de la GPU como un todo y operar con los datos sin tener que preocuparse de si residen en la memoria de la CPU o en la memoria de la GPU. De este modo se facilita la programación evitando al programador tener que realizar tareas de gestión de memoria manualmente.

La memoria unificada se diferencia de UVA en que añade la funcionalidad y todos los mecanismos de control necesarios para migrar automáticamente el contenido de una memoria física a otra. A partir de la versión 3.2 se incluyó la posibilidad de concatenar dos registros de 32 bits para formar uno de 64 bit (*Funnel shift*). Por último, para las versiones 3.5 y 3.7 se incluye la posibilidad de realizar un paralelismo dinámico. Este nuevo paradigma permite lanzar *kernels* desde los propios hilos de ejecución que se están ejecutando en el dispositivo. En definitiva, los hilos de ejecución pueden lanzar nuevos hilos.

Versión CC	GPU
3.0	GK104, GK106, GK107
3.2	Tegra K1
3.5	GK110, GK208
3.7	GK210

Tabla 2.3: Implementaciones de la arquitectura Kepler con sus CC

Maxwell [58] –comercializada en el 2014– es el nombre de la arquitectura de NVIDIA que sucede a la arquitectura Kepler. Al igual que en la arquitectura Fermi y en la arquitectura Kepler, las GPU con arquitectura Maxwell están compuestas por diferentes configuraciones de GPC, varios multiprocesador de flujos de Maxwell (Maxwell stream multiprocessor, SMM) –que son una evolución de los SMX de la generación anterior– y controladores de memoria.

En las figuras 2.14 y 2.15 se representa el esquema interno de la GPU GM204 basada en la arquitectura Maxwell. Consiste en 4 GPC, 16 SMM y 4 controladores de memoria. En su totalidad implementa hasta 2048 núcleos CUDA.

Debido a las mejoras en los procesos de integración de elementos en los circuitos, esta nueva generación de GPU duplica el número de SMM respecto a la arquitectura anterior, sin un aumento considerable en el tamaño. Los SMM constituyen el corazón de las GPU ya que prácticamente la totalidad de las operaciones que se realizan en una GPU pasan por un SMM. Y es aquí

dónde las GPU con arquitectura Maxwell han hecho un especial esfuerzo en aumentar el rendimiento por vatio respecto a las generaciones anteriores.

Como resultado, cada núcleo de la arquitectura Maxwell es capaz de conseguir un mayor rendimiento que su predecesor, e incrementa el rendimiento total obtenido por vatio. A nivel de SMM, aunque disponga de un 33 % menos de núcleos que en la arquitectura anterior, se consigue mantener un rendimiento similar. Gracias al espacio libre extra que se ha generado por la nueva versión de SMM, se ha podido duplicar el número de SMM integrados en la GPU y, por tanto, su rendimiento.

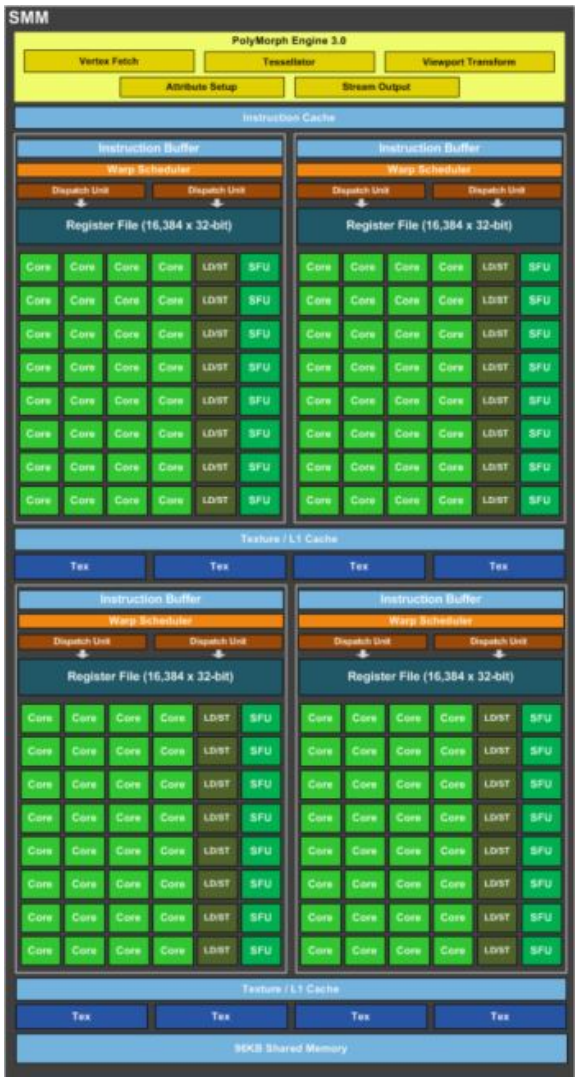

Figura 2.14: Esquema de un SMM de la GPU GM204 de Maxwell

Figura 2.15: Arquitectura Maxwell de la GPU GM204

Versión CC	GPU
5.0	GM107, GM108
5.2	GM200, GM204, GM206
5.3	Tegra X1

Tabla 2.4: Implementaciones de la arquitectura Maxwell con sus CC

En la tabla 2.4 se listan las versiones de CC que se implementaron a lo largo del ciclo de vida de la arquitectura Maxwell, junto a las GPU que forman parte de cada versión. La funcionalidad más importante en esta versión es que se han incluido nuevas operaciones para trabajar con números en coma flotante de mitad de precisión (de 16 bits).

Pascal [59] –comercializada en el 2016– es el nombre de la arquitectura de NVIDIA que sucede a la arquitectura Maxwell.

Al igual que en la arquitectura Fermi, en la arquitectura Kepler y en la arquitectura Maxwell, las GPU con arquitectura Pascal están compuestas por diferentes configuraciones de GPC, varios SM y controladores de memoria.

En las figuras 2.16 y 2.17 se representa el esquema interno de la GPU GP100 basada en la arquitectura Pascal. Consiste en 6 GPC, 60 SM y 8 controladores de memoria. En su totalidad implementa hasta 3840 núcleos CUDA.

Nuevamente, tal y como ya se hizo con la anterior arquitectura Maxwell, han hecho un especial esfuerzo en aumentar el rendimiento por vatio respecto a las generaciones anteriores.

Figura 2.16: Esquema de un SMM de la GPU GP100 de Pascal

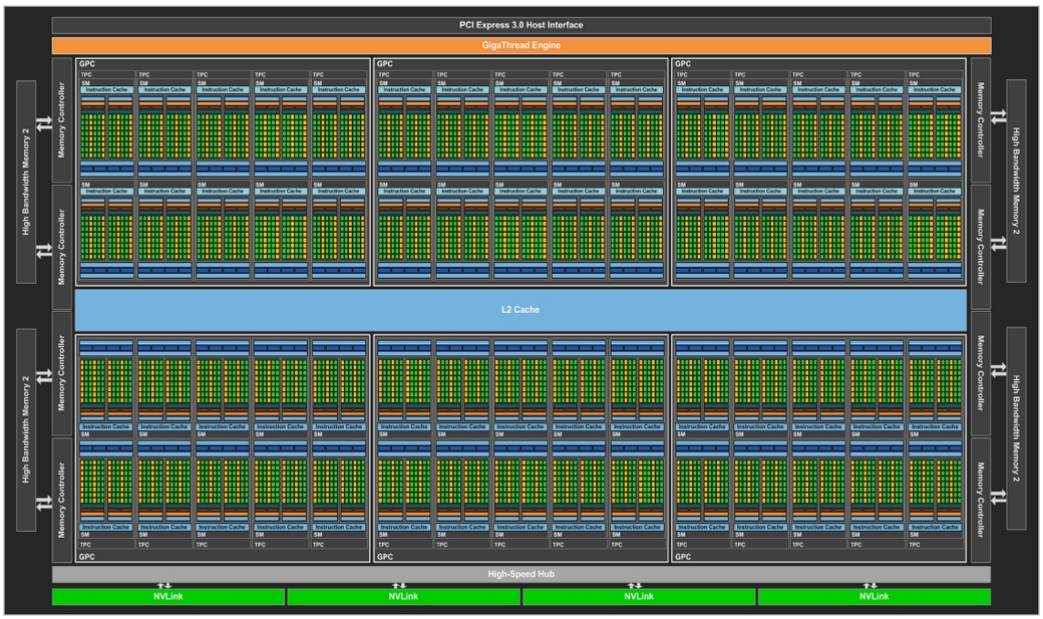

Figura 2.17: Arquitectura Pascal de la GPU GP100

Debido a su reciente comercialización, la única versión de CC que se ha implementado es la 6.0 (ver tabla 2.5).

Versión CC	GPU
6.0	GP100 (Tesla P100)

Tabla 2.5: Implementaciones de la arquitectura Pascal con sus CC

En esta nueva versión no hay funcionalidades nuevas a destacar, más bien se implementan un conjunto de optimizaciones a las funcionalidades ya existentes. En especial, se ha aumentado el rendimiento para los cálculos de precisión doble para 32 y 64bit, así como para los cálculos en coma flotante con 16 bit, y se han mejorado las funciones para acceder a la memoria de manera atómica.

A modo de resumen, en la tabla 2.6 se presentan las principales características que soporta cada versión de CC a partir de la arquitectura Fermi.

Característica soportada	CC					
	2.x	3.0	3.2	3.5, 3.7, 5.0, 5.2	5.3	6.0
Programación de la memoria unificada	No	Sí				
Funnel shift	No		Sí			
Paralelismo dinámico	No			Sí		
Funciones coma flotante 16bit	No					Sí

Tabla 2.6: Principales características soportadas por cada versión de CC a partir de la arquitectura Fermi

Por último, en la hoja de ruta ya se encuentra planificada la siguiente arquitectura para las GPU de NVIDIA con el nombre de **Volta**, que tiene prevista su aparición en 2018.

CUDA

Además de la arquitectura CUDA, descrita anteriormente, también se denomina CUDA a toda la plataforma software diseñada por NVIDIA para poder desarrollar programas para GPU de la arquitectura CUDA. Está compuesta por una plataforma de computación paralela y una API que permite a los desarrolladores de software usar GPU para GPGPU.

La plataforma CUDA es una capa de software que permite el acceso directo al juego de instrucciones y a las unidades de cómputo paralelas que poseen las GPU de NVIDIA.

La primera versión de CUDA se presentó en junio de 2007 (versión 1.0), actualmente se encuentra en la versión 7.5 (publicada en septiembre de 2015) y ya se ha confirmado la versión 8.0 (que será publicada en agosto de 2016) [60]. En la figura 2.18 se presentan las diferentes versiones publicadas de CUDA hasta la actualidad.

Figura 2.18: Versiones publicadas de CUDA

Características del modelo de programación CUDA

El modelo de programación CUDA está compuesto por dos entornos de ejecución distintos pero conectados entre sí. El primero se conoce como ***host*** y se refiere al entorno donde se ejecuta el código de la CPU. El *host* también es el encargado de gestionar las llamadas a las funciones que se van a ejecutar en la GPU. A estas funciones se les conoce con el nombre de *kernels*. El segundo entorno se conoce como ***device*** y se refiere al entorno donde se ejecuta el código de la GPU. Un kernel de CUDA se ejecuta mediante una malla (*grid*) de hilos de ejecución (*thread*). Todos los hilos que forman parte del mismo *grid* ejecutan el mismo código. El identificador de cada hilo se suele utilizar parar acceder a diferentes datos de memoria y para tomar las decisiones de control pertinentes. Por lo tanto, se puede afirmar que la unidad básica de ejecución es un *thread*.

Los hilos de ejecución están organizados en bloques (*block*) de *threads*. Cada bloque se ejecuta en un SM mientras que cada thread se ejecuta en un SP. Estos bloques componen la anteriormente mencionada malla (*grid*) de bloques. La tarea de un *grid* consiste en ejecutar un *kernel* sobre un conjunto de datos. Un *grid* suele disponer de un número alto de bloques, por lo que cada SM ejecuta múltiples bloques. Esta abstracción permite que el mismo código pueda ejecutarse en GPU con distinto número de SM.

En la figura 2.19 se puede observar cuál es la distribución de los hilos de ejecución en bloques y *grids*.

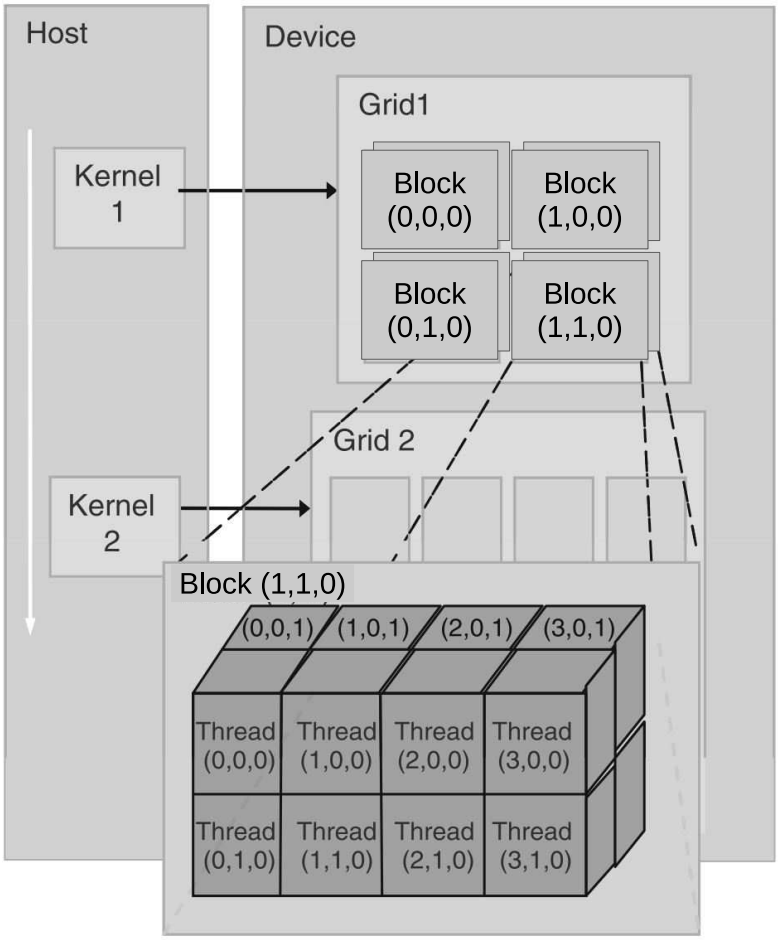

Figura 2.19: Esquema de la distribución de los hilos de ejecución en bloques y *grids*

En un programa escrito en CUDA los hilos se identifican mediante la variable *threadIdx*. Esta variable es un vector de elementos de hasta 3 dimensiones (x,y,z), donde la dimensión de la variable *threadIdx* depende de las necesidades en la ejecución de cada *kernel*. Dependiendo de las necesidades, un *grid* puede ser de hasta 3 dimensiones e internamente dispondrá de bloques que también pueden ser de hasta 3 dimensiones, donde cada uno de ellos puede ser direccionado mediante la variable *blockIdx*. Del mismo modo, la dimensión del bloque puede ser obtenida desde el propio kernel mediante la variable *blockDim*. A día de hoy el número de *threads* por bloque es de hasta 1024 y depende de la CC de la tarjeta gráfica. Los hilos que se encuentran en un mismo bloque pueden cooperar entre sí mediante el uso de memoria compartida y para su

sincronización pueden utilizar el método de sincronización por barreras.

El multiprocesador CUDA divide los bloques de hilos en grupos de 32 hilos consecutivos llamados *warp*. Cada vez que se va a ejecutar una nueva instrucción, el planificador selecciona un *warp* disponible y lanza la misma instrucción para todos los hilos de ese *warp*. Si los hilos de un *warp* divergen debido a una condición de salto, la ejecución de los hilos del *warp* se serializa, ejecutando primero los hilos de un camino para después ejecutar los hilos del otro camino.

Un hilo de ejecución puede acceder a las diferentes memorias dentro de la jerarquía de memoria de una GPU. En primer lugar, puede acceder a su propia memoria local. En segundo lugar, también puede acceder a la memoria compartida por todos los hilos pertenecientes a un mismo bloque de hilos de ejecución. Por último, todos los hilos tienen acceso a la memoria global. Además, existen otros dos tipos de memoria que sólo son accesibles en modo lectura: la memoria constante y la memoria para las texturas.

Acceso coalescente a memoria

En la sección 2.2.2, al analizar la jerarquía de memoria, se ha indicado que la memoria principal es la memoria con mayor capacidad de almacenamiento y también la de mayor latencia (ver la figura 2.5). Esta latencia es la característica que más impacto tiene en el rendimiento de las aplicaciones que se ejecutan en este tipo de arquitecturas. Por ello, para aprovechar el gran ancho de banda en el acceso a memoria que ofrecen las GPU, es de vital importancia acceder a la memoria principal de manera coalescente. La coalescencia se define como la capacidad de una GPU para obtener varias palabras de memoria simultáneamente (en un único acceso) por medio de los hilos de un *warp*. La coalescencia se alcanza bajo ciertos patrones de acceso, y estos patrones dependen de la capacidad de cómputo de la GPU [61]. Así, los accesos a la memoria principal se realizarán mediante transacciones de memoria de 32, 64 o 128 bytes. Estas transacciones de memoria deben estar alineadas de forma natural, ya que sólo los segmentos de 32, 64 o 128 bytes de la memoria principal que se encuentren alineados con su tamaño –es decir, cuya primera dirección sea un múltiplo de su tamaño– podrán ser accedidos tanto para lectura como para escritura en una misma transacción de memoria. Cuando los hilos de un *warp* ejecutan una instrucción que accede a la memoria principal, la GPU intenta realizar todos los accesos con el menor número de transacciones de memoria posibles. Estas transacciones de memoria dependen directamente del tamaño de la palabra a la que se accede por cada hilo y de

la distribución de las direcciones de memoria a las que hay que acceder a lo largo de los diferentes hilos. En general, cuantas mas transacciones se realicen mayor será el número de palabras de memoria que no son necesarias por los hilos las que van a ser transferidas en comparación con las estrictamente necesarias y, por lo tanto, el rendimiento disminuirá de manera acorde a este fenómeno. Por ejemplo, si se realizan transacciones de memoria de 32 bytes para acceder a 4 bytes de datos por cada hilo, el rendimiento se divide por 8.

Tal y como se ha comentado, el número de transacciones de memoria necesarias y el rendimiento que se va a obtener dependen directamente de la capacidad de cómputo de la GPU en donde se estén realizando esas transacciones. Así, los accesos a la memoria principal serán de medio *warp* para GPU con una capacidad de cómputo de 1.X o de un *warp* para GPU con una capacidad de cómputo de 2.0 o superior. Mediante la coalescencia se consigue una mejoría en el ancho de banda de acceso a la memoria en un factor igual al número de hilos de *warp* o del medio *warp*. Esto es, de un factor de 16x para un CC 1.X y de un factor de 32x para un CC 2.0 o superior.

En el caso de las GPU con una capacidad de cómputo de 2.x, los hilos de un *warp* realizarán accesos coalescentes a memoria cuando todos los datos requeridos por esos hilos se consiguen en una única transacción.

En las figuras 2.20, 2.21, 2.22 y 2.23 se representan los conceptos de la coalescencia. En estos ejemplos –a no ser que se indique lo contrario– se asume una CC de 2.x, los accesos a memoria se realizan en unidades de transferencia de 128 bytes y cada hilo accede a 4 bytes.

El caso más sencillo en el que se consigue la coalescencia es aquel en el que los hilos del *warp* acceden a los datos de manera alineada. No es necesario que participen todos los hilos del *warp* e incluso los hilos pueden acceder al mismo dato. Por ejemplo, si los hilos de un *warp* acceden a palabras adyacentes de 4 bytes, se necesitará una único acceso de 128 bytes que permitirá acceder a los datos requeridos en una única transacción coalescente. Este patrón de acceso a los datos se puede ver en la figura 2.20, y el resultado consiste en una única transacción, representada con un color más oscuro. Tampoco es necesario que los hilos del *warp* accedan secuencialmente a los datos para que el acceso a memoria se realice en una única transacción coalescente (ver la figura 2.21).

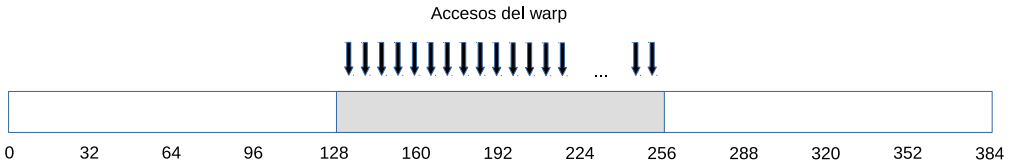

Figura 2.20: Acceso coalescente (datos alineados y secuenciales), todos los hilos consiguen sus datos en una única transacción

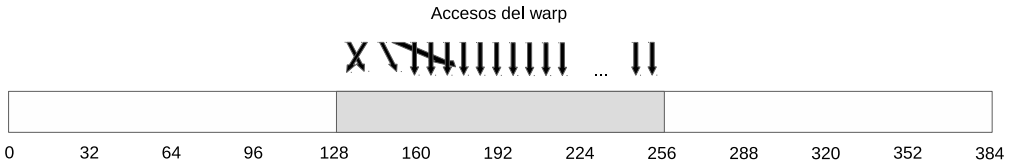

Figura 2.21: Acceso coalescente (datos alineados pero no secuenciales), todos los hilos consiguen sus datos en una única transacción

En los casos en los que hilos secuenciales de un *warp* accedan a datos de memoria que son secuenciales pero que no están alineados, serán necesarios 2 accesos, tal y como se puede ver en la figura 2.22.

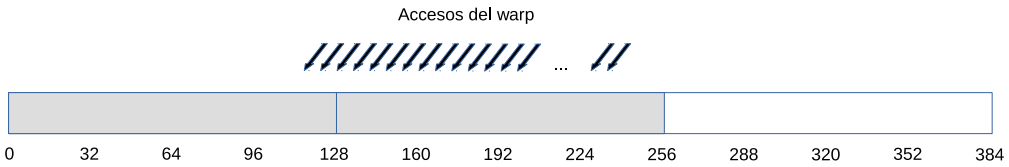

Figura 2.22: Acceso secuencial no alineado que necesita dos transacciones

En ocasiones es conveniente que las transacciones sean de un tamaño menor a los 128 bytes de los ejemplos anteriores. Por ejemplo, cuando los accesos a memoria son aleatorios y no conviene traer más datos de los necesarios. En estos casos pueden realizarse transacciones de 32 bytes, pero tal y como se observa en la figura 2.23 se produce un patrón de acceso equivalente al analizado en la figura 2.22, pero en esta ocasión, se realizan 5 transacciones de menor tamaño (32 bytes) para satisfacer los accesos a memoria.

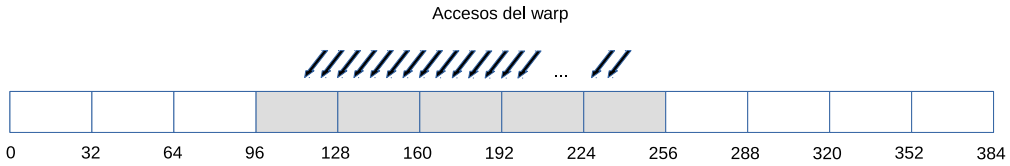

Figura 2.23: Accesos secuenciales no alineados que necesitan cinco transacciones de 32 bytes

2.3. Ámbito matemático

En esta sección, en primer lugar se va a realizar una aproximación a las transformadas de Fourier. A continuación se define la ecuación de Poisson y su resolución. Por último, se va a hablar de las transposiciones, una operación fundamental en este tipo de cálculos.

2.3.1. Análisis de Fourier

El análisis de Fourier, desarrollado por Jean Baptiste Joseph Fourier en el siglo XIX [62], se utiliza para representar una función periódica $f(t)$ como una suma infinita ponderada de términos de senos y cosenos; a esta representación se le conoce como la serie de Fourier. Más tarde, este análisis se generalizó para representar una función no periódica, en este caso, la representación se realiza mediante una integral y se conoce como la transformada de Fourier.

Serie de Fourier

La serie de Fourier se emplea para analizar funciones periódicas a través de la descomposición de dicha función en una suma infinita de funciones sinusoidales de menor complejidad [63, 64]. A continuación se da la definición matemática de la serie de Fourier [65]:

Si $f(t)$ es una función periódica y su periodo es T, la serie de Fourier asociada a $f(t)$ es:

$$f(t)\sim\frac{a_0}{2}+\sum_{n=1}^{\infty}\left[a_n\cos\left(\frac{2n\pi}{T}t\right)+b_n\sin\left(\frac{2n\pi}{T}t\right)\right] \tag{2.1}$$

Donde, a_0, a_n y b_n son los coeficientes de Fourier que toman los valores:

$$a_0=\frac{2}{T}\int_{-T/2}^{T/2}f(t)dt \tag{2.2}$$

$$a_n=\frac{2}{T}\int_{-T/2}^{T/2}f(t)\cos\left(\frac{2n\pi}{T}t\right)dt \tag{2.3}$$

$$b_n=\frac{2}{T}\int_{-T/2}^{T/2}f(t)\sin\left(\frac{2n\pi}{T}t\right)dt \tag{2.4}$$

Por la identidad de Euler [66], la serie de Fourier (ver fórmula 2.1) también pueden expresarse en su forma compleja como:

$$f(t)\sim\sum_{-\infty}^{\infty}C_n e^{2\pi i\frac{n}{T}t},\ \text{donde } i\equiv\sqrt{-1} \tag{2.5}$$

Y por lo tanto, el coeficiente se expresa como:

$$C_n=\frac{1}{T}\int_{-T/2}^{T/2}f(t)e^{2\pi i\frac{n}{T}t}dt \tag{2.6}$$

Transformada de Fourier

La transformada de Fourier se emplea para transformar funciones continuas no periódicas entre diferentes dominios (del tiempo o espacial y de frecuencia) [67, 68].

A continuación se da la definición matemática de la transformada de Fourier [65].

Sea $f(t)$ una función Lebesgue integrable [69, 70]:

$$f\in L^1(\mathbb{R}) \tag{2.7}$$

La transformada de Fourier de f es la función:

$$\mathcal{F}\{f\} : \xi \longmapsto \hat{f}\{\xi\} := \int_{-\infty}^{\infty} f(x)e^{-2\pi i\xi x}dx \tag{2.8}$$

La transformada de Fourier inversa de una función integrable f está definida por:

$$\mathcal{F}^{-1}\{\hat{f}\} = f(x) = \int_{-\infty}^{\infty} \hat{f}(\xi)e^{2\pi i\xi x}d\xi \tag{2.9}$$

Transformada de Fourier Discreta

La DFT es un tipo de transformada discreta utilizada en el análisis de Fourier. Transforma una función matemática en otra, obteniendo una representación en el dominio de la frecuencia, siendo la función original una función en el dominio del tiempo. Requiere que la función de entrada sea una secuencia discreta y de duración finita [71]. Esta transformación únicamente evalúa los suficientes componentes frecuenciales de manera que se pueda reconstruir el segmento finito en el dominio temporal, al contrario de lo que ocurre con la transformada de Fourier en tiempo discreto (Discrete-Time Fourier Transform, DTFT). Utilizar la DFT implica que el segmento que se analiza es un único período de una señal periódica que se extiende de forma infinita; si esto no se cumple, se debe utilizar una ventana para reducir los espurios del espectro. Por la misma razón, la transformada discreta de Fourier inversa (Inverse Discrete Fourier Transform, IDFT) no puede reproducir el dominio del tiempo completo, a no ser que la entrada sea periódica indefinidamente. Por estas razones, se dice que la DFT es una transformada de Fourier para el análisis de señales de tiempo discreto y dominio finito [72].

La datos de entrada de una DFT son una secuencia finita de números reales o complejos, de modo que es ideal para realizar el análisis de Fourier sobre información almacenada en soportes digitales. En particular, la DFT se utiliza comúnmente en procesado digital de señales y otros campos relacionados dedicados a analizar las frecuencias que contiene una señal muestreada, así como para resolver ecuaciones diferenciales parciales, y para llevar a cabo operaciones como convoluciones o multiplicaciones de grandes números ente-

ros. Un factor muy importante para este tipo de aplicaciones es que, en la práctica, la DFT puede ser calculada de forma eficiente utilizando la familia de algoritmos de la FFT.

La DFT, X_k, de una secuencia de longitud finita, x_n, de longitud N es definida como:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn}, \text{ donde } k = 0, ..., N-1 \tag{2.10}$$

La IDFT se define como:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N} kn}, \text{ donde } n = 0, 1, ..., N-1 \tag{2.11}$$

Como x_n puede ser complejo, por cada valor de k en la ecuación 2.10 son necesarias N multiplicaciones complejas y $(N-1)$ sumas complejas para resolver X_k. Por lo tanto, para todos los N valores de k son necesarias N^2 multiplicaciones complejas y $N(N-1)$ sumas complejas. De este modo, el número de operaciones para el cálculo de la DFT crece de manera cuadrática para valores de N grandes [73].

La división inicial $\frac{1}{N}$ de la ecuación 2.11, es necesaria para que la IDFT realmente sea la inversa de la función DFT original. En cualquier caso, esta normalización también se puede realizar en un cálculo posterior a la IDFT, o incluso también puede dividirse en dos pasos donde la mitad de la normalización se realiza en la DFT y la otra mitad en la IDFT.

En muchas aplicaciones, los datos de entrada para una DFT son reales, en cuyo caso, las datos de salida satisfacen la simetría $X_{N-k} = X_k^*$, donde $*$ representa el conjugado complejo. Varios algoritmos para calcular FFT eficientes han sido diseñados para aprovechar esta característica [74]. Esta simetría permite eliminar las partes redundantes de los cálculos, y se consigue ahorrar, hasta en un factor de dos, el tiempo de ejecución y el consumo de memoria. Por ejemplo, es posible expresar una DFT de longitud par y con datos de entrada reales como una DFT con datos complejos y de la mitad de la longitud que la original (donde, la parte real y la parte imaginaria corresponden a los elementos pares e impares de los datos originales), seguido por $O(N)$ operaciones de post-procesamiento.

Fast Fourier Transform (FFT)

Debido a que el cálculo de la DFT crece de manera cuadrática para valores de N grandes, muchos investigadores han centrado sus esfuerzos en reducir el número de operaciones a realizar en el cálculo de la DFT [75]. En 1965, J.W. Cooley y J.W. Tukey [6] presentan un algoritmo que es capaz de reducir el número de operaciones para el cálculo de la DFT de $O\left(N^2\right)$ a $O\left(N\log_2 N\right)$. Este trabajo dio como resultado el desarrollo de una familia de algoritmos numéricos para calcular eficientemente la DFT conocida como FFT [76].

De cara a facilitar la notación se define W_N como:

$$W_N = e^{-i^{(2\pi/N)}} \tag{2.12}$$

La FFT mejora la eficiencia en el cálculo de la DFT explotando las siguientes propiedades del término W_N^{kn} de la ecuación 2.12:

1. Simetría conjugada compleja: $W_N^{k(N-n)} = W_N^{-kn} = (W_N^{kn})^*$.
2. La periodicidad en n y k: $W_N^{kn} = W_N^{k(N+n)} = W_N^{n(k+N)}$.

Aunque la propiedad de la simetría conjugada compleja reduce las multiplicaciones a la mitad, el número de operaciones a realizar sigue siendo proporcional a N^2 [73].

Afortunadamente, la propiedad de periodicidad de W_N^{kn} permite descomponer el cálculo de la DFT en múltiples DFT de menor tamaño que se convertirá en la base para muchas versiones de la FFT.

Muchas de las nuevas librerías para el cálculo de las FFT, como la transformada de Fourier más rápida del oeste (Fastest Fourier Transform in the West, FFTW), usan varias técnicas para acelerar los cálculos, y así reducir el número total de operaciones [7]. La librería cuFFT, desarrollada por NVIDIA, se basa en la librería FFTW, y ha sido diseñada para ser utilizada de manera eficiente por las GPU de NVIDIA [77].

2.3.2. FFT multidimensionales

La DFT consiste en transformar una secuencia o matriz de 1D χ_n siendo una función de una variable discreta n. La DFT multidimensional de un array multidimensional $\chi_{n_1,n_2,...,n_d}$ es una función de d variables discretas $n_\ell = 0, 1, ..., N_\ell - 1$ con valores de ℓ de 1,2,...,d y se defene como [71]:

$$X_{k_1,k_2,...,k_d} = \sum_{n_1=0}^{N_1-1} \left(\omega_{N_1}^{k_1 n_1} \sum_{n_2=0}^{N_2-1} \left(\omega_{N_2}^{k_2 n_2} ... \sum_{n_d=0}^{N_d-1} \omega_{N_d}^{k_d n_d} \cdot \chi_{n_1,n_2,...,n_d} \right) \right) \tag{2.13}$$

donde $\omega_{N_\ell} = exp(-2\pi i/N_\ell)$ y los índices de salida para d son $k_\ell = 0, 1, ..., N_{\ell-1}$. Esta fórmula se puede expresar de forma más compacta usando una notación vectorial, donde $n = (n_1, n_2, ..., n_d)$ y $k = (k_1, k_2, ..., k_d)$ se definen como vectores de dimensión d cuyos índices van desde 0 a $N - 1$ y $N - 1 = (N_1 - 1, N_2 - 1, ..., N_d - 1)$:

$$X_k = \sum_{n=0}^{N-1} e^{-2\pi i k \cdot (n/N)} \chi_n \tag{2.14}$$

donde la división n/N se define como $n/N = (n_1/N_1, ..., n_d/N_d)$ a calcular para cada elemento y la suma denota el conjunto de sumatorios anidados anteriores.

La IDFT multidimensional se define como:

$$\chi_n = \frac{1}{\prod_{\ell=1}^{d} N_\ell} \sum_{k=0}^{N-1} e^{2\pi i n \cdot (k/N)} X_k \tag{2.15}$$

En general, una DFT de m dimensiones puede ser descompuesta en varias DFT de menos dimensiones que sumen m. En este trabajo de investigación se van a utilizar matrices de entrada en 3D, y por este motivo, a continuación se analizan las opciones para descomponer una DFT de 3D en tres DFT de 1D o en una DFT de 2D más una DFT de 1D.

En la figura 2.24 se puede ver la descomposición para una matriz de 3D en los ejes y y z. A este tipo de descomposición se le conoce como una

descomposición de 1D por planos (o 1D *slab*). Todo cálculo que se realice en el estado (a) para los planos x y z se realiza usando la memoria local, mientras que los cálculos para los datos a lo largo de la dimensión y son distribuidos. Cuando es necesario realizar algún cálculo en la dimensión y (por ejemplo, para realizar una FFT de 1D en la dimensión y), se pueden distribuir los datos entre los procesadores para alcanzar el estado (b), donde cualquier cálculo en en la dimensión y se convierte en "local". En el caso de usar la librería MPI, la transición entre los estados (a) y (b) puede realizarse mediante la función *Alltoall*.

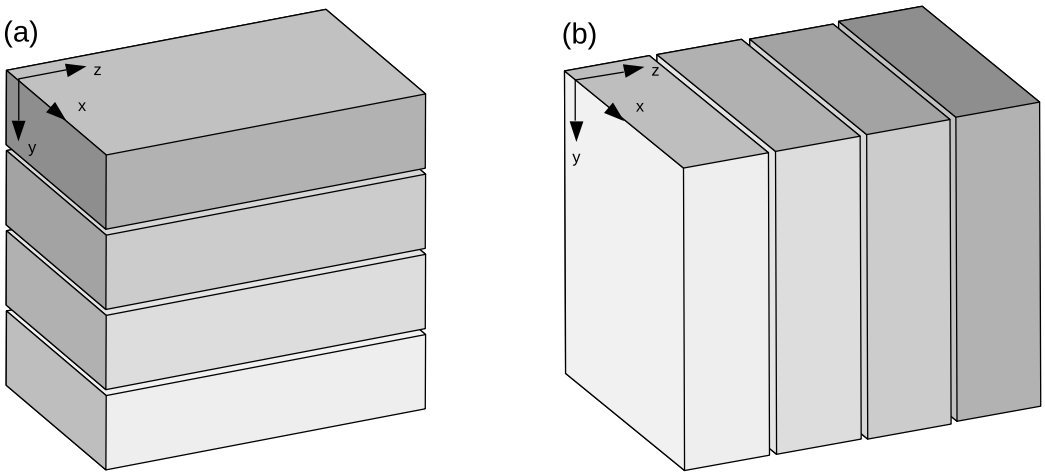

Figura 2.24: Ejemplo de descomposición de un dominio en 1D mediante 4 procesadores: (a) descompuesto en eje Y; (b) descompuesto en eje Z.

Una descomposición en 1D, aún siendo un cálculo sencillo, tiene importantes limitaciones en el grado de paralelismo que permite obtener. Por ejemplo, dada una matriz de entrada cúbica de tamaño N^3, una limitación obvia es que el número máximo de procesos que se puede utilizar en una descomposición 1D es de N, ya que cada *slab* tiene que contener al menos un plano de datos. Su principal ventaja estriba en que requiere un único paso de comunicación.

Una descomposición de bloques en 2D (o 2D *pencil*) es una extensión natural de las descomposiciones en 1D *slab*. La figura 2.25 muestra que la misma matriz de 3D puede ser descompuesta en una matriz de 2D. Los estados (a), (b) y (c) se conocen como *x-pencil*, *y-pencil* y *z-pencil* respectivamente. Mientras que un algoritmo de descomposición de 1D permuta entre dos estados, en una descomposición de 2D se necesitan atravesar 3 estados diferentes utilizando transposiciones globales ((a) =>(b) =>(c)). Una vez más, se

puede realizar un *Alltoall* para realizar las transiciones entre los estados. Sin embargo, esta descomposición es significativamente más compleja y ofrece un peor rendimiento que la descomposición 1D, ya que se necesitan realizar más comunicaciones –operación que supone un gran coste en el tiempo de ejecución. Su principal ventaja estriba en que el grado de paralelismo alcanzable es mayor. Por ejemplo, si tenemos un cubo de datos de entrada de N^3 se necesitan N^2 procesos.

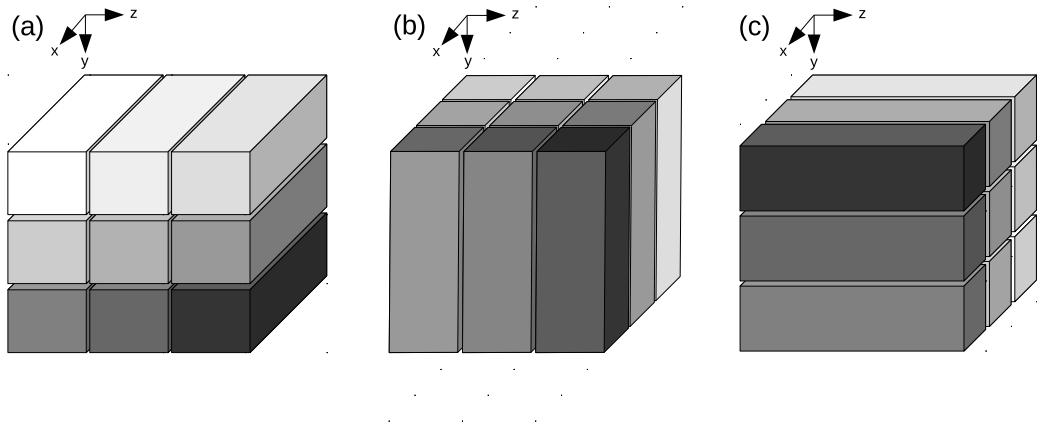

Figura 2.25: Ejemplo de descomposición de un dominio en 2D mediante un grid de 3x3 procesadores: (a) X-pencil; (b) Y-pencil; (c) Z-pencil.

2.3.3. La ecuación de Poisson

La ecuación de Poisson es una ecuación ampliamente usada en electrostática, física teórica, ingeniería mecánica y otras disciplinas científicas. La ecuación se define de la siguiente manera:

$$\Delta\phi = f$$

donde Δ es el operador laplaciano, y ϕ y f son dos funciones reales o complejas.

La ecuación se entiende de manera más clara si la particularizamos, por ejemplo, a 3 dimensiones y coordenadas cartesianas:

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\right)\phi(x,y,z) = f(x,y,z)$$

Es decir, f es la función que obtenemos mediante la suma de las derivadas parciales segundas de ϕ.

Para resolver la ecuación de Poisson hay que obtener ϕ a partir de una f conocida y una condición de contorno determinada. Las condiciones de contorno, determinan la manera en la que se resuelve la ecuación y se clasifican según su tipo. Las condiciones de contorno del tipo *Dirichlet* (también conocidas como de primer tipo) son aquellas en las que se especifican los valores de la solución que necesita la frontera del domino. Las condiciones de contorno de segundo tipo, o de *Neumann*, son aquellas en las que se especifican los valores de la derivada de una solución tomada en la frontera del domino. Otro tipo de condiciones de contorno, como las de *Cauchy*, imponen valores específicos a la solución de una ecuación diferencial que se toma de la frontera del dominio y de la derivada normal en la frontera. Es como aplicar los dos primeros tipos de condiciones al mismo tiempo (*Dirichlet* y *Neumann*). Si las condiciones de contorno se aplican sobre partes diferentes de la frontera del dominio de la ecuación, estaremos usando una condición de frontera mixta. Mientras que si se le especifica una combinación lineal de los valores de una función y los valores de su derivada sobre la frontera del dominio, estaremos aplicando una condición de frontera del tipo *Robin* (las condiciones de frontera de *Robin* son una combinación ponderada de las condiciones del tipo *Dirichlet* y *Neumann*).

La solución analítica de esta ecuación se conoce sólo para un reducido conjunto de funciones por lo que se suele resolver computacionalmente. Para ello, se discretiza la función f por lo que la entrada del algoritmo de resolución es una malla donde el valor de cada punto corresponde al valor de f en dicho punto.

El cálculo directo de esta ecuación requiere $O(N^2)$ operaciones, donde N es el número de puntos de la malla. Sin embargo, existen diversas maneras de resolver esta ecuación discretizada de manera más eficiente, como por ejemplo el *gradiente conjugado* [78], *multigrid* [79] o *Fast Multipole Method* [80]. En este trabajo de investigación se ha utilizado un método, basado en FFT, que ha demostrado una buena capacidad de escalado en clúster con gran cantidad de nodos [5].

En cualquier caso, el hecho de utilizar la FFT para resolver la ecuación de Poisson implica que los cálculos se realizan sobre una serie de datos periódicos que se extienden de forma infinita (en este contexto a las condiciones de contorno se les denomina como condiciones de contorno periódicas –que son precisamente las que se han utilizado en este trabajo–). Si esto no se cumple, se debe utilizar una ventana para reducir los espurios del espectro [20–22], o la técnica a utilizar debe ser adaptada a otro tipo de condiciones de

contorno [23, 81].

La solución a la ecuación de Poisson mediante FFT viene dada por la siguiente ecuación:

$$\phi(r) = \hat{\phi}^{-1}(\hat{f}(k)/|k|^2)$$

donde $\hat{f}(k)$ es la transformada de Fourier (Fourier Transform, FT) de f y $\hat{\phi}^{-1}$ es la transformada de Fourier inversa (Inverse Fourier Transform, IFT) de ϕ.

Esta ecuación se traduce en un algoritmo de tres pasos que resuelve la ecuación de Poisson:

1. calcular la FFT de la función f
2. resolver la ecuación en el espacio frecuencial dividiendo cada punto por el cuadrado del módulo de sus coordenadas, y
3. calcular la IFT del resultado

Recordemos que el algoritmo se completa mediante una etapa de normalización requerida por muchas implementaciones de la FFT y que se calcula dividiendo cada punto por N.

2.3.4. Transposiciones

Las transposiciones reordenan los datos en memoria y frecuentemente se utilizan para optimizar los accesos a memoria y los patrones de comunicación de los programas paralelos [82–85]. Son una parte muy importante en la construcción de numerosos algoritmos numéricos como el cálculo multidimensional de la FFT. También son útiles, entre otras, en aplicaciones de imágenes por radar [86] o para mejorar el acceso a memoria en general [87, 88]. Vale la pena destacar que el cálculo de la FFT es uno de esos algoritmos que hacen un uso extensivo de las transposiciones [17].

Debido a que las transposiciones de matrices son esencialmente un problema de acceso a memoria, el rendimiento de éstas está condicionado por el ancho de banda de la arquitectura en la que se estén realizando. Esta característica hace que las GPU sean una plataforma más adecuada que las CPU

para ejecutar transposiciones debido a su gran ancho de banda de acceso a memoria en comparación con las CPU. La implementación de transposiciones eficientes en una GPU que consigan una gran parte del ancho de banda del pico de acceso a memoria ha sido estudiado en diferentes trabajos [89, 90]. Sin embargo, aunque son más rápidas, la capacidad de memoria en las GPU es más limitada que en las CPU. Si la matriz transpuesta no se puede almacenar en el mismo espacio de memoria que la matriz original (*in-place*), únicamente se podrá utilizar un 50 % de la memoria total de la GPU, ya que este tipo de transposiciones (*out-of-place*) necesitan aproximadamente el doble de espacio que lo que ocupan los datos de entrada.

Para realizar la transposición de una matriz de 2D sólo existe una única opción, en la que, sea A una matriz de 2D con x filas e y columnas y sea A^t la matriz transpuesta. Tenemos que $(A^t)_{xy} = A_{yx},\ 1 \leq x \leq n,\ 1 \leq y \leq m$, donde el elemento a_{yx} de la matriz original A se convertirá en el elemento a_{xy} de la matriz transpuesta A^t.

En el caso de una matriz en 3D, la transposición se define como la permutación de los ejes x, y y z. Por lo tanto, existen 6 permutaciones posibles para transponer el contenido de una matriz en 3D. Una de esas permutaciones es trivial, ya que simplemente copia la matriz de origen a la matriz de destino sin alterar su contenido. Para el resto de las 5 permutaciones, es necesario definir una notación que posteriormente será utilizada en el presente trabajo de investigación. Sea T la transposición de una matriz 3D, y su subíndice la permutación que se le aplica, tenemos que el resultado de la transposición para cada permutación de la matriz de entrada en 3D $A[i, j, k]$ es: $T_{zxy}(A) = A[k, i, j]$, $T_{yzx}(A) = A[j, k, i]$, $T_{xzy}(A) = A[i, k, j]$, $T_{yxz}(A) = A[j, i, k]$ y $T_{zyx}(A) = A[k, j, i]$, donde $0 \leq i < n_x,\ 0 \leq j < n_y,\ 0 \leq k < n_z$.

Cabe reseñar que 3 permutaciones de las transposiciones dejan una de las dimensiones en su posición inicial —$\mathbf{x}zy$, $z\mathbf{y}x$ y $yx\mathbf{z}$— y por lo tanto, pueden ser interpretadas como transposiciones en 2D de una serie de planos. Además, esto implica que son involuciones, es decir, para estos casos se cumple que $T(T(A)) = A$. Obsérvese también que la transposición zyx permuta entre los datos almacenados por columnas a datos ordenados por filas. Por último, el resto de las permutaciones no triviales, yzx y zxy son rotaciones, esto significa que para estos casos $T(T(T(A))) = A$. Del mismo modo, también se cumple que $T_{yzx}(T_{zxy}(A)) = A$ y $T_{yzx}(T_{yzx}(A)) = T_{zxy}(A)$.

2.4. Estado del arte

Una vez introducido el ámbito tecnológico y el ámbito matemático del trabajo, estamos en disposición de mencionar el estado del arte actual.

2.4.1. La transformada rápida de Fourier

Desde la década de los 60 el estudio de la FFT es un aspecto clave para muchos campos de la ciencia y de la ingeniería [91–95]: tratamiento de imagen en la medicina, procesamiento digital de la señal, óptica, astronomía, geología, comunicaciones, cálculo de estructuras electrónicas, etc.

En el apartado 2.3.3 se ha justificado que para la resolución eficiente de la ecuación de Poisson se puede utilizar un método, basado en la familia de algoritmos de la FFT, que ha demostrado una buena capacidad de escalado en clústeres con gran cantidad de nodos [5].

Existen diferentes librerías para el cálculo de la FFT en GPU de NVIDIA: NukadaFFT [14, 96, 97], GPUFFTW [98], clFFT [99], OpenCLFFT [48] y cuFFT [77]. La librería mas utilizada es la librería que proporciona el propio fabricante NVIDIA (cuFFT), que ha desarrollado una librería integrada en el entorno de programación CUDA que permite, mediante una interfaz simple, el cálculo eficiente de la FFT en GPU de NVIDIA.

2.4.2. Transposiciones

El mecanismo de las transposiciones ha sido estudiado desde hace décadas [100, 101]. Entre los estudios más recurrentes se encuentran aquellos que se centran en la utilización eficiente de la memoria caché [84, 85, 102]. Otros estudios se centran en optimizar el rendimiento de las transposiciones minimizando el consumo de memoria necesario en el proceso. Para ello se realizan transposiciones *in-place* [82, 83], donde el resultado del cálculo es almacenado en el mismo espacio de memoria donde estaban los datos de entrada. Estas transposiciones *in-place* principalmente han utilizado algoritmos iterativos con un nivel limitado de paralelismo [103], sobre todo debido a problemas de balanceo de carga. Trabajos más recientes proponen algoritmos

más paralelizables capaces de aprovechar el grado de paralelismo ofrecido por las nuevas arquitecturas [85, 87, 104, 105]. Si no se requiere optimizar el consumo de la memoria, se realiza lo que se denomina como una transposición *out-of-place* [89, 106].

Por último, la transposición, cuyo rendimiento depende directamente de la velocidad de acceso a memoria, es un candidato ideal para aprovechar el gran ancho de memoria que ofrecen las GPU [87, 89, 105].

2.4.3. La ecuación de Poisson

En la bibliografía se pueden encontrar múltiples propuestas que implementan la resolución de la ecuación de Poisson utilizando CPU. En el trabajo de Garcia-Risueño et al. [5] se puede ver una comparativa de los principales métodos: FFT, interpolación de funciones de escalado (Interpolating Scaling Functions, ISF), método multipolar rápido (Fast Multipole Method, FMM), gradiante conjugado (GC) y multigrid (MG). Sin embargo, como se ha mencionado, en este trabajo nos vamos a centrar en las implementaciones para GPU.

Aunque el uso de las GPU para uso científico es una herramienta relativamente reciente, existen diversas soluciones capaces de resolver la ecuación de Poisson en GPU. En 2003, Krüger y Westermann propusieron una manera de resolver la ecuación de Poisson mediante el uso del método de GC en una implementación para GPU [30]. Su trabajo se centraba en la definición de estructuras básicas que pudieran ser utilizadas en el diseño de técnicas más genéricas para el cómputo numérico. Éstas difieren de las hasta entonces conocidas en que estas últimas se centraban en proveer una solución particular a cada problema. En definitiva, sacrificaban rendimiento en el cálculo con el objeto de conseguir más flexibilidad en las implementaciones. No obstante, su algoritmo sólo permitía resolver problemas numéricos de 2D y en los que no fuera necesario realizar cálculos en coma flotante. Este método volvió a ser utilizado posteriormente por otros autores para realizar nuevas implementaciones para GPU en diversos entornos. Por ejemplo, algunos trabajos proponen la resolución del método de GC en un único nodo con múltiples GPU [34], en los que el cálculo se reparte entre la CPU y la GPU mientras que se solapan los procesos de cálculo y comunicación. Otros trabajos proponen un método para resolver el GC en un único nodo con múltiples GPU mediante el uso de un modelo heurístico al que se le aplica una transformación previa que facilita

su posterior cálculo numérico (llamada preacondicionamiento) adecuada en el contexto de sistemas de procesamiento de SIMD [35]. Mientras que otros trabajos hacen uso de múltiples nodos con múltiples GPU para resolver el GC preacondicionado mediante el método de Jacobi [36].

El método MG también ha sido utilizado para resolver la ecuación de Poisson en GPU. En este escenario algunos trabajos usan las GPU como precondicionadores MG eficientes que sustituyen a las soluciones equivalentes en CPU [31]. Otros autores proponen una solución heterogénea (CPU-GPU) mediante el método MG, en la que se simultanean operaciones de cálculo [32].

Otra forma de solucionar la ecuación de Poisson consiste en utilizar el método iterativo de Jacobi [37], donde se solapan las comunicaciones de la MPI y las de la GPU, con las propias operaciones de cálculo en la GPU. Otros autores también proponen calcular la ecuación de Poisson mediante el método iterativo de Jacobi, aunque sólo utilizan un nodo con múltiples GPU [33]. No obstante, los propios autores reconocen que para conseguir soluciones más eficientes es necesaria la utilización de otros métodos, por ejemplo, el método MG.

Todos estos trabajos afrontan la resolución de la ecuación de Poisson en diferentes escenarios. Por ejemplo, cuando el tamaño de los datos de entrada superan el tamaño de la memoria física de la GPU, es necesario distribuir los datos en segmentos que no sobrepasen el tamaño total de la memoria física de la GPU. Esta segmentación implícitamente supone un mayor número de comunicaciones, que usualmente es la parte que más coste conlleva en este tipo de cálculos. Diversos trabajos han estudiado esta problemática. Algunos proponen un algoritmo para calcular la FFT en un sistema con la memoria jerárquica (no local) [17]. Sin embargo, no se introduce ningún mecanismo que permita solapar la comunicación con los cálculos. Trabajos posteriores afrontan el cálculo de la FFT de tamaño superior a la memoria disponible en una GPU utilizando una arquitectura basada en un clúster [18, 19]. Una de estas soluciones permite resolver la FFT 3D en un clúster con 16 nodos GPU [10]. Sin embargo, ninguno de estos trabajos aborda un estudio en profundidad del bus de datos PCI que permita conseguir una solución óptima y que ofrezca un ancho de banda eficiente en la fase de comunicación. En este sentido, algunos autores estudian el rendimiento del bus PCI durante la transferencia de datos en el cálculo de la FFT y proponen un algoritmo de bloqueo para aumentar su ancho de banda efectivo [12]. Los autores de este último trabajo también proponen varias descomposiciones del algoritmo para calcular la FFT con el fin de aumentar la localidad de los datos en memoria,

y de este modo, mejorar la eficiencia del bus PCI.

Si nos centramos en las soluciones para la ecuación de Poisson basadas en FFT, tenemos a autores de referencia como Wu y Jaja, quienes han publicado una serie de trabajos en este área [16, 38–40].

Uno de sus primeros trabajos consiste en calcular FFTs optimizadas para las situaciones en los que los datos en 3D a procesar no superan la memoria física de la GPU [16]. En este trabajo se emplean aceleradores gráficos con las arquitecturas Tesla y Fermi de NVIDIA, donde se aprovecha el alto grado de paralelismo que ofrece esta tecnología con el objeto de poder administrar la memoria del dispositvo de manera óptima, de tal manera que todos los accesos a memoria global se realizan en bloques de 128 bytes de manera coalescente; de este modo se consiguen mitigar los efectos relacionados con el *partition camping* [89], la localidad [107] y la asociatividad. A la vez que todos los cálculos se realizan en los registros del dispositivo, en particular, el número de accesos a la memoria global se reduce al mínimo cuando el conjunto de datos en 3D y los cálculos de la FFT para la dimensión X se solapan, prácticamente en su totalidad, con las transferencias de datos desde la memoria principal para el resto de las dimensiones (Y,Z).

Estos mismos autores en un trabajo posterior, resuelven la ecuación de Poisson mediante el uso de FFT para los casos en los que los datos superan la memoria del dispositivo pero aún no superan la memoria de la CPU [38]. En particular, el algoritmo propuesto incluye sendas implementaciones para las FFT en 2D y 3D y para su resolución hace uso de matrices tridiagonales donde tanto los datos de entrada como los de salida no superan la memoria principal de la CPU. Las principales contribuciones de este trabajo son que los cálculos se organizan de tal manera que la matriz de datos en 3D sólo se transmite una vez entre el dispositivo y la CPU, a la vez que los cálculos en la GPU se solapan en su totalidad con las transferencias de datos por el bus PCI.

En un tercer trabajo vuelven a profundizar en la resolución de la ecuación de Poisson mediante el uso de FFT sobre GPU, aplicable a problemas con condiciones de contorno periódicas en las tres dimensiones o aplicable a problemas con condiciones de contorno periódicas en dos dimensiones y con condición de contorno del tipo *Neumann* para la tercera dimensión [39].

Finalmente, en su último trabajo, enfocado a la resolución de la ecuación de Poisson mediante el uso de FFT sobre GPU, integran todas las optimizaciones

conseguidas en los trabajos anteriores [40]. En este último trabajo definen una estrategia optimizada para descomponer FFT multidimensionales de tamaño superior al tamaño de la memoria de la GPU en plataformas híbridas CPU-GPU, que solapa casi en su totalidad las ejecuciones de las operaciones que se realizan en la GPU, con la transferencia de los datos mediante el bus PCI. Además, los datos de la matriz de entrada únicamente son transferidos una única vez entre la CPU y la GPU y los resultados demuestran que el esquema propuesto es igualmente eficaz para los cálculos en precisión simple como para los realizados en precisión doble.

En cuanto a trabajos específicos sobre el cálculo de la FFT en un clúster de GPU la mayoría se centran en optimizar uno de los pasos más críticos –la comunicación inter-nodo– mediante el uso de primitivas de IB (IB Verbs) en vez de la implementación estándar de MPI [10, 15]. Estas primitivas facilitan optimizaciones a muy bajo nivel, lo que permite reducir la latencia de comunicación a coste de perder la portabilidad del código obtenido. Entre los trabajos más destacados se encuentran el realizado por Chen et al., que para maximizar el ancho de banda *duplex* de la red IB han desarrollado una interfaz que directamente invoca primitivas de IB para la comunicación RDMA [10]. O el realizado por Nukada et al., donde han desarrollado un nuevo *pipeline* para optimizar la comunicación *Alltoall* entre las GPU mediante la utilización de primitivas de la API de IB y de la API de CUDA. Este desarrollo permite minimizar la sobrecarga en la comunicación intra-nodo e inter-nodo y gestionar directamente los recursos disponibles para asignarlos de manera eficiente [15].

Parte II

APORTACIONES

Capítulo 3

Análisis del rendimiento de la librería cuFFT

Índice

3.1. Introducción

En el capítulo 2 (apartado 2.3.3) se ha justificado que para la resolución eficiente de la ecuación de Poisson se puede utilizar un método basado en la familia de algoritmos de la FFT. En ese mismo capítulo, se expone que en el presente trabajo de investigación la tecnología que se va a utilizar está basada en las GPU de NVIDIA, y que para esta tecnología, la librería mas utilizada es la librería que proporciona el propio fabricante NVIDIA –cuFFT–, ver capítulo 2 (apartado 2.4.1).

El uso de esta librería en este trabajo de investigación ha requerido realizar un análisis de los diferentes parámetros y modos de funcionamiento

disponibles, de cara a identificar cuál es la configuración más adecuada para conseguir cálculos de FFT óptimos. Es en este capítulo donde se analiza el comportamiento de esta librería, en cuanto a tiempo de ejecución y uso de memoria, para el cálculo de la FFT con diversas configuraciones.

3.2. Librería cuFFT

La librería cuFFT sigue el modelo de la conocida librería FFTW [7], una de las librerías FFT más extendidas y eficientes que se han implementado para ser ejecutada en una CPU. En particular, la librería cuFFT implementa algoritmos basados en los ampliamente conocidos de Cooley-Tukey [6] y Bluestein [72].

La librería cuFFT dispone de dos parámetros principales para su configuración. Uno de esos parámetros permite realizar FFT sobre datos no contiguos (a la cual llamaremos ***stride***) y el otro parámetro permite realizar varias FFT en una única llamada a las funciones de la librería (a la cual llamaremos ***batch***). Son precisamente estos dos parámetros los que permiten realizar descomposiciones como las analizadas en el capítulo 2 (apartado 2.3.2), ya que permiten calcular FFT en la dimensión que se desee (***stride***) y tantas veces como se necesite (***batch***).

Como para la consecución de los objetivos de este trabajo se ha tenido que descomponer la FFT, se ha hecho un uso masivo de este tipo de operaciones. Para intentar optimizar estos cálculos, se ha realizado un análisis de estas dos parámetros que ofrece la librería cuFFT.

Tal y como ya se ha comentado, la librería cuFFT permite calcular FFT usando matrices de entrada con datos no contiguos [10]. En realidad, la librería es muy flexible en este aspecto, ya que permite la formación de complicadas ordenaciones muy diversas de los datos de entrada y salida.

El principal uso que se le va a dar a la librería es la de calcular FFT, y en este caso, la librería cuFFT dispone de una característica destacable, y que tal y como se verá más adelante en este capítulo, permite la realización de múltiples transformadas llamando a una única función. Mediante esta característica se puede conseguir un rendimiento superior al que se alcanzaría llamando repetidas veces a una función que realiza el cálculo de una única

FFT.

No obstante, debido al carácter de código cerrado de la librería, como desarrolladores no podemos conocer las implicaciones que tienen las diferentes configuraciones en el rendimiento de la librería. Y por este motivo, en el presente capítulo se analizan las consecuencias de las diferentes configuraciones de la librería cuFFT –tanto en consumo de memoria como en tiempo de ejecución–, y así poder elegir aquella configuración que mejor se adecue a nuestras necesidades para calcular la FFT de manera eficiente.

La librería cuFFT ofrece un mecanismo de configuración muy sencillo, llamado ***plan***, que mediante el uso de bloques internos optimiza los cálculos a realizar para cada configuración y GPU utilizada. Una vez el ***plan*** ha sido definido, los cálculos posteriores se realizan siguiendo su configuración. Una de las ventajas más significativas de este modelo de trabajo consiste en que una vez creado el ***plan***, la librería conserva todos los recursos necesarios para que se pueda ejecutar tantas veces como sea necesario sin tener que realizar, de nuevo, las configuraciones ni repetir cálculos de configuración. Esta manera de trabajar realmente da buenos resultados con la librería cuFFT, ya que cada cálculo de una FFT requiere de una configuración y de recursos de GPU diferentes, y la interfaz que ofrece la función ***plan*** facilita la manera en la que estas configuraciones pueden ser reutilizadas. Sin embargo, en estudios preliminares hemos detectado que en ciertos casos la creación de un ***plan*** consume un espacio de memoria que puede ser considerable. Precisamente por este motivo, uno de los objetivos de este capítulo es cuantificar este consumo de memoria, junto con su tiempo de ejecución.

Para definir un ***plan*** se ha utilizado la función ***cufftPlanMany()*** mientras que para ejecutar los ***planes*** se utiliza la familia de funciones ***cufftExec****.

Mediante la función ***cufftPlanMany()*** (véase la tabla 3.1) se crea la configuración de un ***plan*** FFT de dimensión ***rank*** (1D, 2D o 3D), cuyo tamaño se especifica en el vector ***n***. El parámetro ***type*** indica el tipo de datos con el que se va a trabajar (véase la tabla 3.2) y el parámetro ***batch*** indica cuántas transformaciones han de realizarse al ejecutar el ***plan***.

Esta función permite estructuras de datos de diversa complejidad mediante los parámetros que configuran la opción ***stride***: ***inembed, istride, idist, onembed, ostride*** y ***odist***. Si ***inembed*** u ***onembed*** tienen el valor de ***NULL*** toda la información relativa al ***stride*** es ignorada, y se utilizan los

valores por defecto (los datos de los arrays de entrada son contiguos).

plan	Parámetro de salida con un puntero a un objeto ***cufftHandle***
rank	Dimensión de la transformación (1, 2 o 3)
n	Vector de tamaño rank que indica el tamaño de cada dimensión
inembed	Puntero de tamaño rank que indica las dimensiones de los datos de entrada en memoria. Cuando tiene el valor ***NULL*** se ignoran los parámetros sobre la disposición de los datos
istride	Indica la distancia entre dos elementos de entrada sucesivos en la dimensión menos significativa (es decir, la más interna)
idist	Indica la distancia entre el primer elemento de dos señales consecutivas del batch en los datos de entrada
onembed	Puntero de tamaño rank que indica las dimensiones de los datos de salida en memoria. Cuando tiene el valor ***NULL*** se ignoran los parámetros sobre la disposición de los datos
ostride	Indica la distancia entre dos elementos de salida sucesivos en la dimensión menos significativa (es decir, la más interna)
odist	Indica la distancia entre el primer elemento de dos señales consecutivas del batch en los datos de salida
type	Tipo de los datos a transformar (ver tabla 3.2)
batch	Número de transformaciones a realizar

Tabla 3.1: Parámetros de la función ***cufftPlanMany()***

CUFFT_R2C	Real a complejo (precisión simple)
CUFFT_C2R	Complejo a real (precisión simple)
CUFFT_C2C	Complejo a complejo (precisión simple)
CUFFT_D2Z	Real a complejo (precisión doble)
CUFFT_Z2D	Complejo a real (precisión doble)
CUFFT_Z2Z	Complejo a complejo (precisión doble)

Tabla 3.2: Tipos de datos de la librería cuFFT

Todas las funciones de la librería cuFFT retornan un valor que sirve para

identificar si la operación ha finalizado satisfactoriamente:
(*CUFFT_SUCCESS*), o si ha habido algún error:
(*CUFFT_ALLOC_FAILED*, *CUFFT_INVALID_VALUE*...).

Una vez se ha creado el ***plan***, se obtiene un identificador en el parámetro ***plan*** (véase la tabla 3.1) que permite ejecutar el ***plan*** deseado tantas veces como sea necesario mediante una función de la familia ***cufftExec****.

Las funciones para ejecutar un ***plan*** se agrupan dentro de la familia de funciones ***cufftExec**** y sirven para indicar los tipos de datos utilizados (ver tabla 3.2). El listado de funciones dentro de esta familia es el siguiente:

- *cufftExecC2C*
- *cufftExecZ2Z*
- *cufftExecR2C*
- *cufftExecD2Z*
- *cufftExecC2R*
- *cufftExecZ2D*

Todas estas funciones utilizan un conjunto de parámetros de entrada (véase tabla 3.3).

plan	Identificador del ***plan***
idata	Puntero a los datos de entrada a transformar de la memoria de la GPU
odata	Parámetro de salida: puntero a los datos de salida a transformar de la memoria de la GPU
direction	Dirección de la transformación –directa o inversa– (sólo para las funciones ***cufftExecC2C()*** y ***cufftExecZ2Z()*** ya que en el resto está implícito)

Tabla 3.3: Parámetros de entrada para la familia de funciones ***cufftExec****

Por ejemplo, la función *cufftExecD2Z()* ejecuta la transformación de datos reales a complejos en precisión doble según la configuración del ***plan***. Si los punteros de los datos de entrada y salida coinciden, estas funciones realizan lo que se conoce como una transformación *in-place*, es decir, el resultado

del cálculo es almacenado en el mismo espacio de memoria donde estaba la entrada. Si no, se realiza lo que se denomina como una transformación *out-of-place*.

3.3. Metodología experimental

La experimentación se ha realizado en un nodo con una GPU Tesla M2090 de NVIDIA. Esta GPU está basada en la arquitectura ***Fermi*** de CUDA. Dispone de 512 núcleos CUDA que ofrecen un rendimiento máximo de 665 Gigaflop/s para datos en coma flotante de doble precisión. Para datos en coma flotante de precisión simple el rendimiento supera el Teraflop/s. Dispone de una memoria de acceso aleatorio (Random Access Memory, DRAM) de 6 GB DDR5 con un ancho de banda teórico de 177 GB/s. Toda la experimentación ha sido realizada usando la versión 6.5 de CUDA y la versión 340.29 del driver de CUDA.

En la experimentación, por cada prueba realizada se ha medido tanto el rendimiento obtenido –tiempo de ejecución por FFT– como el consumo de memoria en la GPU asociado a la generación del plan. Los experimentos han sido planificados de tal manera que sea sencillo observar cómo afectan los diferentes valores de los parámetros de configuración de un ***plan*** a las variables que se quieren medir en la ejecución de las FFT. Todos los experimentos han sido repetidos para realizar transformaciones de real a complejo, de complejo a real y de complejo a complejo –directa o inversa– y los cálculos se han realizado utilizando números de precisión doble. El tiempo de ejecución ha sido calculado mediante eventos de CUDA como el promedio de 100 ejecuciones tras 10 ejecuciones de calentamiento, y no se han tenido en cuenta los tiempos de copia para los datos entre la CPU y GPU. El consumo de memoria ha sido calculado preguntando al sistema la memoria libre disponible antes y después de la creación de un ***plan***. Para este fin se ha utilizado la función *cudaMemGetInfo()* proporcionada por CUDA.

En pruebas preliminares se pudo observar que, para ambos casos (*in-place* y *out-of-place*), el tiempo de ejecución era similar, y por lo tanto, los resultados que se van a presentar en este capítulo corresponden a la experimentación realizada en el cálculo de la FFT*out-of-place*.

La experimentación se ha realizado con distintos tamaños de lado (N)

y con varias dimensiones (D) para los datos de entrada. Los lados siempre son del mismo tamaño (cuadrado o cubo) y el tamaño de esos lados varía de 32 a 1024, tal y como se describe en la tabla 3.4. Para los tamaños más grandes, no se ha podido realizar la experimentación ya que los datos no entraban en la memoria de la GPU. Para los tamaños más pequeños, aún habiendo realizado la experimentación, no se van a presentar gráficas con los resultados obtenidos debido a que no resultan significativos. Nótese que entre los tamaños de los datos de entrada existen algunas equivalencias (en lo que a tamaño de memoria se refiere) para dimensiones distintas. Por ejemplo, una FFT de 1024 elementos y de 1D equivale a una FFT de 32 elementos y 2D.

D/N	32	64	128	256	512	1024
1D	256B	512B	1KB	2KB	4KB	8KB
2D	8KB	32KB	128KB	512KB	2MB	8MB
3D	256KB	2MB	16MB	128MB	1GB	*

Tabla 3.4: Tamaño de los datos de entrada usados en la experimentación. N es el número de elementos por lado y D el número de dimensiones.
** Los datos no entran en la memoria de la GPU.*

Tal y como se ha comentado al inicio del presente capítulo, el estudio de la librería cuFFT se centra principalmente en dos opciones: ***batch*** y ***stride***. Para analizar la influencia que cada una de estas opciones tienen en el cálculo de la FFT, por una parte, a la opción ***batch*** se le han ido asignando valores entre 1 a 1 000 000 con diferentes saltos: entre 1 y 99 el valor va aumentando de uno en uno, entre 100 y 900 el aumento es de 10^2, entre 1 000 y 9 000 el aumento es de 10^3, entre 10 000 y 90 000 el aumento es de 10^4 y, por último, el aumento es de 10^5 entre 100 000 y 1 000 000. Por otra parte, para la opción ***stride*** esta experimentación se ha centrado en analizar una de las operaciones más frecuentes relativa a la implementación de algoritmos FFT paralelos: descomponer una FFT de tres dimensiones en múltiples FFT de una dimensión. En este contexto, la descomposición de una FFT de 3D de tamaño N^3 implica la utilización de tres FFT de 1D: una con los datos contiguos, otra con un ***stride*** de N y la última con un ***stride*** de N^2. Por lo tanto, esta parte de la experimentación se ha centrado en analizar el cálculo de la FFT para datos no contiguos y con un ***stride*** de N y N^2.

3.4. Resultados

Se van a presentar los resultados correspondientes a transformaciones sobre diferentes tipos de datos (D2Z, Z2D y Z2Z –directa e inversa–). En el análisis de los resultados, primero, se van a presentar los resultados para los datos contiguos, y a continuación, se van a presentar los resultados para los datos no contiguos, tanto para la opción ***batch***, como para diferentes valores de los lados de los datos de entrada (N) y dimensiones utilizadas (D).

En la figura 3.1 se puede observar que los resultados obtenidos para las diferentes transformaciones de los datos (D2Z, Z2D y Z2Z) son muy parejos, y por lo tanto, a partir de este punto y mientras no se indique lo contrario, los resultados que se presentan en este capítulo corresponden al cálculo de la FFT para transformaciones de real a complejo (D2Z).

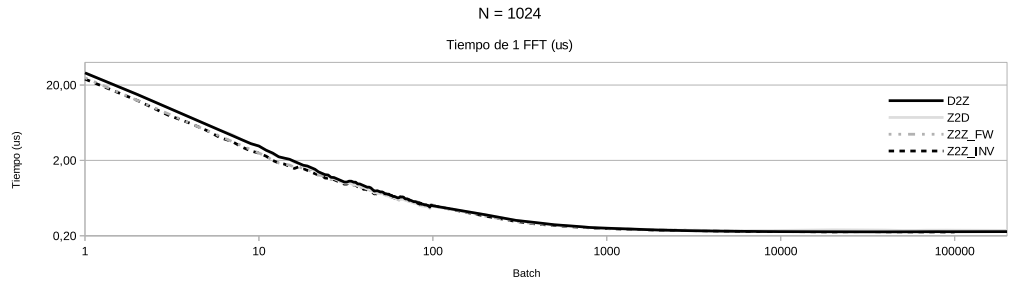

Figura 3.1: Tiempo de ejecución promedio de una FFT de 1D y 1024 elementos dependiendo del número de FFT a ejecutar en ***batch***.

Resultados para opción *batch* (datos contiguos)

En la experimentación se ha podido observar que en la creación de un plan de la librería cuFFT se consume una memoria equivalente a los datos procesados –el producto de todas las dimensiones multiplicado por el tamaño del tipo de datos– multiplicado por el número de FFT a ejecutar en ***batch***. Es decir, el consumo de memoria se puede expresar de la siguiente manera: $N \cdot S \cdot B$, donde N corresponde al número de elementos de elementos de la FFT, S al tamaño de los tipos de datos y B al número de transformaciones a realizar en ***batch***. Esta fórmula permite predecir si la ejecución de la FFT entra en la memoria física de la GPU o no. También se observa que la primera vez que se crea un ***plan***, se necesita una memoria extra de 11MB que no

vuelve a ser requerida en ***planes*** posteriores.

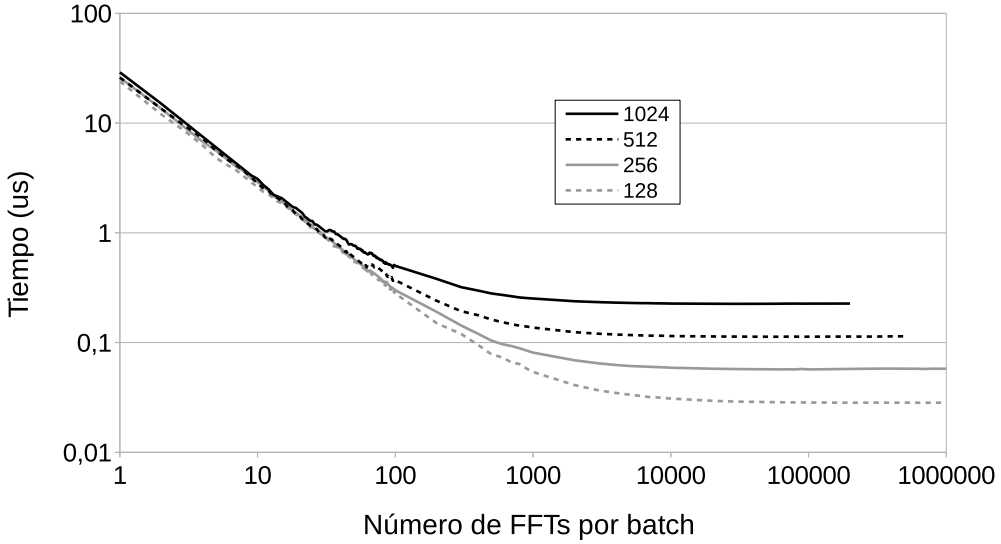

Figura 3.2: Tiempo de ejecución promedio de cada FFT 1D (D2Z) en función del número de FFT que se ejecutan en ***batch***.

La figura 3.2 muestra el tiempo de ejecución de una FFT de 1D de real a complejo con tamaños de datos de entrada de 128, 256, 512 y 1024 elementos. En esta figura se puede observar cómo decrece el tiempo de ejecución por FFT según aumenta el valor del parámetro ***batch***. Este comportamiento se mantiene hasta que el parámetro ***batch*** alcanza un valor específico. A partir de este punto el tiempo de ejecución por FFT se mantiene constante. Para valores pequeños del parámetro ***batch***, la curva sigue una función de potencia –ambos ejes de la gráfica son logarítmicos– y el tiempo de ejecución es muy similar para cada tamaño de FFT. Sin embargo, para tamaños grandes del parámetro ***batch***, la curva prácticamente se convierte en una constante y la influencia del tamaño de la FFT es significativa. Entre estas tendencias, hay una zona de transición donde la curva suavemente pasa de una recta descendente a una línea horizontal. Esta transición comienza antes para tamaños de FFT más grandes.

En la figura 3.3 se puede observar como las FFT de 2D siguen el mismo patrón que las FFT de 1D. A medida que el tamaño de las FFT aumenta, el área de la función de potencia se reduce y, por lo tanto, el área de transición comienza antes. En realidad, la curva correspondiente a la FFT de tamaño 256^2 comienza en la zona de transición.

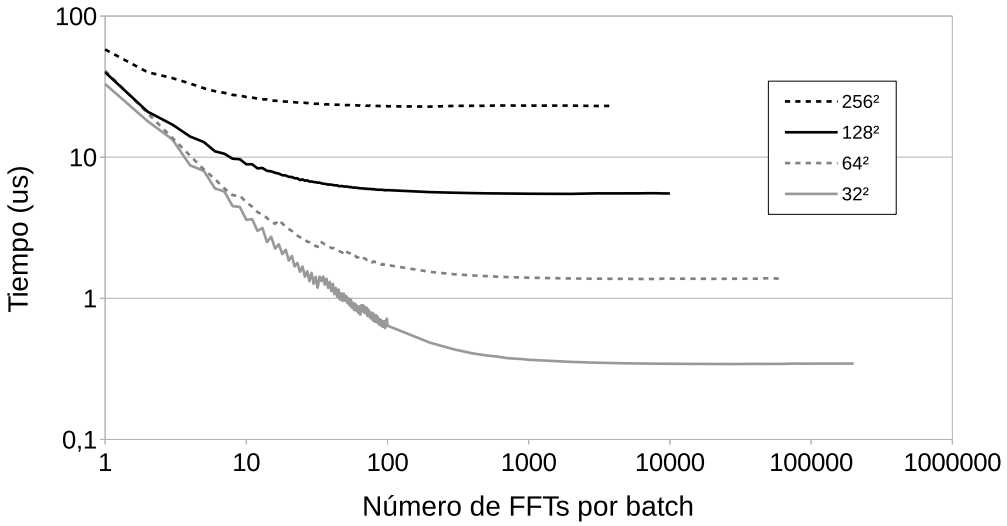

Figura 3.3: Tiempo de ejecución promedio de cada FFT 2D (D2Z) en función del número de FFT que se ejecutan en ***batch***.

En la figura 3.4 se observa que el mismo patrón se repite para las FFT de 3D, donde el tamaño más grande de las FFT calculadas hace que la gráfica comience prácticamente en los valores constantes. En realidad, la dimensionalidad no afecta realmente al comportamiento de la FFT, sin embargo, sí que lo hace el tamaño de los datos de entrada (N).

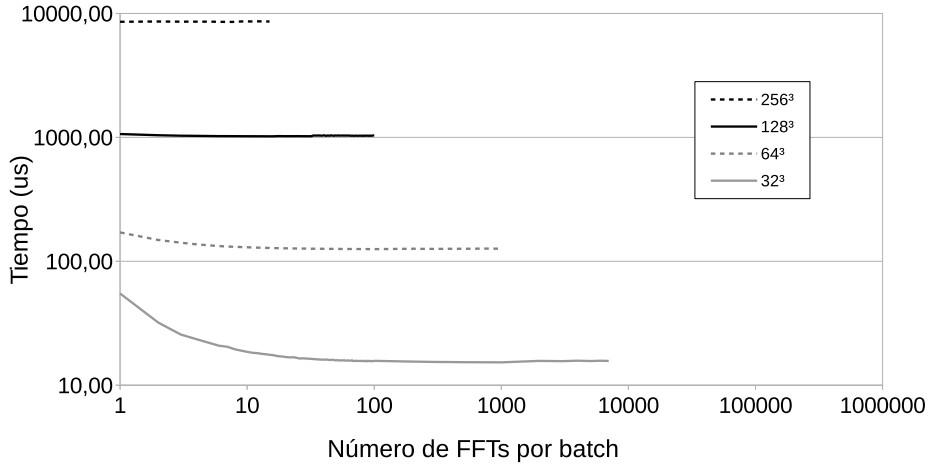

Figura 3.4: Tiempo de ejecución promedio de cada FFT 3D (D2Z) en función del número de FFT que se ejecutan en ***batch***.

Todos los resultados que se han presentado sugieren que cuando el parámetro ***batch*** y el tamaño de la FFT no son lo suficientemente grandes, el hardware de la GPU está infrautilizado, y por lo tanto, la ejecución de más FFT o FFT de mayor tamaño se realiza prácticamente sin coste añadido. A partir de cierto punto, el hardware de la GPU se utiliza de manera más eficiente y la pendiente de la curva se va reduciendo. Nuestra estimación es que esta reducción es perceptible una vez que el tamaño de los datos multiplicado por el parámetro ***batch*** alcanza aproximadamente un tamaño de 256 KB. Por último, para valores altos del parámetro ***batch*** o para tamaños grandes de la FFT, la biblioteca cuFFT es capaz de utilizar completamente los recursos de la GPU y se consigue el menor tiempo de ejecución por FFT. En este punto, incrementar el valor del parámetro ***batch*** no supone una mejora en el tiempo de ejecución por FFT, aunque implica un mayor consumo de memoria. Por lo tanto, una vez alcanzada esta situación y con el fin de ahorrar memoria en la GPU para realizar otras operaciones, es más eficiente realizar el cálculo de las FFT restantes volviendo a ejecutar el mismo ***plan*** en vez de incrementar el valor del parámetro ***batch***. En la experimentación que se ha realizado este punto aproximadamente se alcanza cuando el consumo de memoria se encuentra entre 25MB y 50MB.

Resultados para opción *stride* (datos no contiguos)

A continuación se analiza el impacto que supone utilizar la opción ***stride*** para el cálculo de la FFT, tanto en consumo de memoria como en tiempo de ejecución.

El consumo de memoria se puede calcular mediante el mismo método que se utilizó al calcular FFT con datos contiguos. A pesar del carácter no contiguo de los datos, la realidad es que la librería cuFFT parece que es capaz de compactar la memoria de manera eficiente y no consumir más memoria que cuando los datos son contiguos.

La figura 3.5 muestra el tiempo de ejecución obtenido tras la ejecución de varias FFT de 1D con tamaños de 256 y 512 elementos y un ***stride*** de N^2. No se ha podido probar con un tamaño de 1024, ya que debido al ***stride*** los datos no entran en la memoria. Los resultados están normalizados con el correspondiente tiempo de ejecución de la misma FFT con datos contiguos. De este modo, se puede diferenciar fácilmente la penalización en la que incurre la ejecución de FFT con datos no contiguos. También se ha realizado la misma

prueba para un ***stride*** de N y los resultados han sido prácticamente idénticos.

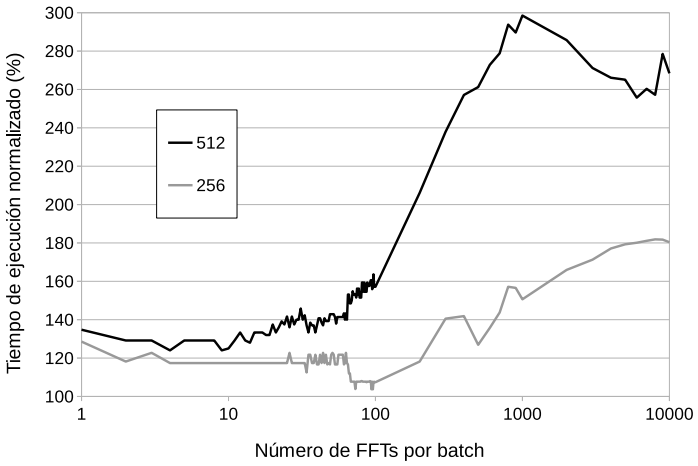

Figura 3.5: Tiempo de ejecución para varias FFT de 1D con transformación de datos de complejo a complejo y datos no contiguos con un ***stride*** de N^2, normalizado con el correspondiente tiempo de ejecución de la misma FFT con datos contiguos.

En la figura 3.5 se puede observar, tal y como era predecible, que el rendimiento obtenido en el cálculo de FFT con datos no contiguos es siempre inferior al rendimiento obtenido en el cálculo de la misma FFT con datos contiguos. Para valores pequeños del parámetro ***batch*** –aproximadamente durante la fase en la que consideramos que la GPU está infrautilizada (valores inferiores a 100)– la penalización oscila entre los valores del 20 % y del 40 %. No obstante, esta penalización aumenta sustancialmente para valores más altos del parámetro ***batch***. Por ejemplo, para FFT de tamaño de 256 elementos se alcanza una penalización del 80 % para valores altos del parámetro ***batch***. Incluso estas penalizaciones son superiores en los casos en los que se calculan FFT de tamaño de 512 elementos, llegando hasta a triplicar el tiempo de ejecución de la misma FFT con datos contiguos. Por este motivo, es recomendable intentar utilizar, en la medida de lo posible, datos contiguos, y el uso de transposiciones puede ofrecer una solución para ordenar los datos de manera adecuada antes de calcular las FFT.

3.5. Conclusiones

En este capítulo se ha analizado el rendimiento que ofrece la librería cuFFT de NVIDIA, tanto en consumo de memoria como en tiempo de ejecución, para dos de las principales opciones que ofrece: la opción ***batch*** que permite calcular varias FFT con la llamada a una única función y la opción ***stride*** que permite calcular la FFT con matrices de entrada y salida con datos no contiguos. En la experimentación se han realizado transformaciones sobre diferentes tipos de datos (D2Z, Z2D y Z2Z –directa e inversa–) tanto para datos contiguos como para datos no contiguos y para diferentes valores de los lados de los datos de entrada (N) y dimensiones utilizadas (D).

El uso de la opción ***stride*** no afecta a la memoria requerida por la librería cuFFT, ya que la cantidad de memoria a usar es, en ambos casos, equivalente al tamaño de los datos a procesar. Sin embargo, la opción ***batch*** sí que afecta directamente a los requerimientos de memoria, ya que aumenta linealmente el consumo de la memoria por el número de FFT a calcular en un mismo ***plan***.

Respecto al tiempo de ejecución, la opción ***batch*** afecta de manera significativa al cálculo de las FFT de tamaño pequeño y medio. La razón principal de este impacto para FFT de este tamaño, radica en la infrautilización del hardware de la GPU. Así, el rendimiento obtenido es muy escaso comparado con el potencial teórico que ofrece. La ejecución de múltiples FFT al mismo tiempo permite a la librería cuFFT ocupar la mayor parte de los recursos de la GPU, y por lo tanto, se reduce el tiempo de ejecución para cada cálculo de la FFT. De hecho, se ha observado que el incrementar el número de FFT a calcular en ***batch*** prácticamente no incrementa el tiempo total de ejecución hasta que el tamaño de la FFT multiplicado por el número de transformadas a calcular en ***batch*** alcanza un umbral determinado que depende del tamaño de la FFT. A partir de este punto, se saturan los recursos que dispone la GPU y el tiempo de ejecución por FFT se estabiliza.

Estos resultados ofrecen dos conclusiones prácticas. En primer lugar, se puede calcular, de manera sencilla, los requerimientos de memoria necesarios para el cálculo de las FFT que se han planificado, y por lo tanto, garantizar si la memoria de la GPU que se va a utilizar dispone de suficiente espacio para almacenar todos los datos necesarios para realizar los cálculos. En el caso de que la cantidad de memoria disponible no sea suficiente, se puede reducir el número de FFT a calcular en ***batch*** hasta que los requerimientos de memoria estén acordes con el tamaño de memoria disponible. En segundo

lugar, se puede estimar el impacto que esta reducción tiene en el tiempo de ejecución de las FFT.

Ante el dilema de incrementar el número de transformadas a calcular en ***batch***, en vez de realizar repetidas llamadas a la librería cuFFT con valores más pequeños en la opción ***batch***, según la experimentación, siempre que la cantidad de memoria necesaria para la ejecución del ***plan*** no supere la memoria disponible en la GPU, es posible utilizar la primera opción con un rendimiento adecuado. Sin embargo, si el código desarrollado requiere limitar la cantidad de memoria necesaria para el cálculo de las transformadas, con el fin de disponer memoria libre en la GPU para otras tareas que puedan ser útiles, es posible limitar el consumo de memoria de la librería cuFFT alrededor de los 50 MB, ya que, a partir de esta cantidad de memoria, la opción ***batch*** no aporta beneficios en el cálculo de las transformadas.

Al analizar la penalización a la que se incurre al calcular FFT con datos no contiguos, los resultados muestran que esta penalización está condicionada por el tamaño de la FFT y con el número de FFT a calcular en ***batch***. La penalización más significativa que ha sido identificada en la experimentación –alcanzando incluso una penalización del 300 % respecto a su equivalente FFT con datos contiguos– se produce en el cálculo de FFT de 512 elementos y con aproximadamente 100 FFT a calcular en ***batch***.

Capítulo 4

Transposiciones eficientes en GPU

Índice

4.1. Introducción

En el capítulo 1 (sección 1.2) se ha comentado que la resolución de la ecuación de Poisson en un clúster de GPU implica la comunicación de datos entre diferentes GPU. En el capítulo 2 (apartado 2.3.3) se ha justificado que en esta comunicación los datos pueden estar distribuidos en memoria de diferentes maneras, y éstas no siempre tienen porqué ser la más adecuadas para conseguir la mayor eficiencia en la fase de comunicación. De cara a realizar la

comunicación de manera eficiente, es importante disponer de un mecanismo que permita modificar la distribución de los datos en la memoria. Debido a que poseen memorias que ofrecen mayor velocidad de acceso, se ha optado por realizar estas transposiciones en GPU. Por este motivo, el objetivo del presente capítulo consiste en presentar la librería que se ha desarrollado para realizar **transposiciones** 3D en GPU de manera eficiente y que posteriormente será utilizada –en el capítulo 5– para mejorar el rendimiento de la comunicación MPI interproceso, en concreto, en la fase *Alltoall* de la implementación desarrollada para la resolución de la ecuación de Poisson en un clúster de GPU.

4.2. Transposiciones

4.2.1. Notación y terminología

Como se ha comentado, existen dos posibilidades a la hora de realizar transposiciones que dependen de si se requiere optimizar el uso de memoria (*in-place*) o no se requiere optimizar el uso de memoria (*out-of-place*).

Dependiendo de las dimensiones de la matriz a tratar, existen diferentes maneras para transponerla, además de la transpuesta trivial que simplemente copia la matriz dejándola inalterada (transposición de identidad). Por otra parte, una matriz se puede almacenar en memoria con diferentes ordenaciones de sus datos y gráficamente puede ser representado de varias maneras.

Tal y como se ha dicho en el capítulo 2 (sección 2.3.4), en el caso de una matriz en 3D, la transposición se puede definir como una permutación de los ejes x, y y z. Por lo tanto, existen 6 permutaciones posibles para transponer el contenido de una matriz en 3D. Y como ya se ha dicho, una de esas permutaciones es trivial, ya que simplemente copia la matriz de origen a la matriz de destino sin alterar su contenido. Para el resto de las 5 permutaciones, es necesario definir una notación que posteriormente será utilizada en el presente trabajo de investigación. Sea T la transposición de una matriz 3D, y su subíndice, la permutación que se le aplica, tenemos que el resultado de la transposición para cada permutación de la matriz de entrada en 3D $A[i, j, k]$ es:

- $T_{zxy}(A) = A[k, i, j]$,
- $T_{yzx}(A) = A[j, k, i]$,
- $T_{xzy}(A) = A[i, k, j]$,
- $T_{yxz}(A) = A[j, i, k]$ y
- $T_{zyx}(A) = A[k, j, i]$, donde $0 \leq i < n_x,\ 0 \leq j < n_y,\ 0 \leq k < n_z$.

Cabe reseñar que 3 permutaciones de las transposiciones dejan una de las dimensiones en su posición inicial —$\mathbf{x}zy$, $z\mathbf{y}x$ y $yx\mathbf{z}$— y por lo tanto, pueden ser interpretadas como transposiciones en 2D de una serie de planos. Además, esto implica que son involuciones, es decir, para estos casos se cumple que $T(T(A)) = A$. Obsérvese también que la transposición zyx permuta entre los datos almacenados por columnas a datos ordenados por filas. Por último, el resto de las permutaciones no triviales, yzx y zxy son rotaciones, lo cual significa que para estos casos $T(T(T(A))) = A$. Del mismo modo, también se cumple que $T_{yzx}(T_{zxy}(A)) = A$ y $T_{yzx}(T_{yzx}(A)) = T_{zxy}(A)$.

A continuación, se va a describir la notación utilizada a lo largo de este capítulo para explicar las diferentes implementaciones que se han realizado. Supongamos que tenemos una matriz de datos en tres dimensiones almacenada de forma lineal en la memoria de una GPU. Llamaremos x, y y z a las tres dimensiones de la matriz, que representan las filas, las columnas y los planos, respectivamente. El tamaño de la matriz será de $N = n_x \times n_y \times n_z$ elementos. En la figura 4.1 se observa una matriz de $N = 4 \times 4 \times 4$ elementos.

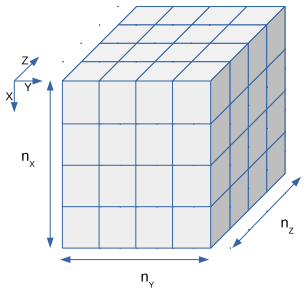

Figura 4.1: Matriz de $4 \times 4 \times 4$ elementos

Vamos a suponer que la matriz se almacena en la memoria de la GPU ordenada por columnas, tal que los elementos en la dimensión x se encuentran en posiciones contiguas en memoria, mientras que los elementos en las dimensiones y y z se separan por n_x y $n_x \times n_y$ elementos, respectivamente. En

otras palabras, el desplazamiento en memoria del elemento (i, j, k) respecto al primer elemento de la matriz es de $i + j \times n_x + k \times n_x \times n_y$ elementos.

4.3. Metodología experimental

Todas las transposiciones en 3D que se han desarrollado en este trabajo tienen en cuenta las optimizaciones descritas por G. Ruetsch y P. Micikevicius para transponer eficientemente una matriz de 2D en una GPU [89]. En ese trabajo, las matrices se descomponen en baldosas cuadradas de 32×32 elementos donde cada bloque de hilos transpone una de esas baldosas. El tamaño de estos bloques es de 32×4 hilos, y de esta manera, cada hilo transpone 8 elementos, por lo que gran parte del costo de cálculo de los índices para acceder a memoria se amortiza con esta distribución, ya que una vez calculado el índice del primer elemento, para el resto de elementos no es necesario repetir los mismos cálculos.

La primera optimización que se propone en el trabajo de G. Ruetsch y P. Micikevicius, consiste en almacenar temporalmente los datos en la memoria compartida. De esta manera, los hilos podrían tanto, leer y escribir de manera coalescente en la memoria global. La segunda optimización consiste en aumentar el tamaño de la memoria compartida para evitar los conflictos con los bancos de memoria (*padding*). La tercera, y última optimización consiste en distribuir los bloques diagonalmente para que no disminuya el rendimiento a la hora de acceder a los datos en memoria debido al problema denominado *partition camping*.

Para nuestro trabajo se han definido baldosas (*tiles*) de 2 dimensiones que han sido transpuestas por bloques de 2 dimensiones de CUDA.

Por ejemplo, como la transposición yxz implica la transposición del plano xy, se han definido $\left\lceil \frac{n_x}{TD} \right\rceil \times \left\lceil \frac{n_y}{TD} \right\rceil \times n_z$ baldosas, donde TD corresponde al tamaño del lado de la baldosa (*Tile Dimension*). En otras palabras, hemos cubierto cada uno de los n_z planos xy como un conjunto de baldosas cuadradas.

Hemos añadido una comprobación, no incluida en el trabajo de G. Ruetsch y P. Micikevicius, para asegurarnos de que los hilos cuyos índices superan el tamaño de las dimensiones de la matriz no accedan a la memoria global. Esta modificación permite la realización de transposiciones de matrices cuyas

dimensiones no sean múltiplos de una baldosa.

Con el fin de seleccionar el tamaño óptimo de la baldosa, se han realizado pruebas con tres tamaños de baldosa diferentes: 8×8, 16×16 y 32×32. No se ha probado con el tamaño de baldosa de 64×64 debido a que dependiendo del tipo de los datos, se puede desbordar el tamaño de la memoria compartida de los dispositivos actuales o, incluso en caso de caber, el nivel de ocupación de los SM es bajo, con lo cual se pierde rendimiento. Resultados preliminares descartan claramente la utilización de baldosas de tamaño de 8×8, debido a que realizan accesos a memoria con coalescencia limitada y porque el tamaño de los bloques de hilos que han de utilizarse son demasiado pequeños. El tamaño de baldosa de 16×16 se comportó mejor en líneas generales, lo que finalmente ha hecho decantarse por este tamaño de baldosa para realizar la experimentación.

En lo que respecta al tamaño de los bloques de hilos, el cual afecta directamente al número de elementos que un solo hilo transpone, la estrategia más sencilla puede ser que cada hilo transponga un sólo elemento. Pero también puede ser rentable que cada hilo se encargue de transponer más de un elemento y así –tal y como se ha dicho anteriormente– ahorrar en los cálculos de los índices de los elementos. De este modo, tenemos dos estrategias posibles, donde los bloques se pueden hacer más pequeños o hacer menos bloques. Para encontrar cuál es la estrategia más adecuada, se han realizado pruebas con tres tamaños de bloque diferentes: 16×16, 16×8 y 16×4, donde cada hilo transpone 1, 2 o 4 elementos respectivamente. También se ha realizado un análisis para examinar si el número de baldosas a transponer por cada bloque afecta en el rendimiento final obtenido. Aquí, la estrategia es que cada bloque se encargue de más de una baldosa. Los resultados muestran que este parámetro no es relevante y, por lo tanto, ha sido descartado en la experimentación final.

Toda la experimentación se ha realizado en GPU con arquitectura Fermi, donde el problema del *partition camping* se ha reducido drásticamente gracias al nuevo diseño implementado en los nuevos controladores de memoria [89]. Por lo tanto, esta optimización ha sido ignorada en esta experimentación.

En los siguientes apartados se va a describir la implementación que se ha desarrollado para realizar transposiciones en 3D. En primer lugar, se describen las transposiciones *out-of-place* y, a continuación, se describen las transposiciones *in-place*, limitadas a un conjunto particular de tamaños para la matriz de entrada.

4.3.1. Transposiciones out-of-place

La transposición de identidad

Si bien la transposición de identidad, xyz, es una transposición trivial que implica una simple copia de datos, se va a usar para evaluar la calidad de los resultados del resto de las transposiciones, ya que define el límite del rendimiento máximo obtenible en este tipo de operaciones en memoria.

A continuación se van a presentar los algoritmos correspondientes a las implementaciones para las transposiciones *out-of-place*. Para todos estos algoritmos hay que tener en cuenta que, por claridad, únicamente se van a mostrar los algoritmos que transponen un único elemento por hilo ($EpT = 1$). La variable b corresponde a la coordenada del bloque dentro de la malla (*grid*) de CUDA (lo que en terminología CUDA se denomina la variable *blockIdx*). Igualmente, l es la coordenada local del hilo dentro del bloque de CUDA (lo que en terminología CUDA se denomina la variable *threadIdx*). Desde estas coordenadas, cada hilo puede calcular las coordenadas globales de cualquier punto que necesite leer (x_r, y_r, z_r) y donde necesite escribir (x_w, y_w, z_w). Por último, las variables (n_x, n_y, n_z) indican el número de elementos por cada dimensión.

En el algoritmo 1 se explica en términos generales la implementación del kernel encargado de realizar la transposición de identidad xyz.

Algoritmo 1 Algoritmo del kernel para la transposición out-of-place xyz.

Entrada: A, n_x, n_y
Salida: $T_{xyz}(A)$

1: $x_r = x_w = b_x \cdot TD + l_x$
2: $y_r = y_w = b_y \cdot TD + l_y$
3: $z_r = z_w = b_z$
4:
5: **if** $x_r >= n_x$ **or** $y_r >= n_y$ **then**
6: **return**
7: **end if**
8:
9: $\mathrm{A}[x_r + (y_r + z_r \cdot n_y) \cdot n_x] = \mathrm{A}[x_r + (y_r + z_r \cdot n_y) \cdot n_x]$

En la línea 9 de este algoritmo se realizan las operaciones de lectura y escritura en la misma posición de la memoria global.

Finalmente, en las líneas 5 y 6 el algoritmo se asegura de que los hilos del kernel no puedan acceder fuera de los límites de la memoria, finalizando aquellos que sí superen esos límites.

Las transposiciones de involución

Las transposiciones de involución son las inversas de ellas mismas. En otras palabras, son aquellas que fijando una dimensión transponen las dos restantes. El resultado es que la transposición en 3D se reduce a transposiciones en 2D de todos los planos de las dimensiones que se intercambian.

Transposición yxz

La transposición más simple corresponde a yxz. En este caso, la transposición 3D se reduce eficazmente a transposiciones 2D de n_z planos xy (ver figura 4.2a).

Por lo tanto, se lanzan $\left\lceil \frac{n_x}{TD} \right\rceil \times \left\lceil \frac{n_y}{TD} \right\rceil \times n_z$ bloques de tamaño $TD \times \frac{TD}{EpT} \times 1$, donde EpT corresponde al número de elementos que transpone cada hilo (*Elements-per-Thread*). Finalmente, cada bloque transpone una baldosa del plano xy.

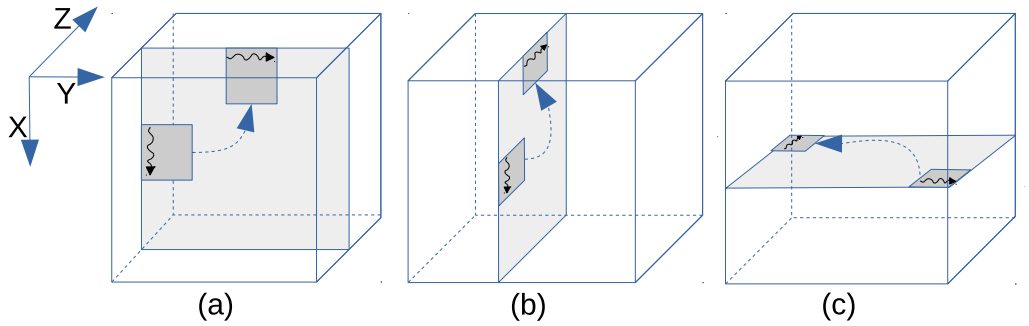

Figura 4.2: Tipos de transposiciones de involución: (a) yxz, (b) zyx and (c) xzy

En el algoritmo 2 se explica en términos generales la implementación del

kernel encargado de realizar la transposición yxz. Nótese que $z_r = z_w$ porque el kernel transpone planos xy, donde la coordenada z se mantiene fija.

Algoritmo 2 Algoritmo del kernel para la transposición out-of-place yxz.

Entrada: A, n_x, n_y
Salida: $T_{yxz}(A)$

1: $x_r = b_x \cdot TD + l_x$; $x_w = b_x \cdot TD + l_y$
2: $y_r = b_y \cdot TD + l_y$; $y_w = b_y \cdot TD + l_x$
3: $z_r = z_w = b_z$
4:
5: **if** $x_r < n_x$ **and** $y_r < n_y$ **then**
6: tile$[l_x][l_y]$ = A$[x_r + (y_r + z_r \cdot n_y) \cdot n_x]$
7: **end if**
8: barrier
9: **if** $x_w < n_y$ **and** $y_w < n_x$ **then**
10: A$[y_w + (x_w + z_w \cdot n_x) \cdot n_y]$ = tile$[l_y][l_x]$
11: **end if**

En las líneas 6 y 10 de este algoritmo se realizan las operaciones de lectura y escritura en la memoria global, respectivamente. Nótese que la operación de escritura, del mismo modo que en el cálculo de las coordenadas x_w e y_w, intercambian la "posición natural" de las coordenadas l_x y l_y. Este intercambio posibilita al kernel acceder a la memoria global de manera coalescente. No obstante, requiere que todos los hilos de un bloque se sincronicen antes de realizar la operación de escritura (línea 8).

Finalmente, en las líneas 5 y 9 el algoritmo se asegura de que el kernel también funcione para matrices cuyas dimensiones n_x y n_y no sean múltiplos de TD. Para ello, simplemente se inhibe el acceso a memoria para aquellos hilos que puedan acceder fuera de los límites de la memoria. Nótese que estos hilos no son destruidos porque aunque hayan sido inhibidos para la operación de lectura pueden ser requeridos en la operación de escritura.

Transposición zyx

La transposición zyx, aún siendo conceptualmente similar, dispone de una serie de características que merece la pena analizar. En este caso, la transposición se realiza sobre el plano xz (ver figura 4.2b). Aunque en este caso también se estén realizando transposiciones de un conjunto de planos, éstos ya no se encuentran en posiciones contiguas de memoria. Realmente,

cada columna se encuentra ordenada de manera contigua, pero separada de su columna vecina por un salto de $n_x \times n_y$ elementos. De todos modos, esta diferencia no supone un gran cambio, ya que se pueden lanzar $\left\lceil \frac{n_x}{TD} \right\rceil \times n_y \times \left\lceil \frac{n_z}{TD} \right\rceil$ bloques de tamaño $TD \times 1 \times \frac{TD}{EpT}$ y modificar el kernel para que tenga en cuenta dichos cambios. Desafortunadamente, esta configuración no obtiene unos resultados óptimos, ya que parece que CUDA penaliza el aumento de bloques de hilos en la dimensión z. La solución consiste en intercambiar las dimensiones de los bloques y y z de CUDA y adaptar el código para que las dimensiones y y z del *grid* de bloques de CUDA correspondan a las dimensiones z e y de la matriz de datos. El resultado supone una mejora en la eficiencia al realizar transposiciones a costa de un código de mayor complejidad.

En el algoritmo 3 se explica en términos generales la implementación del kernel encargado de realizar la transposición zyx. Nótese que $y_r = y_w$ porque el kernel transpone planos xz, donde la coordenada y se mantiene fija.

Algoritmo 3 Algoritmo del kernel para la transposición out-of-place zyx.
Entrada: A, n_x, n_y, n_z
Salida: $T_{zyx}(A)$

```
1: x_r = b_x · TD + l_x; x_w = b_x · TD + l_y
2: z_r = b_y · TD + l_y; z_w = b_y · TD + l_x
3: y_r = y_w = b_z
4:
5: if x_r < n_x and z_r < n_z then
6:     tile[l_x][l_y] = A[x_r + (y_r + z_r · n_y) · n_x]
7: end if
8: barrier
9: if z_w < n_z and x_w < n_x then
10:    A[z_w + (y_w + x_w · n_y) · n_z] = tile[l_y][l_x]
11: end if
```

Al igual que en el algoritmo 2, en las líneas 6 y 10 se realizan las operaciones de lectura y escritura en la memoria global y en las líneas 5 y 9 el algoritmo se asegura de que el kernel también funcione para matrices cuyas dimensiones n_x y n_z no sean múltiplos de TD.

Estas mismas instrucciones se repiten en el resto de algoritmos que se han implementado para todas las transposiciones *out-of-place* a presentar en esta

sección. Por lo que, por no repetirse, esta explicación se ha omitido en el resto de algoritmos.

Transposición xzy

La última transposición de involución corresponde a la xzy. Una vez más, la transposición en 3D puede ser sustituida por n_x transposiciones 2D de los planos yz (ver figura 4.2c). No obstante, esta aproximación es a todas luces ineficiente.

De cara a entender cuál es el problema, hay que recordar que los datos se almacenan ordenados por columnas, y por lo tanto, se encuentran en posiciones de memoria contiguas para el eje x. Las dos transposiciones anteriores pueden acceder a la memoria global de manera coalescente porque los planos transpuestos –xy y xz– trabajan con la dimensión x, y por lo tanto, pueden ser vistas como compuestas por columnas. El paso intermedio en la memoria compartida permite que las operaciones tanto de lectura como de escritura se realicen de manera coalescente.

Sin embargo, la transposición xzy se realiza sobre el plano yz, que no afecta al eje x, y por lo tanto, no puede considerarse que esté compuesta por columnas. Esta simple aproximación no obtiene ningún tipo de coalescencia para ningún acceso a memoria.

Afortunadamente, una ordenación diferente para las baldosas, permite conseguir accesos totalmente coalescentes. Para ello, las baldosas se han ordenado tal y como se hicieron para la transposición yxz. De esta manera, cada baldosa se encarga de TD segmentos de filas ordenadas verticalmente, que corresponden a TD planos yz. La coalescencia se obtiene porque todos los elementos i de cada segmento de las filas de la baldosa conforman una columna, y por lo tanto, se encuentran en posiciones de memoria contiguas. El resultado es que cada bloque no transpone la baldosa en el plano yz sino que transpone TD segmentos de fila en el plano xy (ver la figura 4.3 para comprender la solución propuesta).

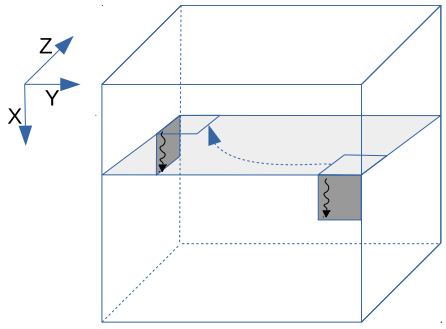

Figura 4.3: Implementación de la transposición xzy con acceso coalescente a memoria. Cada bloque transpone un segmento de fila de TD planos yz

En el algoritmo 4 se explica en términos generales la implementación del kernel encargado de realizar la transposición xzy. Nótese que $x_r = x_w$ porque el kernel transpone planos yz, donde la coordenada x se mantiene fija.

Algoritmo 4 Algoritmo del kernel para la transposición out-of-place xzy.

Entrada: A, n_x, n_y, n_z
Salida: $T_{xzy}(A)$

1: $x_r = x_w = b_x \cdot TD + l_x$
2: $y_r = y_w = b_y \cdot TD + l_y$
3: $z_r = z_w = b_z$
4:
5: **if** $x_r < n_x$ **and** $y_r < n_y$ **then**
6: $\quad \text{tile}[l_x][l_y] = \text{A}[x_r + (y_r + z_r \cdot n_y) \cdot n_x]$
7: **end if**
8: barrier
9: **if** $x_w < n_x$ **and** $y_w < n_y$ **then**
10: $\quad \text{A}[x_w + (z_w + y_w \cdot n_z) \cdot n_x] = \text{tile}[l_x][l_y]$
11: **end if**

Las transposiciones de rotación

Para las restantes dos transposiciones –yzx y zxy– la solución basada en la realización de transposiciones con matrices de 2D, no es válida. Sin embargo, un ligero cambio de punto de vista hace que la estrategia utilizada anteriormente sea válida para este tipo de transposiciones.

Transposición yzx

Las anteriores transposiciones movían los datos que se encontraban dentro de un plano sin afectar a otros planos. En la nueva estrategia, sí que se ven afectados otros planos, ya que para realizar la transpuesta yzx lo que se hace es almacenar un plano xy como un plano xz. Tal y como se puede ver en la figura 4.4, cada bloque lee una baldosa del plano xy. A continuación, se transpone, y finalmente, se escribe el resultado en una baldosa del plano xz. Hay que tener en cuenta que ambos planos constan del eje x, por lo que los accesos a memoria pueden realizarse de manera coalescente.

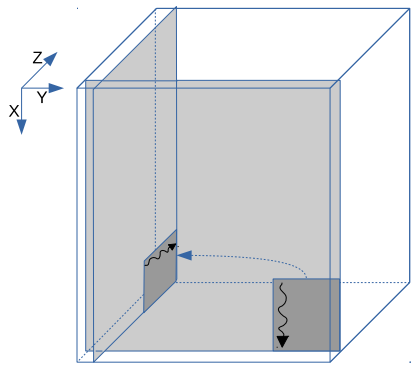

Figura 4.4: Implementación de la transposición de rotación yzx

En el algoritmo 5 se explica en términos generales la implementación del kernel encargado de realizar la transposición yzx.

Algoritmo 5 Algoritmo del kernel para la transposición out-of-place yzx.

Entrada: A, n_x, n_y, n_z
Salida: $T_{yzx}(A)$

1: $x_r = b_x \cdot TD + l_x$; $x_w = b_x \cdot TD + l_y$
2: $y_r = b_y \cdot TD + l_y$; $y_w = b_y \cdot TD + l_x$
3: $z_r = z_w = b_z$
4:
5: **if** $x_r < n_x$ **and** $y_r < n_y$ **then**
6: $\quad \text{tile}[l_x][l_y] = \text{A}[x_r + (y_r + z_r \cdot n_y) \cdot n_x]$
7: **end if**
8: barrier
9: **if** $y_w < n_y$ **and** $x_w < n_x$ **then**
10: $\quad \text{A}[y_w + (z_w + x_w \cdot n_z) \cdot n_y] = \text{tile}[l_y][l_x]$
11: **end if**

Transposición zxy

La rotación zxy se calcula de manera inversa. Cada bloque lee una baldosa del plano xz y la escribe en una baldosa de plano xy. Hay que tener en cuenta que, la lectura de una baldosa en el plano xy supone tener que escribirlo en el plano yz, impidiendo un acceso coalescente a memoria. Es decir, la selección del plano para ordenar las baldosas no es arbitraria, y debe realizarse con el objetivo de conseguir un acceso coalescente a memoria.

En el algoritmo 6 se explica en términos generales la implementación del kernel encargado de realizar la transposición zxy.

Algoritmo 6 Algoritmo del kernel para la transposición out-of-place zxy.
Entrada: A, n_x, n_y, n_z
Salida: $T_{zxy}(A)$

1: $x_r = b_x \cdot TD + l_x$; $x_w = b_x \cdot TD + l_y$
2: $z_r = b_y \cdot TD + l_y$; $z_w = b_y \cdot TD + l_x$
3: $y_r = y_w = b_z$
4:
5: **if** $x_r < n_x$ **and** $z_r < n_z$ **then**
6: tile$[l_x][l_y]$ = A$[x_r + (y_r + z_r \cdot n_y) \cdot n_x]$
7: **end if**
8: barrier
9: **if** $x_w < n_x$ **and** $z_w < n_z$ **then**
10: A$[z_w + (x_w + y_w \cdot n_x) \cdot n_z]$ = tile$[l_y][l_x]$
11: **end if**

4.3.2. Transposiciones in-place

Las transposiciones de involución

La utilización de transposiciones *in-place* aportan un beneficio indudable a cualquier aplicación en la que se implementen debido a que reducen el uso de la memoria. Este menor consumo de memoria permite que problemas con mayor tamaño de datos puedan ser almacenados en la memoria de la GPU, reduciendo así el ratio de sobrecarga por transferencia/cálculo.

Aunque la implementación de transposiciones *in-place* no ha sido uno de

los objetivos principales de este trabajo de investigación, se ha implementado y analizado las transposiciones *in-place* que no requieren de algoritmos complejos. Por ejemplo, para las transposiciones de involución se traduce en que los planos transpuestos deben ser cuadrados. Este hecho asegura que el elemento X se moverá a la posición del elemento Y, que a su vez, se moverá a la posición del elemento X. Por lo tanto, podemos modificar el kernel de la transposición de modo que un hilo intercambia dos elementos de los datos, en lugar de sólo mover uno de ellos. Esta implementación requiere de sólo un espacio auxiliar constante igual al doble del tamaño de una baldosa (256 elementos), y que es almacenado en la memoria compartida.

Para las transposiciones yxz y zyx, los hilos de la diagonal superior de cada plano son desechados, mientras que los hilos de la diagonal inferior leen los datos correspondientes a dos baldosas, que posteriormente son llevadas a la memoria compartida y que una vez han sido transpuestas son trasladadas a la memoria principal. Finalmente, aquellos hilos que se encuentran en la diagonal se transponen de la misma manera que para las transposiciones *out-of-place*.

A continuación se detallan los algoritmos de las transposiciones de involución *in-place*.

Transposición yxz

En el algoritmo 7 se explica en términos generales la implementación del kernel encargado de realizar la transposición yxz.

Aunque el algoritmo necesita los valores de n_x y n_y, como se ha dicho que los datos a transponer tienen que ser planos cuadrados, sólo es necesario pasar uno de ellos –ya que ambos tienen el mismo valor– y por claridad en el algoritmo se usa la variable n_x para referenciar a ambas (n_x y n_y).

Por claridad en los algoritmos, se han omitido todas las instrucciones condicionales encargadas de verificar los tamaños de las matrices de entrada

Algoritmo 7 Algoritmo del kernel para la transposición in-place yxz.

Entrada: A, n_x
Salida: $T_{yxz}(A)$

1: **if** $b_x > b_y$ **then**
2: **return**
3: **end if**
4:
5: $x_{inf} = b_x \cdot TD + l_x$; $y_{inf} = b_y \cdot TD + l_y$;
6: $x_{sup} = b_x \cdot TD + l_y$; $y_{sup} = b_y \cdot TD + l_x$;
7: $z_{inf} = z_{sup} = b_z$
8: $o_{inf} = x_{inf} + (y_{inf} + z_{inf} \cdot n_x) \cdot n_x$
9: $o_{sup} = y_{sup} + (x_{sup} + z_{sup} \cdot n_x) \cdot n_x$
10: inf_tile$[l_x][l_y]$ = A$[o_{inf}]$
11:
12: **if** $b_x < b_y$ **then**
13: sup_tile$[l_x][l_y]$ = A$[o_{sup}]$
14: **else**
15: $o_{sup} = o_{inf}$
16: **end if**
17: barrier
18: A$[o_{sup}]$ = inf_tile$[l_y][l_x]$
19: **if** $b_x < b_y$ **then**
20: A$[o_{inf}]$ = sup_tile$[l_y][l_x]$
21: **end if**

Hay que recordar que los hilos de la diagonal superior de cada plano son desechados para todos los algoritmos (línea 1 de los algoritmos 7 y 8). Entonces, cada hilo calcula sus propias coordenadas para cada baldosa (x,y) y el correspondiente desplazamiento (*offset*, o) dentro de la matriz.

Del mismo modo, en los algoritmos ha sido necesario implementar algún mecanismo de sincronización a nivel de bloque, ya que cada hilo no escribe en la memoria global el mismo valor que ha leído (línea 17).

Transposición zyx

En el algoritmo 8 se explica en términos generales la implementación del kernel encargado de realizar la transposición zyx.

Algoritmo 8 Algoritmo del kernel para la transposición in-place zyx.

Entrada: A, n_x
Salida: $T_{zyx}(A)$

1: **if** $b_x > b_y$ **then**
2: **return**
3: **end if**
4:
5: $x_{inf} = b_x \cdot TD + l_x$; $z_{inf} = b_y \cdot TD + l_y$;
6: $x_{sup} = b_x \cdot TD + l_y$; $z_{sup} = b_y \cdot TD + l_x$;
7: $y_{inf} = y_{sup} = b_z$
8: $o_{inf} = x_{inf} + (y_{inf} + z_{inf} \cdot n_x) \cdot n_x$
9: $o_{sup} = z_{sup} + (y_{sup} + x_{sup} \cdot n_x) \cdot n_x$
10: inf_tile$[l_x][l_y]$ = A$[o_{inf}]$
11:
12: **if** $b_x < b_y$ **then**
13: sup_tile$[l_x][l_y]$ = A$[o_{sup}]$
14: **else**
15: $o_{sup} = o_{inf}$
16: **end if**
17: barrier
18: A$[o_{sup}]$ = inf_tile$[l_y][l_x]$
19: **if** $b_x < b_y$ **then**
20: A$[o_{inf}]$ = sup_tile$[l_y][l_x]$
21: **end if**

Este algoritmo es muy similar al algoritmo 7.

Transposición xzy

La transposición *out-of-place* xzy ha requerido de una implementación diferente de cara a acceder a memoria coalescentemente. Por ello, las modificaciones necesarias para convertirla en una transposición *in-place* difieren de las anteriormente descritas, en las que cada hilo que no se encuentre en la diagonal inferior del plano yz es desechado. Los hilos restantes se modifican para que intercambien dos elementos, en lugar de sólo mover uno de ellos.

En el algoritmo 9 se explica en términos generales la implementación del kernel encargado de realizar la transposición xzy.

Algoritmo 9 Algoritmo del kernel para la transposición in-place xzy.

Entrada: A, n_x
Salida: $T_{xzy}(A)$

1: $x_r = x_w = b_x \cdot TD + l_x$
2: $y_r = y_w = b_y \cdot TD + l_y$
3: $z_r = z_w = b_z$
4:
5: **if** $z_r > y_r$ **then**
6: **return**
7: **end if**
8: $o_{inf} = x_r + (y_r + z_r \cdot n_y) \cdot n_x$
9: $o_{sup} = x_r + (z_r + y_r \cdot n_y) \cdot n_x$
10: inf_tile$[l_x][l_y]$ = A$[o_{inf}]$
11: sup_tile$[l_x][l_y]$ = A$[o_{sup}]$
12: barrier
13: A$[o_{sup}]$ = inf_tile$[l_x][l_y]$
14: A$[o_{inf}]$ = sup_tile$[l_x][l_y]$

Al igual que en los anteriores algoritmos, algunos hilos de cada plano son desechados (línea 5) y, por lo tanto, es necesario calcular el correspondiente desplazamiento. No obstante, a diferencia de los anteriores algoritmos los hilos que se desechan corresponden al plano yz.

Como cada hilo no escribe en la memoria global el mismo valor que ha leído, es necesario implementar algún mecanismo de sincronización a nivel de bloque (línea 12).

Las transposiciones de rotación

En el caso de las transposiciones de rotación, la adaptación a una implementación *in-place* es aún más compleja. Tal y como se ha indicado en el apartado anterior, esta transposición no puede se reducida a un conjunto de transposiciones 2D, ya que las baldosas de un plano deben ser movidas a otro plano. Por lo tanto, el problema radica en que si el elemento X debe ser movido a la posición del elemento Y, el elemento Y debe ser movido a la posición del elemento Z, $X \neq Z$.

Afortunadamente, debido a que estamos trabajando con rotaciones, se

puede observar que, si la matriz de entrada es cúbica ($n_x = n_y = n_z$), el elemento Z debe ser movido a la posición del elemento X. Por lo tanto, para este trabajo, las transposiciones de rotación únicamente se han implementado para los casos en los que los datos de entrada son matrices cúbicas.

Transposición yzx

La transposición yzx implica el movimiento de una baldosa desde el plano xy al plano xz. Para ello, se debe mover la baldosa que ha sido sobrescrita –y que previamente se ha almacenado en la memoria compartida– al plano yz. Para, en un último paso, mover la baldosa en el plano yz –que también ha sido almacenada en la memoria compartida– a la posición de la primera baldosa del plano xy.

Estos movimientos implican que cada hilo debe realizar 3 lecturas y 3 escrituras en la memoria global. Nótese que, tal y como se ha visto para la transposición xzy, para aquellos elementos que se encuentren en el plano yz no se podrá conseguir un acceso coalescente a memoria. Por lo tanto, esta implementación obtiene unos resultados no óptimos –entre 3 y 4 veces inferiores a los obtenidos para las transposiciones *in-place* de involución.

La mejor manera para conseguir la coalescencia para cualquier plano consiste en utilizar baldosas 3D cúbicas a los cuales les hemos llamado *"ladrillos"*. El principal inconveniente de esta aproximación consiste en la necesidad de disponer de memoria compartida de mayor tamaño. Por ejemplo, para un ladrillo de $16 \times 16 \times 16$ elementos en coma flotante de doble precisión, se necesitan 32 kilobytes (en realidad, esta cifra es un poco más grande debido al relleno que se ha introducido para evitar los conflictos en los bancos de memoria). Debido a que se necesitan mover 3 ladrillos por cada bloque, no hay espacio suficiente en la memoria compartida de una GPU –incluso optimizando el espacio necesario a tan solo 2 ladrillos, tal y como se ha realizado (ver el algoritmo 10). Por lo tanto, se han usado ladrillos de $8 \times 8 \times 8$, que, aunque reducen a la mitad el nivel de coalescencia obtenido, permiten realizar transposiciones con datos reales en coma flotante de doble precisión e incrementan la ocupación debido al menor requerimiento de espacio en la memoria compartida.

La implementación puede verse de manera gráfica en la figura 4.5, y algorítmicamente en el algoritmo 10.

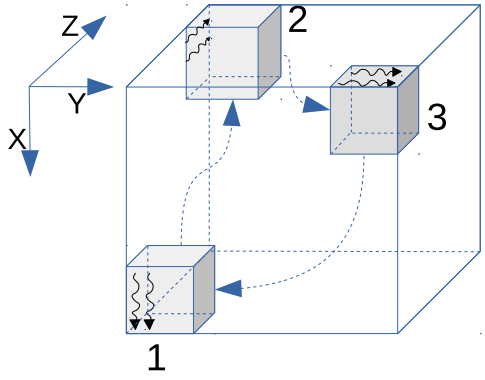

Figura 4.5: Implementación de la transposición in-place yzx

Algoritmo 10 Algoritmo del kernel para la transposición in-place yzx.

Entrada: A, n_x
Salida: $T_{yzx}(A)$

1: **if** $b_x > b_y$ **or** $b_x > b_z$ **or**
   $((b_x = b_y$ **or** $b_x = b_z)$ **and** $b_y > b_z)$ **then**
2: **return**
3: **end if**
4:
5: $x_1 = b_x \cdot BD + l_x$; $y_1 = b_y \cdot BD + l_y$; $z_1 = b_z \cdot BD + l_z$
6: $x_2 = b_y \cdot BD + l_x$; $y_2 = b_z \cdot BD + l_y$; $z_2 = b_x \cdot BD + l_z$
7: $x_3 = b_z \cdot BD + l_x$; $y_3 = b_x \cdot BD + l_y$; $z_3 = b_y \cdot BD + l_z$
8: $o_1 = x_1 + (y_1 + z_1 \cdot n_x) \cdot n_x$
9: $o_2 = x_2 + (y_2 + z_2 \cdot n_x) \cdot n_x$
10: $o_3 = x_3 + (y_3 + z_3 \cdot n_x) \cdot n_x$
11:
12: brick1$[l_y][l_x][l_z]$ = A$[o_1]$; barrier
13: **if** $b_x = b_y = b_z$ **then**
14: A$[o_1]$ = brick1$[l_x][l_z][l_y]$
15: **else**
16: brick2$[l_x][l_y][l_z]$ = A$[o_2]$; barrier
17: A$[o_2]$ = brick1$[l_x][l_y][l_z]$; barrier
18: brick1$[l_x][l_y][l_z]$ = A$[o_3]$
19: A$[o_3]$ = brick2$[l_y][l_z][l_x]$; barrier
20: A$[o_1]$ = brick1$[l_z][l_y][l_x]$
21: **end if**

Aunqque los algoritmos 10 y 11 necesitan los valores de n_x, n_y y n_z, como se ha dicho que los datos a transponer tienen que ser baldosas 3D cúbicas, sólo es necesario pasar uno de ellos –ya que todos tienen el mismo valor– y por claridad en el algoritmo se usa la variable n_x para referenciar a todas ellas (n_x, n_y y n_z).

Hay que recordar que en el caso de ambos algoritmos cada hilo transpone 3 elementos –uno por plano– y por lo tanto, sólo se deben mantener 1/3 de los hilos que se han creado (línea 1 de ambos algoritmos). Entonces, cada hilo calcula sus propias coordenadas para cada ladrillo (x, y, z) y el correspondiente desplazamiento (*offset*, o) dentro de la matriz.

A continuación, cada bloque lee el primer ladrillo de la memoria compartida. Si el bloque está en la diagonal principal ($b_x = b_y = b_z$), el ladrillo se transpone directamente y se escribe en la misma ubicación de la memoria global. En caso contrario, el segundo ladrillo se trae a la memoria compartida y el primero es transpuesto y escrito desde la memoria compartida a la memoria global.

Estas operaciones, liberan los espacios de la memoria compartida que contenían al primer ladrillo y que van a ser usados para almacenar el tercer ladrillo. Una vez que el tercer ladrillo ha sido almacenado en la memoria compartida, es reemplazado por la versión transpuesta del segundo ladrillo que está almacenada en la memoria global. Por último, el tercer ladrillo es transpuesto y almacenado en la memoria global en la misma posición de la que fue leído el primer ladrillo.

Por claridad en los algoritmos, se han omitido todas las instrucciones condicionales encargadas de verificar los tamaños de las matrices de entradas para cuando no son múltiplos del tamaño del ladrillo. También se ha sustituido el término TD por BD (dimensión del ladrillo, *Brick Dimension*).

Del mismo modo, en ambos algoritmos ha sido necesario implementar algunos mecanismos de sincronización a nivel de bloque, ya que cada hilo no escribe en la memoria global el mismo valor que ha leído. No obstante, una pequeña modificación en la manera de almacenar el primer ladrillo en la memoria compartida permite la eliminación de la barrera de la línea 17 en ambos algoritmos. Esta versión optimizada ha sido la que se ha utilizado en las experimentaciones, aunque la mejora del rendimiento obtenida no ha sido muy significativa.

Es necesario recordar que el tamaño del ladrillo es de 8 elementos, y

que para todas sus dimensiones nos hemos asegurado de que los datos sean accedidos en grupos de 8 elementos y en posiciones de memoria consecutivas.

Transposición zxy

La transposición zxy, implica el movimiento de una baldosa desde el plano xy al plano yz. Para ello, se debe mover la baldosa que ha sido sobrescrita –y que previamente se ha almacenado en la memoria compartida– al plano xz. Para, en un último paso, mover la baldosa en el plano xz –que también ha sido almacenada en la memoria compartida– a la posición de la primera baldosa del plano xy.

Estos movimientos implican que cada hilo debe realizar 3 lecturas y 3 escrituras en la memoria global. Nótese que, tal y como se ha visto para la transposición yzx, hay dificultades conseguir un acceso coalescente a memoria. Que se soluciona mediante la utilización de baldosas 3D cúbicas, tal y como se ha hecho en la transposición yzx.

La implementación puede verse de manera gráfica en la figura 4.6, y algorítmicamente en el algoritmo 11.

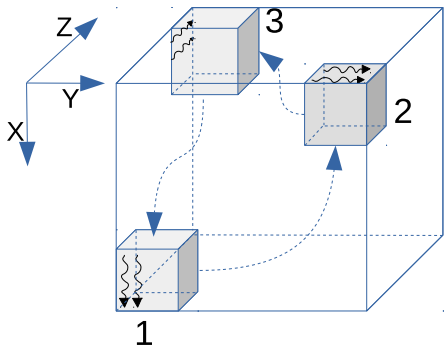

Figura 4.6: Implementación de la transposición in-place zxy

En el algoritmo 11 se explica en términos generales la implementación del kernel encargado de realizar la transposición zxy.

Algoritmo 11 Algoritmo del kernel para la transposición in-place zxy.

Entrada: A, n_x
Salida: $T_{zxy}(A)$

1: **if** $b_x > b_y$ **or** $b_x > b_z$ **or**
   $((b_x = b_y$ **or** $b_x = b_z)$ **and** $b_y > b_z)$ **then**
2: **return**
3: **end if**
4:
5: $x_1 = b_x \cdot BD + l_x$; $y_1 = b_y \cdot BD + l_y$; $z_1 = b_z \cdot BD + l_z$
6: $x_2 = b_y \cdot BD + l_x$; $y_2 = b_z \cdot BD + l_y$; $z_2 = b_x \cdot BD + l_z$
7: $x_3 = b_z \cdot BD + l_x$; $y_3 = b_x \cdot BD + l_y$; $z_3 = b_y \cdot BD + l_z$
8: $o_1 = x_1 + (y_1 + z_1 \cdot n_x) \cdot n_x$
9: $o_2 = x_2 + (y_2 + z_2 \cdot n_x) \cdot n_x$
10: $o_3 = x_3 + (y_3 + z_3 \cdot n_x) \cdot n_x$
11:
12: brick1$[l_z][l_y][l_x]$ = A$[o_1]$; barrier
13: **if** $b_x = b_y = b_z$ **then**
14: A$[o_1]$ = brick1$[l_x][l_z][l_y]$
15: **else**
16: brick2$[l_x][l_y][l_z]$ = A$[o_3]$; barrier
17: A$[o_3]$ = brick1$[l_x][l_y][l_z]$; barrier
18: brick1$[l_x][l_y][l_z]$ = A$[o_2]$
19: A$[o_2]$ = brick2$[l_z][l_x][l_y]$; barrier
20: A$[o_1]$ = brick1$[l_y][l_x][l_z]$
21: **end if**

En definitiva, en la implementación de la transposición in-place zxy se utiliza la misma idea que se ha utilizado en la implementación de la transposición in-place yzx pero en sentido contrario.

4.4. Resultados

Una vez que todas las transposiciones han sido implementadas, se ha analizado su rendimiento en diferentes GPU. El ancho de banda teórico de la memoria de cualquier GPU no es más que un límite teórico superior el cual nunca es alcanzado. Por este motivo, se ha usado el ancho de banda obtenido en la transposición trivial xyz –la copia de una matriz– como base para medir

la eficacia del rendimiento del resto de transposiciones.

La experimentación se ha planificado para analizar el impacto de tres parámetros sobre el rendimiento de una transposición.

En primer lugar, se ha medido el impacto que el número de elementos que transpone cada hilo tiene en el rendimiento de una transposición. A este parámetro se le han asignado tres valores: 1, 2 y 4. De cara a evitar interferencias, se han implementado kernels independientes para cada valor del parámetro.

En segundo lugar, se ha medido el impacto que el tamaño de los datos tiene en el rendimiento de una transposición. Además de realizar transposiciones con números reales, también se ha tenido en cuenta la posibilidad de realizar transposiciones con números complejos, operación que se realiza frecuentemente en el cálculo de FFT. Por consiguiente, en la experimentación se han utilizado dos tamaños de datos: 8 y 16 bytes. El primero podría representar, por ejemplo, a números reales de doble precisión, así como a números complejos de precisión simple. Mientras que el segundo, podría representar a números complejos de precisión doble.

Finalmente, también se ha querido conocer si los resultados varían de una GPU a otra, incluso perteneciendo a la misma arquitectura. Por lo que toda la experimentación se ha repetido para 4 GPU de la arquitectura Fermi: GeForce GTX 550Ti, GeForce GTX 560Ti, Tesla M2050 y Tesla M2090. Las dos GeForce son GPU similares de gama baja con un CC de 2.1, mientras que las dos Tesla son GPU similares de gama alta con un CC de 2.0.

Obviamente, el tamaño de la matriz a transponer también afecta al rendimiento de los resultados, por lo menos para tamaños muy pequeños, y por este motivo la experimentación se ha realizado con un amplio número de tamaños. No obstante, y de cara a limitar la experimentación a un número razonable de pruebas, toda la experimentación se ha realizado con tamaños de matriz cúbicos. En cualquier caso, varias pruebas con matrices no cúbicas han dado resultados similares, por lo que consideramos que los resultados son extrapolables.

El rendimiento se ha obtenido aplicando la fórmula: $p = \frac{2 \cdot N \cdot s}{t}$, donde p es el ancho de banda efectivo en GB/s, N es el número total de elementos, s es el tamaño del elemento básico de la matriz en bytes y t es el tiempo que se necesita para realizar la transposición en ns. El tiempo se computa

mediante eventos CUDA y se calcula realizando la media de 10 ejecuciones del kernel. Dada la estabilidad de los resultados obtenidos, se omite el dato de la varianza. Toda la experimentación se ha realizado usando la versión 5.0 de CUDA.

Consideraciones generales

Como es obvio para cualquiera que conozca la arquitectura CUDA, el kernel encargado de realizar la transposición obtiene mejor rendimiento para los casos donde el tamaño de la dimensión contigua de la matriz es un múltiplo de 16. Este tamaño permite realizar accesos a memoria alineados, y por lo tanto, conseguir una coalescencia completa.

La figura 4.7 muestra el rendimiento obtenido para la transposición *out-of-place* yxz en las GPU Tesla M2090 y GeForce GTX 560Ti cuando 1 elemento de 8 bytes es transpuesto por cada hilo. Los problemas de alineamiento que se han comentado anteriormente son fácilmente perceptibles observando las irregularidades que aparecen en el gráfico y que se repiten para todas las restantes medidas que se han realizado. De hecho, para evitar este tipo de problemas de rendimiento el relleno de datos es muy común en CUDA; y la API de CUDA tiene incluso una función específica para asignar memoria con este objetivo (*cudaMallocPitch*). Por lo tanto, por simplicidad y porque los resultados más interesantes corresponden a las matrices que han sido rellenadas, únicamente se han tenido en cuenta los resultados para tamaños múltiplos de 16.

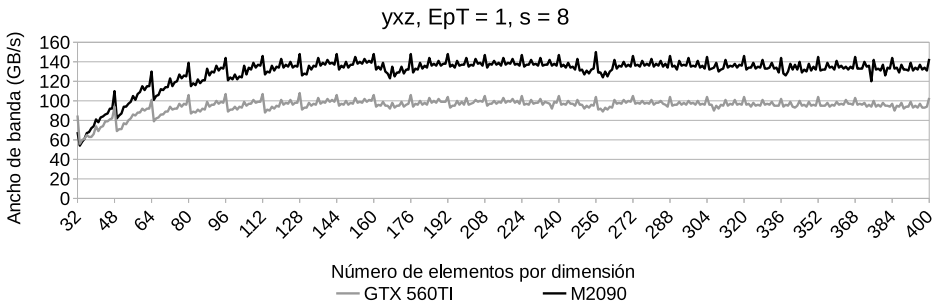

Figura 4.7: Ancho de banda alcanzado para la transposición yxz en las GPU Tesla M2090 y GeForce GTX 560Ti cuando cada hilo transpone un elemento de 8 bytes

Todo gráfico que se ha obtenido, del estilo al que se puede observar en la figura 4.7, presenta un comportamiento similar: el rendimiento es bajo para matrices pequeñas, pero se va incrementando rápidamente junto con el aumento del tamaño de la matriz. Finalmente, el rendimiento se estabiliza cuando la matriz alcanza un tamaño total aproximado de 80^3 o 100^3 elementos.

Este comportamiento general es extremadamente ventajoso en esta sección ya que un sólo valor –el ancho de banda obtenido en la meseta– mantiene la información más importante de la gráfica sin pérdida de información práctica. Para esta simplificación se ha utilizado el valor de la mediana del ancho de banda, que cancela efectivamente los valores bajos de rendimiento para los tamaños pequeños de la matriz y da una buena aproximación del ancho de banda estabilizado.

Transposiciones out-of-place

En esta sección se presentan los resultados correspondientes a las transposiciones *out-of-place*. En la tabla 4.1 se observa el ancho de banda obtenido para la transposición trivial *xyz* –copia de datos– en GB/s. La última columna muestra el ancho de banda teórico para cada GPU. El mejor ancho de banda obtenido para esta transposición es del 85 % del teórico, excepto para la GPU GeForce GTX 550Ti, la cual alcanzó un 70 %. Como se ha dicho, estos valores van a servir de referencia para el resto de las transposiciones.

Tabla 4.1: Ancho de banda obtenido para la transposición *xyz* (copia de datos) para varias GPU y número de elementos. La última columna muestra el ancho de banda teórico para cada GPU. Todos los valores están en GB/s.

	Tamaño del elemento						
	8 bytes			16 bytes			
EpT →	1	2	4	1	2	4	Teór.
GTX 550Ti	69	69	50	64	64	61	98
GTX 560Ti	108	107	89	104	104	102	128
Tesla M2050	125	130	108	112	124	124	148
Tesla M2090	149	149	134	136	137	141	177

Esta tabla también muestra que el parámetro *EpT* –número de elementos

a transponer por cada hilo– afecta al ancho de banda. Por ejemplo, el ancho de banda se degrada para matrices de elementos de 8 bytes y $EpT = 4$, mientras que para elementos de 16 bytes y GPU de alto rendimiento los peores resultados se obtienen cuando $EpT = 1$.

Las figuras 4.8-4.11 muestran el rendimiento obtenido para las 4 GPU que se han utilizado en la experimentación. Cada figura muestra 6 conjuntos de 5 barras donde cada barra representa el ancho de banda obtenido para una particular transposición. Los 6 conjuntos corresponden a los 3 valores del parámetro EpT –1, 2 y 4– y para los dos tamaños de elemento –8 y 16 bytes. La línea superior representa el ancho de banda obtenido por la copia, la cual se ha visto en la tabla 4.1.

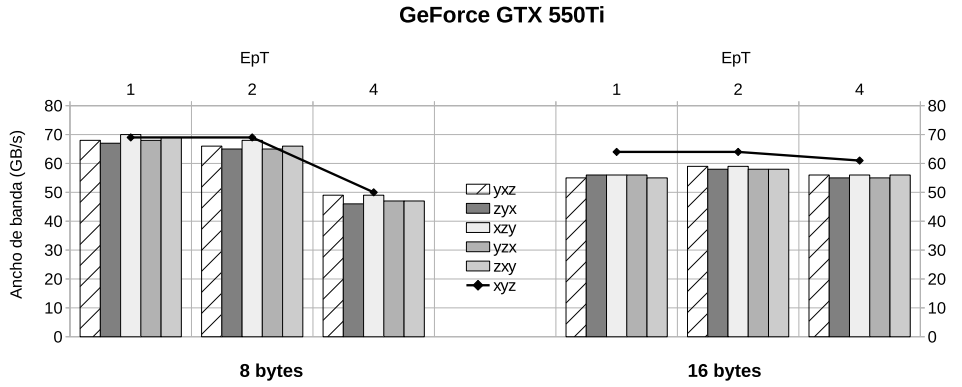

Figura 4.8: Ancho de banda para cualquier transposición out-of-place en la GPU GeForce GTX 550Ti. La línea en la parte más alta representa el límite superior definido por la transposición trivial xyz

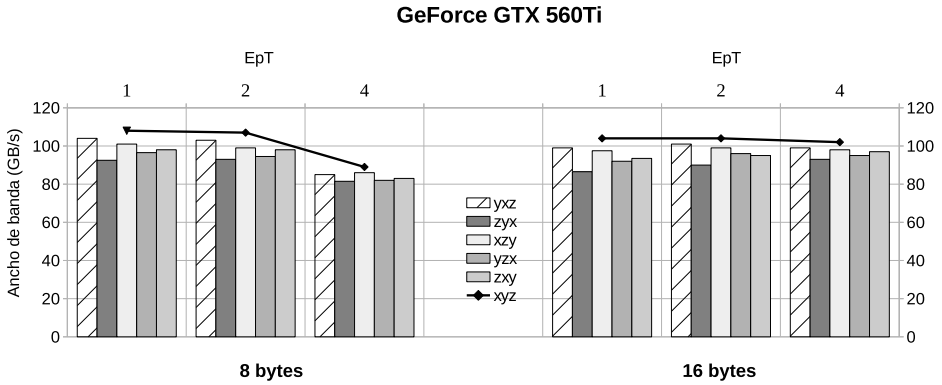

Figura 4.9: Ancho de banda para cualquier transposición out-of-place en la GPU GeForce GTX 560Ti. La línea en la parte más alta representa el límite superior definido por la transposición trivial xyz

En el análisis de los resultados se obtienen unas interesantes conclusiones. En primer lugar, el ancho de banda obtenido es muy cercano al que se obtiene al realizar una simple copia. Si se normalizan los resultados respecto al límite superior, la media para las 5 transposiciones es de 93,4 % con una desviación estándar de sólo 3,6. Por lo tanto, se puede afirmar que para todas las transposiciones que se han implementado se consigue un ancho de banda alto.

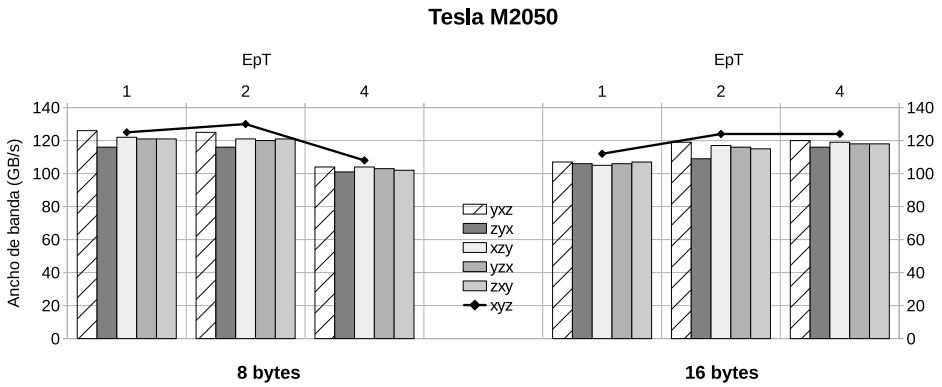

Figura 4.10: Ancho de banda para cualquier transposición out-of-place en la GPU Tesla M2050. La línea en la parte más alta representa el límite superior definido por la transposición trivial xyz

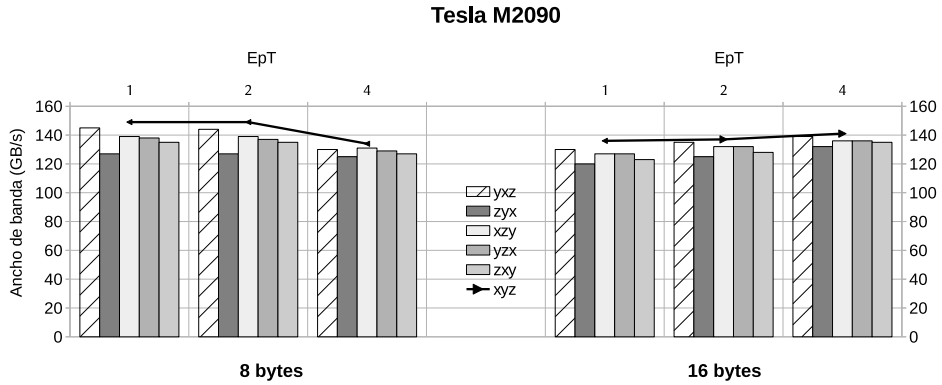

Figura 4.11: Ancho de banda para cualquier transposición out-of-place en la GPU Tesla M2090. La línea en la parte más alta representa el límite superior definido por la transposición trivial xyz

En segundo lugar, el número de elementos a transponer por cada hilo parece que sí afecta al ancho de banda obtenido. Es más, el efecto varía dependiendo del tamaño del elemento de la matriz. Para el caso en el que el tamaño del elemento sea de 8 bytes, hay una clara tendencia a favor de los valores más bajos de EpT. La degradación en el rendimiento es notable cuando $EpT = 4$ para tres de las GPU, pero más leve en el caso de la GPU Tesla M2090. Cuando los elementos son de 16 bytes, no hay una tendencia clara. Para las GPU GeForce, existen ligeras variaciones aunque los efectos no son significativos. Para las GPU Tesla, cuando $EpT = 1$ los resultados ofrecen un rendimiento ligeramente inferior. Como consecuencia, parece que cuando $EpT = 2$ se alcanza, en términos generales, buenos niveles de rendimiento.

En tercer lugar, a pesar del efecto del tamaño del elemento en el comportamiento del parámetro EpT, el efecto del tamaño del elemento en el nivel de rendimiento total es insignificante, a excepción de en la GPU GeForce GTX 550 Ti. En este caso, con elementos de 16 bytes se reduce el ancho de banda obtenido entre un 5 % y un 10 % aproximadamente.

Por último, las 5 transposiciones han mostrado unos resultados ligeramente diferentes. Sin embargo, estas variaciones son tan leves que no afectan prácticamente a los resultados.

Transposiciones in-place

Antes de presentar los resultados para las transposiciones *in-place* es necesario aclarar una serie de cuestiones. Primero, como referencia para el ancho de banda se ha utilizado la transposición trivial *out-of-place*, ya que la transposición trivial *in-place* no realiza ningún movimiento de datos. Segundo, nótese que para este caso, cada hilo transpone 2 o 3 elementos –permuta elementos para las transposiciones de involución o rota elementos para las transposiciones de rotación. Estas modificaciones interfieren con el parámetro EpT, y en unos resultados preliminares se ha observado que el efecto de este parámetro se reduce considerablemente para las transposiciones *in-place*. Por este motivo, los resultados que se van a presentar en esta sección corresponden a la experimentación cuando los hilos realizan una única permutación o rotación.

La figura 4.12 muestra los resultados correspondientes a las transposiciones *in-place* en las 4 GPU. En un primer vistazo se puede observar que las transposiciones de involución mantienen un nivel alto en el rendimiento –entorno a un 90,5 % del ancho de banda de la copia–, mientras que el rendimiento para las transposiciones de rotación desciende hasta el 60,8 %. De todos modos, este descenso en el rendimiento era esperado, ya que al pasar de la estrategia de baldosas (2D) a ladrillos (3D) asumíamos una reducción del nivel de coalescencia. De todos modos, se mejora la primera implementación realizada mediante baldosas (2D) obteniendo un rendimiento 2 o 3 veces superior. Todas las transposiciones de involución muestran un rendimiento similar, aunque la transposición zyx nuevamente es ligeramente más lenta que el resto.

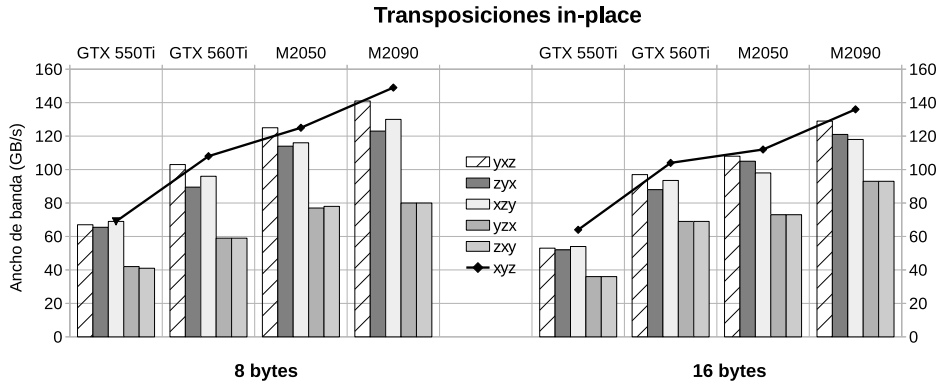

Figura 4.12: Ancho de banda para todas las transposiciones in-place y GPU. La línea en la parte más alta representa el límite superior definido por la transposición trivial out-of-place xyz

Por otro lado, la pérdida del rendimiento en la GPU GeForce GTX 550 Ti para transposiciones *out-of-place* y elementos de 16 bytes se amplía hasta un 17 % para las transposiciones *in-place*. Incluso la Tesla M2050 también muestra un comportamiento similar, aunque en menor grado (entorno al 11 %).

4.5. Conclusiones

En este capítulo se ha presentado una implementación para transponer de manera eficiente matrices de 3D en una GPU. Aunque las transposiciones se basan en las transposiciones de matrices en 2D, que ya habían sido resueltas de manera eficiente, la particular arquitectura de las GPU ha hecho que el presente trabajo no sea una tarea trivial. No obstante, el trabajo experimental demuestra que para todas las posibles transposiciones *out-of-place* se consigue un rendimiento alto.

También se ha descubierto la importancia del número de elementos a transponer por cada hilo (EpT). Éste es un parámetro que afecta sensiblemente al rendimiento de las transposiciones y que depende del tamaño de los elementos de la matriz y de la GPU donde se ejecuta la transposición. Sin embargo, la experimentación que se ha realizado para 2 tamaños de elementos y 4 GPU con la arquitectura Fermi, muestran que transponer 2 elementos

por cada hilo es una buena opción.

Los otros parámetros, como el tipo de la transposición que se va a ejecutar o el número de baldosas a transponer por bloque, no muestran variaciones significativas en el rendimiento.

Las transposiciones *in-place* plantean un problema más complejo y están fuera del ámbito de este trabajo. Sin embargo, se han implementado una serie de transposiciones *in-place* básicas, que en la mayoría de los casos, también han demostrado un rendimiento alto. La implementación eficiente de las dos transposiciones de rotación no ha sido un trabajo fácil, reduciendo, el rendimiento obtenido un 30 %. En cualquier caso, se ha duplicado el rendimiento en comparación con la implementación básica basada en baldosas que se había realizado inicialmente.

En el capítulo 3 se ha analizado el rendimiento que ofrece la librería cuFFT de NVIDIA. Una de las principales opciones que se ha analizado ha sido aquella que permite calcular la FFT con matrices de entrada y salida con datos no contiguos. En el presente capítulo se ha presentado una manera para realizar transposiciones en 3D de manera eficiente [108]. Gracias a esta técnica se ha realizado un análisis preliminar para saber si el cálculo de la FFT con datos contiguos, tras transponer los datos de entrada, puede superar el rendimiento en el cálculo de la misma FFT con datos no contiguos. Un análisis rápido de los resultados obtenidos indican que puede ser beneficioso transponer los datos mediante un algoritmo eficiente para calcular FFT con datos contiguos, frente a hacer uso de la opción ***stride*** de la librería cuFFT.

Capítulo 5

Resolución eficiente de la ecuación de Poisson en un clúster GPU

Índice

5.1. Introducción

En el presente capítulo se presenta la propuesta de solución para la resolución eficiente de la ecuación de Poisson en un clúster de GPU.

El algoritmo que se ha implementado resuelve la ecuación de Poisson en tres dimensiones y con unas condiciones de contorno periódicas mediante el uso de transformadas (FFT) en un clúster de GPU.

5.2. La ecuación de Poisson

Tal y como se ha explicado en el capítulo 2 (apartado 2.3.3), la ecuación de Poisson se define de la siguiente manera:

$$\Delta\phi = f$$

donde Δ es el operador laplaciano, y ϕ y f son dos funciones reales o complejas.

Como el objetivo consiste en resolver la ecuación de Poisson en tres dimensiones, se puede particularizar a 3 dimensiones y coordenadas cartesianas del siguiente modo:

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\right)\phi(x,y,z) = f(x,y,z)$$

Es decir, f es la función que obtenemos mediante la suma de las derivadas parciales segundas de ϕ.

Para resolver la ecuación de Poisson hay que obtener ϕ a partir de una f conocida y una condición de contorno determinada.

El cálculo directo de esta ecuación requiere $O(N^2)$ operaciones, donde N es el número de puntos de la malla. En este trabajo de investigación se ha utilizado un método basado en FFT, que ha demostrado una buena capacidad de escalado en un clúster con gran cantidad de nodos [5].

La solución a la ecuación de Poisson mediante FFT viene dada por la siguiente ecuación:

$$\phi(r) = \hat{\phi}^{-1}(\hat{f}(k)/|k|^2)$$

donde $\hat{f}(k)$ es la FT de f y $\hat{\phi}^{-1}$ es la IFT de ϕ.

La resolución de esta ecuación se implementa en un algoritmo de tres pasos:

1. calcular la FFT de la función f,
2. resolver la ecuación en el espacio frecuencial dividiendo cada punto por el cuadrado del módulo de sus coordenadas, y
3. calcular la IFT del resultado.

A menudo el algoritmo se completa mediante una etapa de normalización requerida por muchas implementaciones de la FFT y se calcula dividiendo cada punto por N.

Hay que señalar que este método funciona si las condiciones de contorno son periódicas para cada dimensión, condición que se cumple en la definición del objetivo para este capítulo.

5.3. Implementación

Tal y como se ha mencionado anteriormente, la solución propuesta para resolver la ecuación de Poisson se basa en el algoritmo FFT. Por lo que una de las partes clave de esta solución consiste en la ejecución paralela de transformadas en varias GPU. Este paralelismo se consigue mediante una descomposición por planos (*slab*). Esta descomposición distribuye toda la matriz de datos de entrada entre las GPU existentes, mediante la asignación de un conjunto de planos completos a cada una de ellas. Cada GPU calcula las FFT 2D correspondientes a todos los planos que ha recibido y a continuación, mediante la comunicación MPI *Alltoall*, los datos se vuelven a redistribuir entre todas las GPU. Finalmente, cada GPU ejecuta FFT 1D en cada fila de la dimensión no afectada en la anterior FFT 2D.

5.3.1. Aproximación secuencial

La primera aproximación a la solución final, consiste en realizar una implementación simple que calcula los 4 pasos definidos en la sección 5.2 de manera secuencial.

En primer lugar, se calcula la FFT directa en 2D, seguida de la copia de datos entre la GPU y la CPU, la comunicación MPI *Alltoall* inter-nodo, la

copia de datos entre la CPU y la GPU, y por último, el cálculo de FFT 1D. En segundo lugar, se calcula la ecuación de Poisson en el espacio frecuencial. En tercer lugar, se calcula la FFT inversa, la cual es simétrica a la calculada en el primer paso. Finalmente, se normalizan los resultados.

La comunicación inter-nodo se realiza mediante una única llamada a la función MPI *Alltoall*, siempre y cuando los datos se estructuren mediante la creación de tipos de datos MPI adecuados. Estos tipos de datos MPI definen la posición exacta de los datos para antes y después de la comumicación. Sin embargo, se ha observado que esta solución genera mucha sobrecarga ya que obliga a la implementación MPI a realizar complejas operaciones de redistribución de datos, haciendo que la comunicación inter-nodo sea de un orden de magnitud más lento que el resto de operaciones. Como consecuencia, se ha modificado la estrategia de la implementación y se han añadido transformaciones intermedias de datos antes y después de la comunicación inter-nodo. Gracias a estas transposiciones la comunicación MPI se realiza sobre datos contiguos. Para realizar transposiciones de manera eficiente, tal y como se ha recogido en el capítulo 4, se puede aprovechar el mayor ancho de banda de las memorias de las GPU siempre y cuando la implementación tenga en cuenta las particularidades de la arquitectura de la GPU en la que se va a ejecutar [108].

5.3.2. Aproximación segmentada

Desafortunadamente, esta solución secuencial infrautiliza el hardware del que se dispone para la experimentación. En particular, las GPU que se utilizan están provistas de 2 unidades de transferencias de datos, por lo que son capaces de realizar copias bidireccionales entre la CPU y la GPU. Gracias a esta característica nuestro hardware permite el solapamiento de los siguientes tipos de operaciones: ejecución de kernel, copias de datos de GPU a CPU, copias de datos de CPU a GPU y comunicación inter-nodo. Es obvio que una implementación eficiente debe aprovechar esta característica y ejecutar todas estas operaciones simultáneamente maximizando el grado de paralelismo. Con este objetivo en mente, se ha vuelto a diseñar el primer algoritmo para implementar una versión segmentada del mismo. Esta nueva versión segmentada de la implementación mantiene la estructura de la anterior, pero modifica ligeramente las operaciones que se ejecutan en cada fase.

Para la primera fase se ha diseñado un esquema de segmentación (*pipeline*)

compuesto por las siguientes operaciones: calcular la FFT 2D, transponer los datos antes de enviarlos, copia de los datos entre la GPU y la CPU, comunicación MPI *Alltoall* inter-nodo, copia de los datos de CPU a GPU, y por último, una última transposición de los datos para devolverlos a su posición inicial. Todas las operaciones mencionadas anteriormente pueden ejecutarse paralelamente en esta primera fase.

En la figura 5.1 se puede observar el *pipeline* propuesto para esta primera fase. En la primera fila se encuentran los *kernels* que se ejecutan en la GPU. En las filas segunda y cuarta se observan las copias de datos –del dispositivo al host (D2H) y del host al dispositivo (H2D)– realizadas por las dos unidades de transferencia de datos disponibles. Por último, en la tercera fila se observa la comunicación de datos inter-nodo (comunicación MPI). Las celdas oscurecidas muestran los estados necesarios para procesar el segmento i.

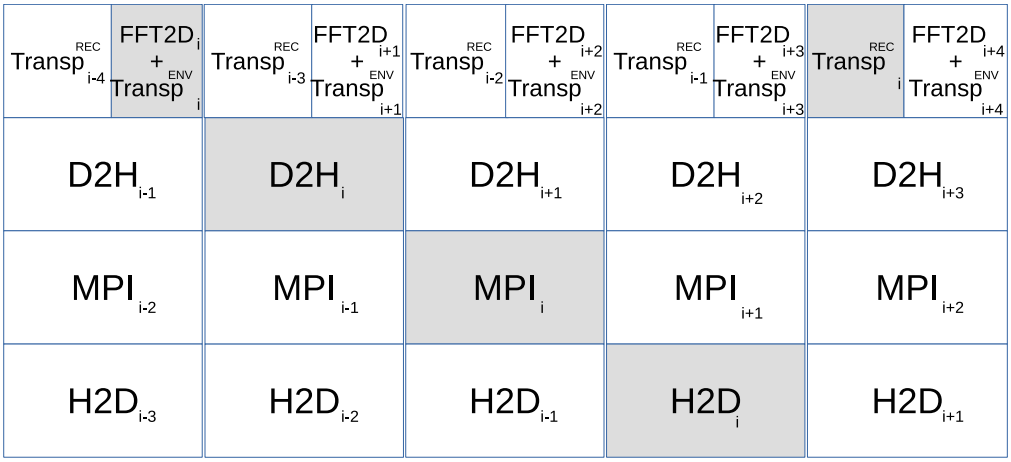

Figura 5.1: Esquema de segmentación que se aplica a la ejecución de la primera fase del algoritmo.

La sincronización entre las diferentes operaciones no es una tarea trivial. Nótese que se deben sincronizar operaciones gestionadas por el planificador de la GPU –ejecución de *kernels* y copia de datos– y operaciones gestionadas por los hilos de la CPU –la comunicación MPI. Para alcanzar el nivel de sincronización necesario se ha hecho uso de varios CUDA *streams*.

Las operaciones que se ejecutan en un único CUDA *stream* se ejecutan de manera secuencial, mientras que las operaciones que se encuentran en diferentes CUDA *streams* pueden ejecutarse en paralelo. Para poder sincroni-

zar las operaciones $FFT2D + Transp^{ENV}$ con la copia de datos $D2H$ son necesarios 2 *streams*: los segmentos pares se envían a un *stream*, mientras que los segmentos impares se envían a otro *stream*. Por lo tanto, los *kernels* y las copias de datos pertenecientes al mismo *stream* se ejecutan de manera secuencial, pero pueden ejecutarse en paralelo junto con los *kernels* y las copias de datos de los segmentos vecinos. La misma estrategia se sigue con la copia de datos $H2D$ y la ejecución del *kernel* $Transp^{REC}$. Para sincronizar la copia de datos $D2H$ y la comunicación MPI se hace uso de los eventos de CUDA. En el algoritmo 12 se muestra la implementación de este *pipeline*. El bucle principal está precedido por un prólogo donde se encolan las primeras operaciones para los segmentos 0 y 1. El código de este prólogo se ha omitido por claridad, ya que es el mismo código que se encuentra entre las líneas 6 y 9. En el código de estas líneas se encolan 3 operaciones a ejecutar en GPU: calcular la FFT 2D, transponer los datos antes de su envío y copiar los datos de la GPU a la CPU. A continuación, se registra un evento CUDA para sincronizar la copia de datos con su comunicación MPI. Una vez que estas operaciones han sido encoladas, el hilo de la CPU se bloquea hasta que se genere el correspondiente evento CUDA. Es entonces cuando se ejecuta la llamada síncrona MPI *Alltoall* y el código del prólogo se repite para el siguiente segmento. En último lugar se encuentran las operaciones que deben ejecutarse sobre los datos que se han recibido –copiar los datos a la GPU y transponerlos–, que son encolados en el *stream* correspondiente.

Algoritmo 12 Primera fase de la implementación segmentada.

```
1: Prólogo de los segmentos 0 y 1 (Líneas 6 - 9)
2: for i = 0 to #segs −1 do
3:    Esperar al evento de CUDA (i)
4:    MPI_Alltoall (i)
5:    if i ≤ #segs −3 then
6:       Calcular la FFT 2D (i + 2)
7:       Transponer antes de enviar los datos (i + 2)
8:       Copiar los datos D2H (i + 2)
9:       Registrar el evento CUDA (i + 2)
10:   end if
11:   Copiar los datos H2D (i)
12:   Transponer tras recepción de datos (i)
13: end for
```

En la segunda fase se ejecutan las siguientes operaciones en la GPU: calcular la FFT 1D directa, resolver la ecuación de Poisson en el espacio

frecuencial y calcular la FFT 1D inversa.

Este segundo grupo de operaciones no pueden ser solapadas con el primer grupo de operaciones, ya que la ejecución de la FFT 1D no puede realizarse hasta que todas las FFT 2D hayan finalizado[1]. De todos modos, todas estas operaciones no podrían ejecutarse de manera paralela, ya que todas ellas utilizan el mismo hardware. Sin embargo, una característica que ha sido añadida recientemente a la librería cuFFT (llamada *callback routines*), de alguna manera podría acelerar este proceso ya que permite definir una función previa o posterior a la FFT sin necesidad de tener que leer o escribir los datos en memoria. No obstante, debido a que es una característica en estado experimental y que tras analizar su funcionamiento se ha visto que no aporta grandes beneficios a nuestra implementación, no ha sido incluida en este trabajo.

La última fase es inversa a la primera fase, con la diferencia de que en esta fase se incluye una operación final de normalización. A pesar de que la operación de normalización no puede ser ejecutada en paralelo junto con el resto de kernels (para el cálculo de la FFT 2D inversa o para transponer los datos), vale la pena incluirla en el algoritmo segmentado. De este modo, si las ejecuciones de los kernels no son las operaciones que más tiempo consumen –tal y como se ha comprobado en experimentos anteriores– todas ellas pueden ser solapadas con tareas más lentas.

En la figura 5.2 se puede observar el *pipeline* propuesto para esta última fase. En la primera fila se encuentran los *kernels* que se ejecutan en la GPU. En las filas segunda y cuarta se observan las copias de datos –del dispositivo al host (D2H) y del host al dispositivo (H2D)– realizadas por las dos unidades de transferencia de datos disponibles. Por último, en la tercera fila se observa la comunicación de datos inter-nodo (comunicación MPI). Debido a su similitud con la figura 5.1 y por claridad, sólo se ha representado una etapa del *pipeline* de la última fase.

[1]En realidad, esto no es exacto ya que la FFT 1D se puede descomponer y algunas de sus operaciones podrían ejecutarse antes. No obstante, un rápido análisis ha mostrado que los beneficios potenciales no compensan la sobrecarga en las operaciones de control adicionales.

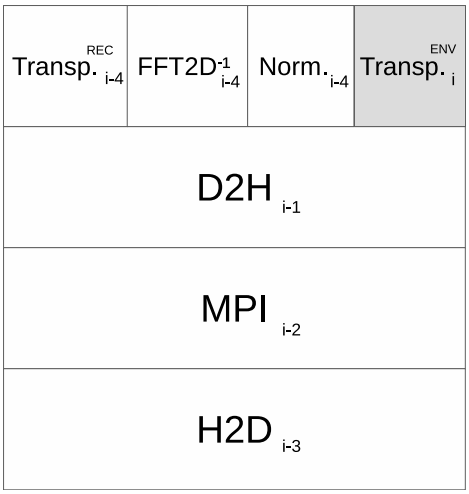

Figura 5.2: Esquema de segmentación que se aplica a la ejecución de la última fase del algoritmo.

En el algoritmo 13 se muestra la implementación de este *pipeline*. El bucle principal está precedido por un prólogo donde se encolan las primeras operaciones para los segmentos 0 y 1. El código de este prólogo se ha omitido por claridad, ya que es el mismo código que se encuentra entre las líneas 6 y 8. En el código de estas líneas se encolan 2 operaciones a ejecutar en GPU: transponer los datos antes de enviarlos y copiar los datos de la GPU a la CPU. A continuación, se registra un evento CUDA para sincronizar la copia de datos con su comunicación MPI. Una vez que estas operaciones han sido encoladas, el hilo de la CPU se bloquea hasta que se genere el correspondiente evento CUDA. Es entonces cuando se ejecuta la llamada síncrona MPI *Alltoall* y el código del prólogo se repite para el siguiente segmento. En último lugar se encuentran las operaciones que deben ejecutarse sobre los datos que se han recibido –copiar los datos a la GPU, transponerlos, cálculo de la FFT 2D inversa y la normalización– y se encolan en el *stream* correspondiente.

Algoritmo 13 Tercera fase de la implementación segmentada.

```
1: Prólogo del segmento 0 y 1 (Líneas 6 y 8)
2: for i = 0 to #segs −1 do
3:    Esperar al evento de CUDA (i)
4:    MPI_Alltoall (i)
5:    if i ≤ #segs −3 then
6:       Transponer antes de enviar los datos (i + 2)
7:       Copiar los datos D2H (i + 2)
8:       Registrar el evento CUDA (i + 2)
9:    end if
10:   Copiar los datos H2D (i)
11:   Transponer tras recepción de datos (i)
12:   Calcular la FFT 2D inversa (i)
13:   Normalización (i)
14: end for
```

Nótese que la FFT 2D debe operar en todo un plano de la malla (*grid*), y por lo tanto define el grano más fino posible en nuestra segmentación. Como consecuencia, el tamaño de segmento de nuestra aplicación se define mediante el número de planos y ésta es la convención que se va a seguir en el resto de este capítulo.

5.4. Resultados

5.4.1. Metodología experimental

Toda la experimentación se ha realizado en un clúster Bull [109]. El clúster dispone de 9 nodos equipados con GPU. Cada nodo dispone de 2 CPU Intel Xeon E5645 con 6 núcleos cada uno, 2 GPU Tesla M2090 (mismo hardware que el usado en la experimentación del capítulo 3) y 2 interfaces QDR 4x de InfiniBand (IB) [110, 111]. Las CPU disponen de 24 GB de RAM por nodo, mientras que cada GPU dispone de 6 GB de memoria. También se ha utilizado el compilador de Intel en la versión 14.0.2, la versión de OpenMPI 1.6 optimizada para Bull [112] y el controlador 340.29 de CUDA [113]. Todos los datos se representan como números reales de coma flotante de precisión doble.

En la figura 5.3 se detalla la arquitectura de cada nodo del clúster. Cada nodo usa un *hub* de Entrada/Salida (*I/O Hub*) para conectarse al bus PCIe 2.0 que da acceso a la GPU y a la interfaz IB. Esta estructura penaliza la comunicación interna entre las GPU, ya que no permite que la tecnología CUDA *peer-to-peer* –también conocida como GPUDirect v2– se lleve a cabo. Sin embargo, provee una comunicación completamente paralela entre cada grupo CPU-GPU-IB. Desafortunadamente, el hardware que se ha descrito –en particular las GPU y las interfaces IB– no aprovecha las características que ofrece la nueva tecnología GPUDirect RDMA de CUDA, la cual permite una comunicación directa entre la GPU y la interfaz IB.

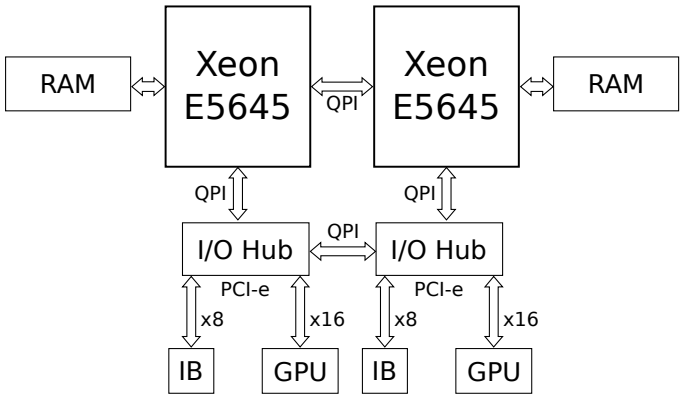

Figura 5.3: Arquitectura de cada nodo del clúster.

Para la experimentación se han utilizado matrices cúbicas de 256^3, 512^3 y 1024^3 elementos. El tamaño de cada matriz es de 128 MB, 1 GB y 8 GB, respectivamente. Debido al tamaño de la matriz más grande no siempre se ha podido calcular la solución con el hardware disponible.

Un aplicación científica que entre sus operaciones requiera resolver la ecuación de Poisson, por lo general, ejecuta este cálculo intercalándolo con otras operaciones. Como estas operaciones también se calculan en la GPU no hay necesidad de transferir los datos para resolver la ecuación de Poisson entre la CPU y la GPU. Por este motivo, los resultados que se van a mostrar no incluyen las transferencias de datos iniciales y finales entre la CPU y la GPU. Como es natural, el coste de toda transferencia de datos entre la CPU y la GPU necesaria para realizar a la comunicación MPI ha sido incluida en los resultados.

5.4.2. Resultados para un único nodo

Aunque este trabajo se centra en la resolución eficiente de la ecuación de Poisson basado en el algoritmo FFT en un clúster GPU, consideramos importante analizar los resultados correspondientes a un sólo nodo. Por este motivo, se ha implementado una versión con un único proceso –sin comunicación MPI– que gestiona varias GPU en un único nodo. En particular, en esta implementación se lanza un hilo de ejecución por cada GPU del nodo. Desafortunadamente, tal y como se ha comentado anteriormente, el hardware de nuestro clúster no permite una comunicación directa entre las GPU del nodo, por lo que las GPU deben comunicarse entre sí mediante la memoria de la CPU, lo que conduce a una solución ineficiente.

Aunque en la implementación realizada no se superponen cómputo y comunicación, nos da una valiosa información acerca de la importancia de la comunicación directa con la GPU.

La tabla 5.1 muestra los resultados correspondientes a un sólo nodo. Se muestran los resultados correspondientes a 1 CPU, 2 CPU, 1 GPU y 2 GPU. La solución propuesta para CPU se basa en una implementación multihilo de la librería FFTW (versión 3.3.3) y hace uso de todos los núcleos disponibles en cada CPU (6 nucleos por cada CPU).

Tabla 5.1: Tiempo de ejecución (ms) para la implementación en un único nodo. La última columna muestra la aceleración obtenida entre el mejor tiempo de ejecución en GPU respecto al mejor tiempo de ejecución en CPU.

N	CPU		GPU		Aceleración
	1	2	1	2	
256^3	224	267	21	50	10,67
512^3	2128	1858	218	433	8,52

Los resultados muestran que la implementación en GPU resuelve de manera eficiente la ecuación de Poisson. En el hardware que se ha descrito se consigue resolver la ecuación 10 veces más rápido en una GPU que en una CPU. El cálculo de la FFT corresponde, más o menos, al 85 % del tiempo total para una CPU y una GPU. Sin embargo, cuando se hace uso de dos GPU alrededor del 75 % del tiempo se invierte en la fase de comunicación

de los datos, por lo que es más recomendable usar una única GPU aunque dispongamos de 2 GPU.

El uso de dos CPU no ha supuesto ninguna mejora sustancial respecto al uso de una única CPU. Se baraja la hipótesis de que la ausencia de una mejora sustancial en el tiempo de ejecución pudiera deberse a un problema de saturación del bus de acceso a memoria, ya que el cálculo de la FFT hace un uso intensivo de la memoria. Sin embargo, no se ha profundizado en esta cuestión, ya que está fuera del ámbito de este trabajo.

Aunque la implementación en la CPU ha sido capaz de resolver una matriz con 1024^3 elementos, no ha sido posible hacer lo mismo en una GPU, debido a las limitaciones de memoria. Sin embargo, la gran cantidad de memoria requerida para almacenar la matriz de datos parece que produce algún tipo de problema de paginación en la memoria de la CPU, ya que el tiempo de ejecución aumenta a 52 438 ms para el cálculo en una CPU y a 35 597 ms para el cálculo en dos CPU –lo que supone un tiempo de dos a tres veces superior a lo esperable.

5.4.3. Resultados para múltiples nodos

En esta sección se van a mostrar los resultados correspondientes a la implementación MPI que se ha desarrollado para resolver la ecuación de Poisson en un clúster de GPU. La primera parte de esta sección está dedicada a examinar el comportamiento de la implementación no segmentada para calcular la ecuación de Poisson, ya que esto nos facilitará el posterior análisis de la versión segmentada; mientras que la segunda parte de esta sección está dedicada a la implementación segmentada.

Aproximación no segmentada

La figura 5.4 muestra el tiempo requerido por la implementación no segmentada que se ha desarrollado para resolver la ecuación de Poisson para diferentes tamaños de datos de entrada y para 2, 4, 8 y 16 procesos MPI. En concreto, se lanzan dos procesos MPI por cada nodo –uno por cada CPU– y donde cada uno de ellos controla una GPU. Conviene recordar que esta implementación no solapa ni cálculos ni comunicación. Como referencia, las

líneas discontinuas muestran el tiempo de ejecución de los resultados para un nodo de la tabla 5.1. Los resultados que se muestran en esta figura, en comparación con los resultados presentados en la tabla 5.1, muestran que el uso de la comunicación MPI supone una penalización muy alta, ya que la ejecución con dos procesos MPI es hasta 5 veces más lenta que la ejecución con un único proceso (valores representados en la figura mediante líneas grises discontinuas). Sin embargo, la figura muestra una escalabilidad muy buena hasta 16 GPU –incluso llega a ser superlinear hasta 8 GPU–, donde la ecuación de Poisson se resuelve para datos de entrada de 512^3 elementos en 128 ms. Otro beneficio de usar varias GPU es que permite calcular la ecuación de Poisson en 8 y 16 GPU para datos de entrada de 1024^3 elementos, lo cual no era posible en la implementación de un sólo nodo. En este caso, el tiempo de ejecución ha sido de 2770 ms para 8 GPU y de 1129 ms para 16 GPU.

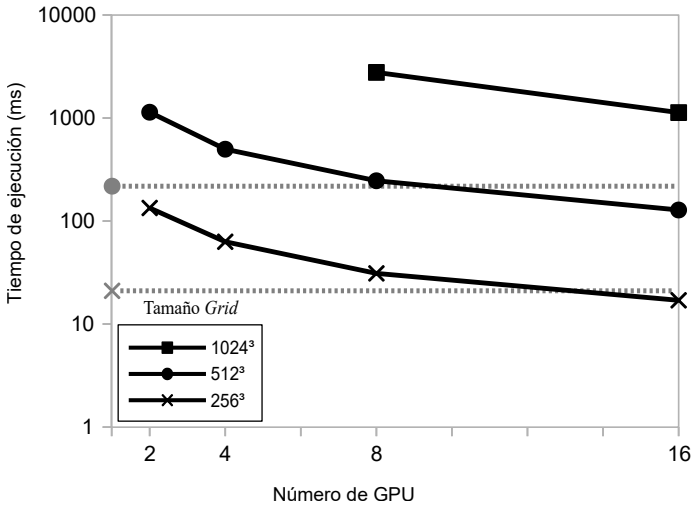

Figura 5.4: Tiempo para resolver la ecuación de Poisson de manera no segmentada en 2, 4, 8 y 16 GPU y con tamaños del *grid* de datos de entrada de 256^3, 512^3 y 1024^3 elementos.

En la figura 5.5 se muestra visualmente y en la tabla 5.2 se puede analizar numéricamente la descomposición del tiempo de ejecución requerido para resolver la ecuación de Poisson para datos de entrada de 512^3 elementos en 2, 4, 8 y 16 GPU. Se puede ver que aproximadamente la mitad del tiempo total se invierte en la comunicación MPI, mientras que una tercera parte se invierte en la transferencia de datos entre la CPU y la GPU.

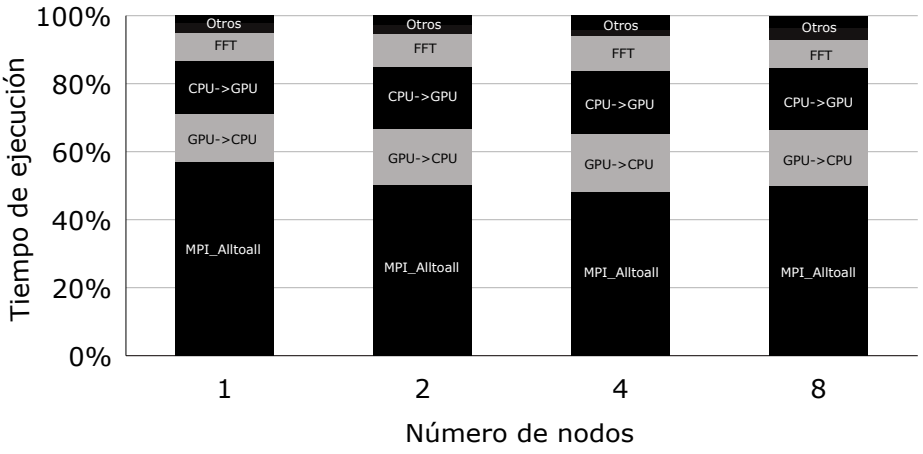

Figura 5.5: Descomposición del tiempo de ejecución requerido por el algoritmo no segmentado para resolver la ecuación de Poisson para datos de entrada de 512^3 elementos en 2, 4, 8 y 16 GPU. Las etapas que se muestran corresponden a: comunicación MPI, transferencia de datos entre CPU/GPU, cálculo de la FFT y ejecución de otros *kernels* (transposción de datos, resolver la ecuación de Poisson en el campo frecuencia y normalización de los resultados).

Estos resultados confirman el hecho ya esperado de que el cálculo de la FFT en paralelo está dominado por el componente de la comunicación, y que en este caso, se ve agravado por la comunicación adicional entre la CPU y la GPU.

Debido a que la comunicación MPI es la tarea dominante en la ejecución de este algoritmo, se ha analizado con mayor profundidad. En la figura 5.6 se muestra cómo el ancho de banda obtenido por la comunicación MPI *Alltoall* varía, en nuestra máquina, dependiendo del tamaño del mensaje y del número de procesos MPI. Cuando se hace referencia al tamaño del mensaje, nos referimos al número total de bytes enviados por cada proceso MPI. Se observa cómo el ancho de banda crece a la par que crece el tamaño del mensaje, hasta que se alcanza una meseta. La reducción del ancho de banda que se observa al pasar de tamaños medios de mensaje a tamaños más grandes de mensaje, puede deberse al hecho de que las diversas implementaciones MPI existentes utilizan estrategias diferentes para tamaños de mensaje diferentes [114].

Tabla 5.2: Descomposición del tiempo de ejecución (ms) requerido por el algoritmo no segmentado para resolver la ecuación de Poisson para datos de entrada de 512^3 elementos en 2, 4, 8 y 16 GPU. Las etapas que se muestran corresponden a: la comunicación MPI, transferencia de datos entre CPU/GPU, cálculo de la FFT, cálculo de las transposiciones y la ejecución de otros *kernels* (resolver la ecuación de Poisson en el espacio frecuencial y la normalización de los resultados).

	# de GPU			
	2	4	8	16
MPI_Alltoall	653	249	117	63
D2H	164	82	41	21
H2D	179	90	45	23
FFT	95	48	25	11
Transposiciones	40	18	10	6
Otros	17	8	4	2
Total	1148	495	242	126

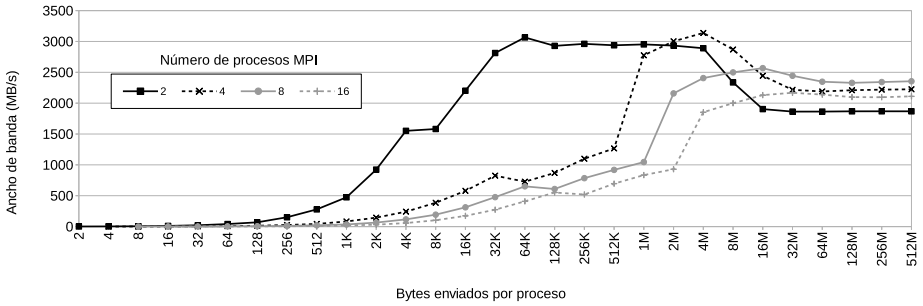

Figura 5.6: Ancho de banda efectivo del *Alltoall* para diferentes tamaños de mensaje y número de procesos.

Como el tamaño más pequeño del *grid* de los datos de entrada es de 256^3 elementos, el tamaño de los mensajes MPI tienen un tamaño mínimo de 8 MB. Por lo que, en base al rango de datos que se está utilizando en la experimentación, estamos principalmente en anchos de banda de la meseta. Estos valores del ancho de banda en la meseta explican el comportamiento superlinear de la implementación hasta 8 GPU, ya que la comunicación MPI es más rápida hasta alcanzar esos valores.

En particular, la figura 5.6 muestra que el ancho de banda obtenido en la meseta para 2 procesos MPI en la comunicación *Alltoall* es de unos 1,9 GB/s. Este ancho de banda aumenta hasta 2,2 GB/s y 2,3 GB/s para 4 y 8 procesos respectivamente. Finalmente, el ancho de banda correspondiente a la comunicación *Alltoall* para 16 procesos MPI es de 2,1 GB/s.

Debido a la importancia que tiene la comunicación MPI, se ha considerado ejecutar el código creado con otras implementaciones MPI. En particular, se han realizado pruebas con las implementaciones de OpenMPI 1.8.4 y MVAPICH2 2.0.1. La figura 5.7 muestra el ancho de banda alcanzado para cada implementación MPI con tamaño de mensajes de 128 MB (tamaño de mensaje de la meseta), mientras que la figura 5.8 muestra el tiempo de ejecución para resolver la ecuación de Poisson para un *grid* de datos de entrada de 256^3 elementos.

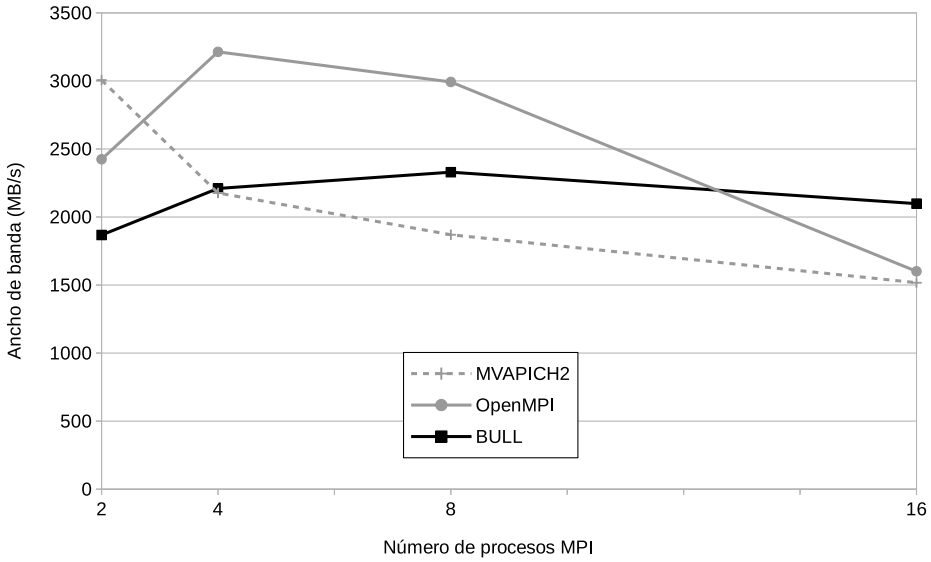

Figura 5.7: Ancho de banda del *Alltoall* para tamaños de mensaje de 128 MB.

Ambas figuras son coherentes ya que una comunicación MPI más rápida implica una resolución de la ecuación más eficiente. Aunque en los casos en los que se usan 2 y 4 procesos las implementaciones MPI de MVAPICH2 y de OpenMPI son más rápidas, la única que mantiene un ancho de banda estable para 8 y 16 procesos es la implementación de Bull. También se ha probado con diferentes tamaños de datos de entrada, obteniendo unas conclusiones similares. Por lo que, habiendo analizado estos resultados, se ha decidido

continuar la experimentación con la implementación MPI de Bull, y a partir de ahora todos los resultados van a estar basados en esta implementación.

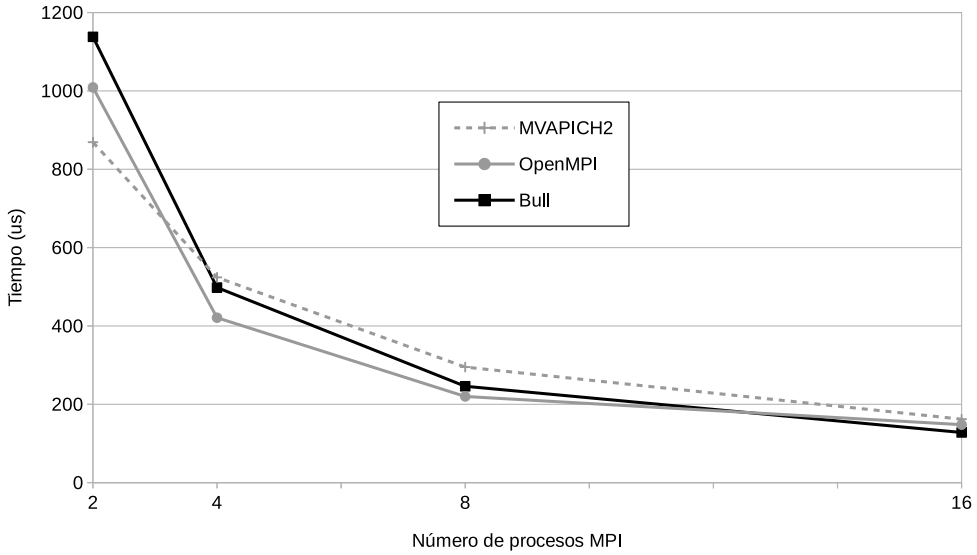

Figura 5.8: Tiempo de ejecución para resolver la ecuación de Poisson para tamaño de *grid* de datos de entrada de 512^3 elementos y para diferentes implementaciones MPI.

Aproximación segmentada

A continuación se va a analizar la ejecución segmentada, donde el cálculo de la FFT, la comunicación entre la CPU y la GPU y la comunicación entre procesos se ha solapado cuando ha sido posible.

Un factor importante a tener en cuenta es la cantidad de datos que se procesa en cada fase. Tal y como se ha dicho anteriormente el tamaño mínimo es un plano, pero en ocasiones puede ser conveniente agrupar estos planos para poder procesar datos de mayor tamaño. A ese conjunto de planos se le ha llamado segmento y su tamaño se define en número de planos. Hay que tener en cuenta que el número de planos es finito y definir tamaños de segmento muy grandes resultará en pocos segmentos.

Las figuras 5.9, 5.10 y 5.11 muestran el tiempo necesario (en ms) para resolver la ecuación de Poisson para diferentes tamaños de segmento y con

tamaños de datos de 256^3, 512^3 y 1024^3 elementos respectivamente. En estas últimas tres figuras se incluye el tiempo de ejecución de la versión no segmentada (línea gris más clara discontinua) como referencia de la mejora obtenida.

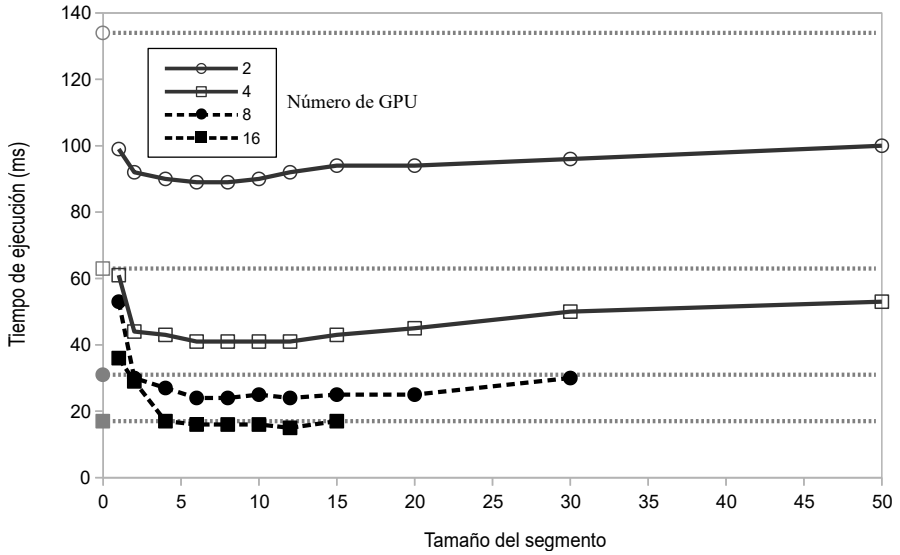

Figura 5.9: Tiempo de ejecución (ms) para un *grid* de 256^3 elementos y 2, 4, 8 y 16 GPU dependiente del tamaño de segmento. El tamaño de segmento viene dado en número de planos.

En términos generales, todas las líneas poseen una ligera forma de una curva cóncava. Este hecho es fácilmente explicable por dos factores: (a) para tamaños de segmento pequeños la comunicación MPI es ineficiente y (b) para tamaños de segmento grandes el número de segmentos no es suficiente para un *pipeline* eficiente.

El primer factor es respaldado por los resultados mostrados en la figura 5.6. Nótese que el tamaño del plano es de $129 \times 256 \times 16 \approx 512$ KB para el *grid* más pequeño y de $257 \times 512 \times 16 \approx 2$ MB para el *grid* más grande. Para entender el segundo factor, nótese que para un *grid* de 256^3 elementos y 16 GPU cada proceso MPI le corresponde sólo $256/16 = 16$ planos. Por lo tanto, excepto para tamaños de segmento muy pequeños, el número de segmentos está limitado a valores muy pequeños, lo que conduce a una *pipeline* ineficiente.

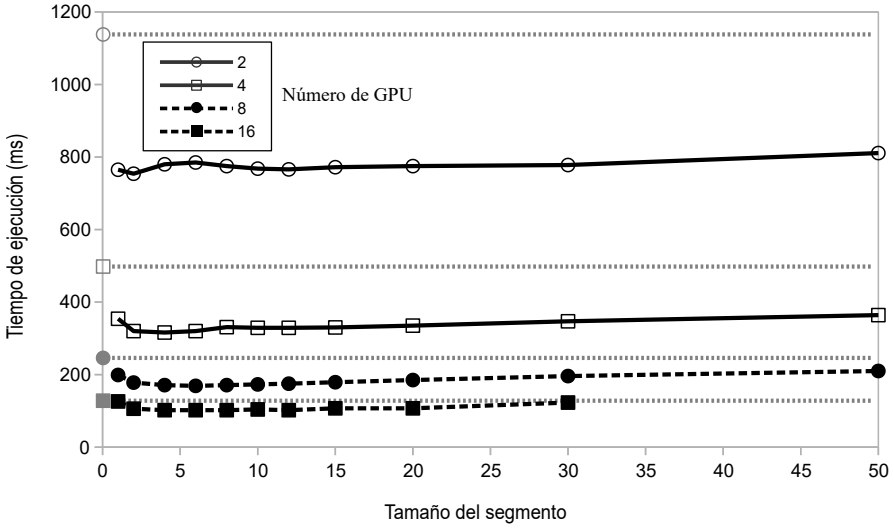

Figura 5.10: Tiempo de ejecución (ms) para un *grid* de 512^3 elementos y 2, 4, 8 y 16 GPU dependiente del tamaño de segmento. El tamaño de segmento viene dado en número de planos.

La combinación de estos dos factores dificulta la realización de una segmentación eficiente para tamaños de *grid* de 256^3 elementos para 8 y 16 GPU. En este caso, la mejoría respecto a la versión no segmentada es del 20 % y 10 %, respectivamente. Para 2 y 4 GPU, la mejoría alcanza hasta un 35 % y se obtiene para un intervalo de tamaño de segmento más amplio. Para un *grid* de 512^3 elementos, los dos factores se suavizan y se alcanza una mejoría de un 30 % y un 20 % para 8 y 16 GPU. Para ese tamaño y 2 GPU se pueden observar irregularidades en la parte cóncava para tamaños pequeños de segmento. Estas irregularidades pueden ser explicadas por el particular buen desempeño que la comunicación MPI tiene con los valores de 1 y 2 planos por segmento –2 y 4 MB– (ver figura 5.6), con lo que los dos primeros puntos de la curva se ubican más abajo de lo esperado.

Para un *grid* de 1024^3 elementos (ver la figura 5.11) la implementación se puede ejecutar en 8 y 16 GPU. En ambos casos los mejores resultados se obtienen para el tamaño de 4 planos por segmento, alcanzando una cota de mejoría de hasta el 44 % y el 23 % para 8 y 16 GPU respectivamente.

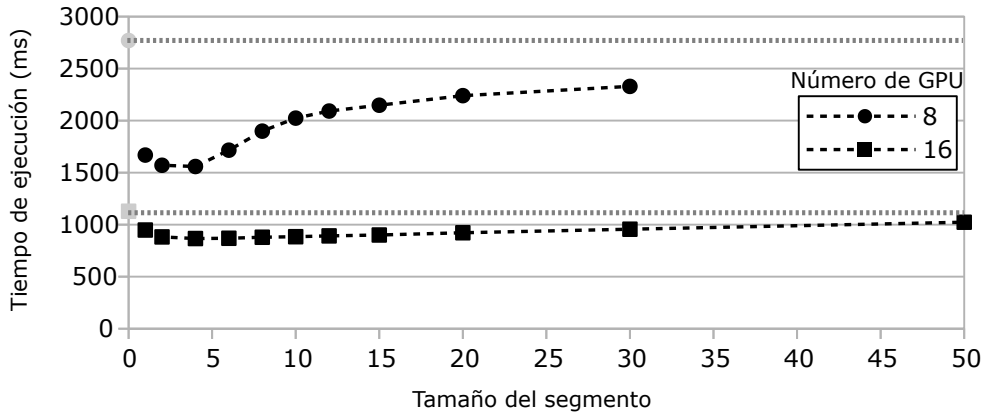

Figura 5.11: Tiempo de ejecución (ms) para un *grid* de 1024^3 elementos y 8 y 16 GPU dependiente del tamaño de segmento. El tamaño de segmento viene dado en número de planos.

Respecto a la escalabilidad de la implementación segmentada, se observa que, tal y como ocurre con la implementación no segmentada, alcanza una escalabilidad superlineal hasta 8 GPU. De nuevo, atribuimos este comportamiento a los diferentes valores que puede alcanzar el ancho de banda para la comunicación MPI *Alltoall* en función del número de procesos MPI involucrados y el tamaño de los mensajes enviados. En la tabla 5.3 se observa la aceleración que se ha obtenido en comparación a la ejecución en 2 GPU. El tamaño de segmento se ha fijado en 4 planos para un *grid* de 512^3 elementos y se ha ampliado a 12 para un *grid* de 256^3 elementos. Para 16 GPU y un *grid* de 512^3 elementos, la escalabilidad aún es muy buena ya que se consigue una eficiencia del 96 %. Para un *grid* pequeño de 256^3 elementos es más complicado mantener la eficiencia; aún así, la implementación alcanza valores de hasta el 77 %.

Tabla 5.3: Speedup de la implementación segmentada respecto a la ejecución en 2 GPU.

	GPUs		
N	4	8	16
256^3	2,24	3,83	6,13
512^3	2,47	4,56	7,65

Influencia de MPI Alltoall

Tal y como ya se ha podido observar en la figura 5.5, la operación más lenta es la comunicación MPI *Alltoall*, por lo que se espera que sea la tarea dominante en el *pipeline* de la implementación segmentada. De hecho, si nos centramos en el *grid* de 512^3 elementos, y se desechan los casos extremos donde la mejoría respecto a la versión secuencial es inferior al 15 %, se ha comprobado que el tiempo invertido en la comunicación MPI *Alltoall*, de media, supone un 89 % del tiempo total de ejecución. Esto significa que la mayoría de operaciones restantes se solapan con la comunicación MPI *Alltoall*. Para este cálculo sólo se ha tenido en consideración el tiempo correspondiente a las tareas de la parte segmentada. Sin embargo, también se han analizado las principales operaciones no segmentadas –el cálculo de la FFT 1D y el cálculo de la ecuación de Poisson en el espacio frecuencial– y se concluye que su tiempo de ejecución siempre es inferior al 10 % del tiempo total.

Desafortunadamente, la comunicación MPI *Alltoall* y la comunicación entre la CPU y la GPU comparten ciertos recursos –memoria de la CPU, enlaces *QPI links* y el *hub* de E/S (ver figura 5.3)– por lo que estas comunicaciones no van a poder ser tan rápidas como lo son para la implementación no segmentada. Gracias a unos test que se han realizado, se ha podido observar que la comunicación bidireccional entre la CPU y la GPU es entre un 25 %-30 % más lenta que la comunicación unidireccional. También se ha probado si este mismo efecto perdura cuando la comunicación MPI *Alltoall* está presente, y se ha podido comprobar que la comunicación bidireccional entre la CPU y la GPU es incluso un 20 % más lenta en estos casos.

La comunicación MPI *Alltoall*, que es la tarea más costosa, también muestra una desaceleración en el rendimiento obtenido cuando comparte recursos con la comunicación entre la CPU y la GPU. Sin embargo, se estima que esta desaceleración es débil al nivel de la meseta –sobre el 10 %–, si bien es más significativa para los puntos máximos del ancho de banda en la figura 5.6.

Un ejemplo práctico del esquema de segmentación

A continuación se va a profundizar, mediante un ejemplo, en el esquema de segmentación planteado anteriormente en la sección 5.3.2 del presente capítulo (figuras: 5.1 y 5.2). Para realizar este ejemplo se ha elegido una de las combinaciones de los resultados presentados en la figura 5.12, donde se muestra la evolución del rendimiento en el tiempo de ejecución en base al

número de segmentos utilizados en la implementación para 8 GPU y una matriz de entrada de 512^3 elementos. La línea continua indica el tiempo real consumido para la ejecución paralela del algoritmo segmentado, mientras que el resto de los valores indican la suma del tiempo de ejecución de todas las operaciones para ese tamaño de segmento.

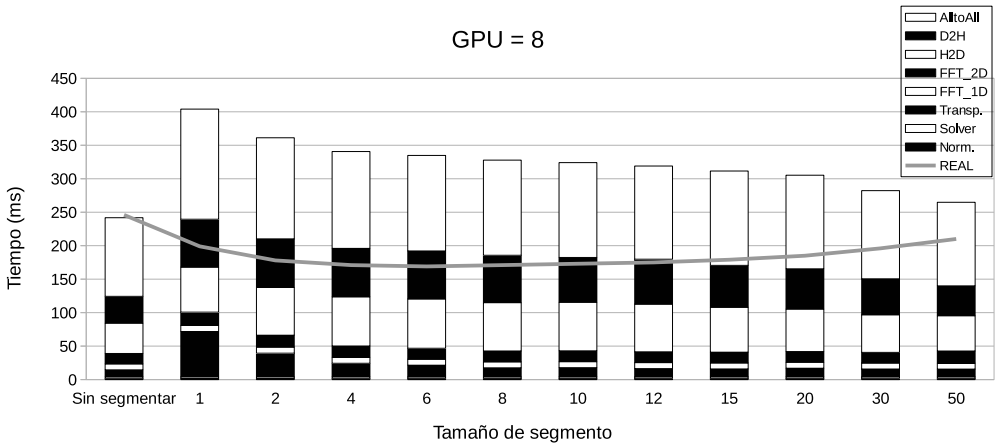

Figura 5.12: Evolución del rendimiento en el tiempo de ejecución en base al número de segmentos utilizados en la implementación para 8 GPU y una matriz de entrada de 512^3 elementos. La línea continua indica el tiempo real consumido en la ejecución paralela del algoritmo segmentado mientras que el resto de los valores indican la suma de todas las operaciones.

Si se analiza la figura, se confirman los datos ya comentados anteriormente, donde el algoritmo segmentado consigue mejor rendimiento –se ejecuta en menor tiempo– que su equivalente sin segmentar. A modo de ejemplo se va a analizar uno de los casos presentes en esta figura, en concreto, la ejecución con 6 segmentos. En la tabla 5.4 se puede observar el tiempo de ejecución necesario para resolver la ecuación de Poisson en un clúster con 8 GPU para una matriz de entrada de 512^3 elementos mediante la utilización de 6 segmentos. En la primera columna se encuentran todas la operaciones que se deben realizar para resolver la ecuación de Poisson, en la segunda columna se indica el tiempo total que se ha necesitado para cada conjunto de operaciones del mismo tipo, en la tercera columna aparece el número de veces que se ejecuta esa operación, y en la última columna se indica el tiempo de ejecución individual de cada tarea. En las dos últimas filas se muestra, en primer lugar, la suma del tiempo de ejecución de todas las operaciones en la versión segmentada y, en segundo lugar, el tiempo de ejecución real.

Tabla 5.4: Tiempo de ejecución (*Suma* –suma del tiempo de ejecución de todas las operaciones– y *Real* –tiempo de ejecución final para la ejecución segmentada–) para el cálculo de Poisson en un clúster con 8 GPU para una matriz de entrada de 512^3 elementos y 6 segmentos. La columna *Total*, agrupa los tiempos según el número de repeticiones (*Rep.*). La última columna (*Indiv.*) muestra el tiempo de ejecución individual.

	Tiempo de ejecución (ms)		
Operación	*Total*	*Rep.*	*Indiv.*
Alltoall	143	2	71,5
H2D	73,66	2	36,86
D2H	71,61	2	35,8
FFT2D	16,38	2	8,19
FFT1D	8,96	2	4,48
Transp.	17,3	4	4,33
Poisson	2,35	1	2,35
Normal.	1,83	1	1,83
Suma	335,09		
Real	**169**		

Tal y como se ha comentado anteriormente y se confirma con los datos de esta tabla, en el cálculo de la ecuación de Poisson las operaciones dominantes son aquellas que están relacionadas con la comunicación; donde destaca especialmente la comunicación MPI *Alltoall*.

A continuación se presenta el resultado de la aplicación de las diferentes fases de la segmentación teórica vista en la figura 5.1 a una matriz de entrada de 512^3 elementos sobre 8 GPU y 6 segmentos.

En la figura 5.13 se presenta la aplicación de la primera fase de la segmentación. En la figura se muestra la organización de los segmentos definidos en la sección 5.3.2, donde la ejecución de los *kernels* se solapa con la comunicación entre la CPU y la GPU y con la comunicación MPI *Alltoall*. Para conseguir este objetivo, tal y como se ha comentado anteriormente, en la implementación se han utilizado diferentes *streams* de CUDA, que permiten realizar varias operaciones simultáneamente en la misma GPU. Por ejemplo, mientras que un *stream* se encarga de ejecutar los *kernels* (FFT 2D y transponer los datos)

y la comunicación D2H para el segmento número 4, otro *stream* se encarga de realizar la comunicación H2D para el segmento número 2. Además, no hay que olvidar que al mismo tiempo se está realizando la comunicación MPI *Alltoall* del segmento número 3 en la CPU. Nótese que los bloques de estas figuras están a escala acorde a su tiempo de ejecución, y que por ese motivo, la distribución visual varía respecto al propuesto en la segmentación teórica, donde todas las operaciones tenían el mismo tamaño.

Para la segunda fase no hay variaciones de ningún tipo respecto a la segmentación teórica que se ha planteado en la sección 5.3.2, ya que las operaciones a ejecutar en esta etapa se ejecutan una detrás de otra en la misma GPU y no pueden ser solapadas con las operaciones de otras fases.

En la figura 5.14 se presenta la aplicación de la tercera fase de la segmentación teórica. En esta fase, básicamente los solapes se realizan con la operación de la comunicación MPI *Alltoall*. Por ejemplo, mientras se realiza la comunicación MPI *Alltoall* para el segmento número 3 en la CPU, un *stream* se encarga de ejecutar los *kernels* (transponer los datos, FFT 2D inversa y normalización) y la comunicación H2D para el segmento número 2. Al mismo tiempo, otro *stream* se encarga de ejecutar el *kernel* para transponer los datos y realiza la comunicación D2H para el segmento número 4. Nótese que aquí también los bloques de las figuras están a escala acorde a su tiempo de ejecución.

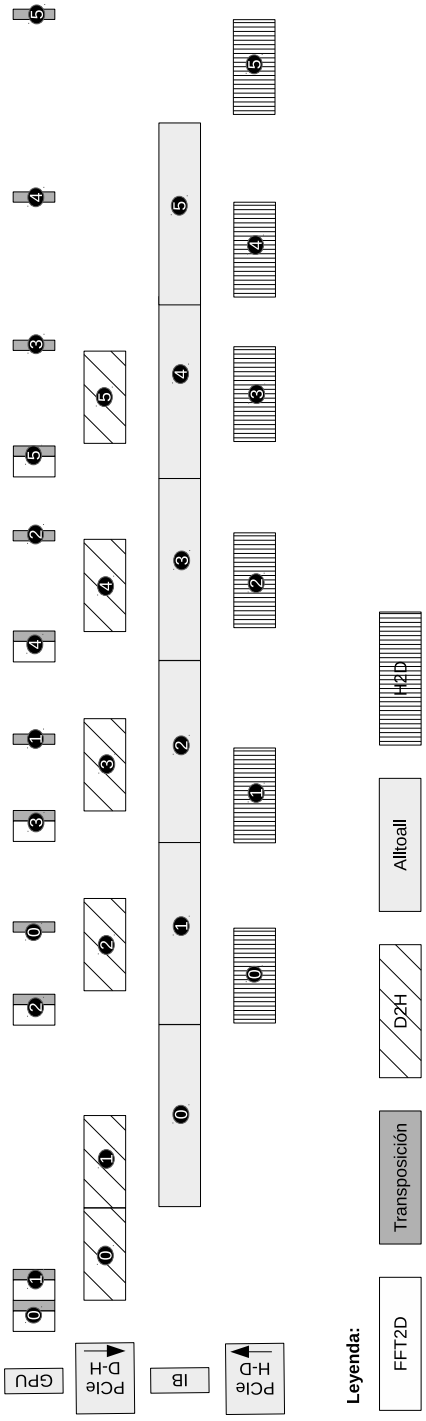

Figura 5.13: Aplicación de la primera fase de la segmentación teórica a una matriz de entrada de 512^3 elementos sobre 8 GPU y 6 segmentos.

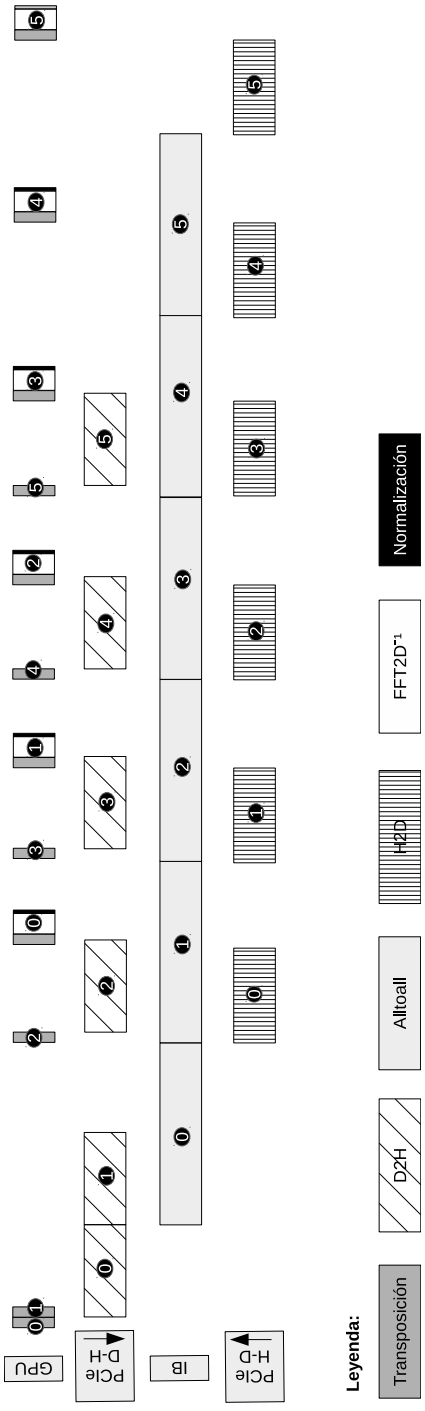

Figura 5.14: Aplicación de la tercera fase de la segmentación teórica a una matriz de entrada de 512^3 elementos sobre 8 GPU y 6 segmentos.

Comparación con la implementación en un clúster de CPU

Para finalizar con la experimentación, se ha comparado el rendimiento obtenido para la solución que se ha implementado en el clúster de GPU con el rendimiento obtenido con la implementación basada en la librería FFTW en múltiples CPU. Afortunadamente, la librería FFTW es capaz de calcular FFT en entornos distribuidos, aspecto que ha sido aprovechado en este trabajo. En este caso, la experimentación se ha realizado lanzando 1 proceso MPI por cada CPU y un hilo de ejecución para cada núcleo en la CPU. Para la implementación en la GPU, se ha utilizado un tamaño de segmento de 4 planos, excepto para tamaños de *grid* 256^3, donde para este caso el tamaño de segmento de 12 planos es más adecuado.

Aunque los resultados para un único nodo demuestran que el rendimiento obtenido en el cálculo de la FFT en GPU es adecuado, también sabemos que el rendimiento obtenido para la implementación multi-nodo está totalmente dominado por la comunicación MPI. Por lo que no se esperan grandes diferencias en el rendimiento para la implementación de la versión multi-nodo.

Por otra parte, la documentación de la librería FFTW establece que su propia implementación para realizar la comunicación MPI no debería ser más lenta que la implementación del MPI *Alltoall* estándar, ya que internamente considera las implementaciones de varios algoritmos –la implementación del propio MPI *Alltoall* estándar inclusive [115].

Sin embargo, tal y como se puede ver en las figuras 5.15 y 5.16, la implementación GPU multi-nodo siempre es más rápida que la implementación CPU. En la figura 5.15 se pueden ver los tiempos de ejecución para las implementaciones GPU y CPU para tamaños de *grid* de 256^3, 512^3 y 1024^3 elementos. En la figura 5.16 se puede ver el factor de aceleración de la implementación GPU respecto a la implementación CPU. En la misma figura se puede observar que el valor más alto para el factor de aceleración se obtiene para el *grid* de tamaño de 512^3 elementos. En este caso, el factor de aceleración es alrededor de 3 veces superior respecto a la implementación CPU, aunque este valor va aumentando según se incrementa el número de procesos MPI.

Por último, parece que el *grid* de tamaño de 256^3 elementos no es lo suficientemente grande para ocupar todos los recursos de las GPU en la implementación multi-nodo, ya que para este caso el factor de aceleración desciende de 2,6 a 1,9 para 4 y 16 GPU, respectivamente. Para el *grid* de tamaño de 1024^3 elementos sólo se dispone de dos medidas, que corresponden

a los valores obtenido para 8 y 16 procesos MPI, y en ambos casos se obtiene un factor de aceleración de 2,6.

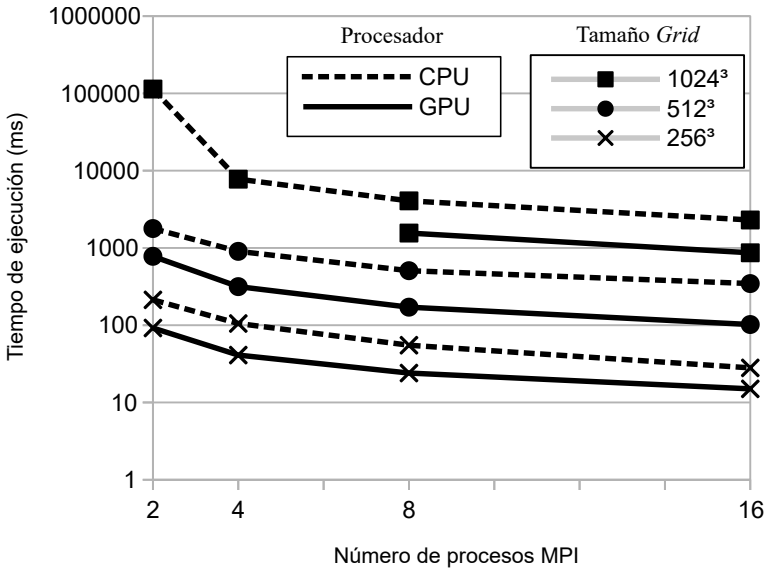

Figura 5.15: Tiempos de ejecución para las implementaciones GPU y CPU.

En general, las figuras muestran que la implementación GPU funciona entre 2 y 3 veces más rápido que la implementación CPU.

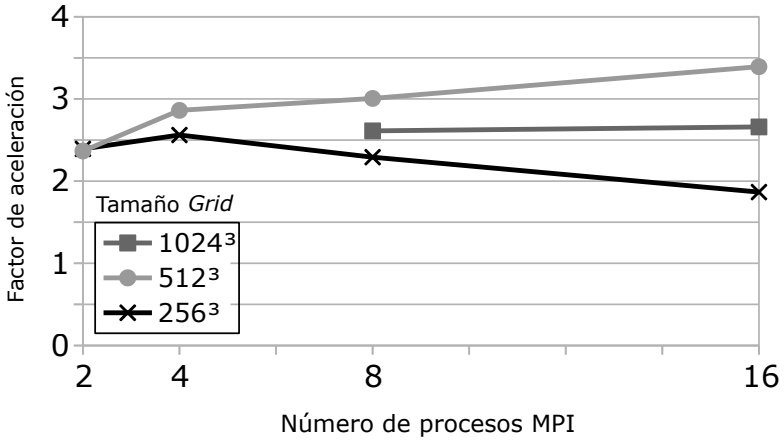

Figura 5.16: Factor de aceleración de la implementación GPU respecto a la implementación CPU.

Comparación con trabajos similares

Para finalizar con la sección de los resultados, se van a comparar los resultados obtenidos en la experimentación con los resultados que otros autores han obtenido en otros trabajos similares.

Dado que este trabajo es el primero que aborda la solución eficiente de la ecuación de Poisson en un clúster de GPU, este apartado se va a centrar en comparar este trabajo con otros que calculan la FFT en un clúster de GPU, que es la parte fundamental en el cálculo de la ecuación de Poisson.

Tal y como se ha mencionado en el capítulo 2 (subsección 2.4.3), en la literatura principalmente existen dos trabajos de este tipo. Ambos poseen importantes diferencias respecto al nuestro, por lo que la comparativa no puede ser muy precisa. De todos modos, la comparativa se limita a un análisis cualitativo.

En el trabajo de Chen et al. [10] la configuración de la red es muy similar a la utilizada en el presente trabajo –con dos interfaces IB por nodo–, aunque las dos GPU de cada nodo no lo son.

El rendimiento teórico de las GPU Tesla C1060 y GeForce GTX285 que han usado en ese trabajo es indudablemente inferior al que se obtiene en una de las Tesla M2090 que se han utilizado en el nuestro. Por otra parte, su experimentación se ha realizado con transformaciones de datos de complejo a complejo mientras que en nuestro trabajo las transformaciones de datos se han realizado de real a complejo. Este hecho hace que la comparación sea muy difícil, aunque hemos añadido los resultados con fines ilustrativos.

El trabajo de Nukada et al. [15] se puede considerar más similar al desarrollado en el nuestro, aunque también posee algunas diferencias significativas. Cada nodo de su clúster dispone de 3 GPU Tesla M2050 y dos interfaces IB. Las GPU son muy similares a las utilizadas en el nuestro, aunque el rendimiento de las GPU Tesla M2050 (515 Gflops) es inferior al de las M2090 (665 Gflops). No obstante, al disponer de 3 M2050 por nodo, el rendimiento total (1545 Gflops) es superior al obtenido con 2 M2090 (1330 Gflops). En su trabajo también han realizado transformaciones de complejo a complejo.

En la tabla 5.5 se pueden ver los resultados que de alguna manera se consideran comparables. Los resultados se muestran en Gflops. Para calcular el número de operaciones en coma flotante se ha utilizado el mismo método al

utilizado en los trabajos a comparar $-15N^3 \log_2 N$ para matrices cúbicas de N elementos por lado– adaptado al hecho de que la resolución de la ecuación de Poisson requiere 2 cálculos de FFT. No se han tenido en cuenta las operaciones relacionadas a la resolución de la ecuación de Poisson presentes en este trabajo de investigación pero no presentes en el resto, puesto que se estima que la mejora obtenida en el rendimiento no sería significativa.

Tabla 5.5: Comparación del rendimiento (Gflops) alcanzado en nuestra implementación en comparación con trabajos similares. Los valores de los resultados de los otros trabajos se han extraído de figuras por lo que hay que considerarlos como valores aproximados.

# de nodos	Nuestro trabajo	Chen et al.	Nukada et al. (1 GPU por nodo)	Nukada et al. (3 GPU por nodo)
2	115	12	N/A	N/A
4	214	20	55	75
8	355	40	110	150

La comparativa muestra que nuestros resultados mejoran los resultados de los trabajos similares. Sin embargo, hay que recordar que los trabajos no son directamente comparables y la comparativa debe servir única y exclusivamente para dar una idea de la magnitud del rendimiento obtenido. Por ejemplo, el hecho de que nosotros realicemos transformadas de real a complejo aproximadamente podría duplicar nuestro rendimiento respecto al suyo. También hay que tener en cuenta que el rendimiento de la GPU Tesla M2090 es aproximadamente un 22 % superior al de la M2050, aunque en una aplicación donde el cuello de botella se encuentra en la comunicación es difícil de estimar el impacto real que este rendimiento extra puede tener en los resultados. En un amplio sentido, afirmamos que el nivel de rendimiento de nuestra implementación es, al menos, comparable al de otros trabajos que calculan FFT en un clúster de GPU. Hay que tener en cuenta que esta afirmación es muy importante, ya que nuestra aplicación se basa en MPI y no se ha realizado ningún tipo de optimización específica relacionada con la red IB, siendo, por tanto, portable a otras redes.

5.5. Conclusiones

En este capítulo se ha descrito la implementación para resolver la ecuación de Poisson en un clúster GPU. La implementación se basa en el algoritmo FFT, por lo que el patrón de comunicación MPI *Alltoall* es la operación dominante en este proceso.

En primer lugar, se ha realizado un análisis de la ecuación de Poisson, donde se han identificado las fases necesarias para su resolución. Estas fases se han dividido en etapas que permitan ejecutar en paralelo las operaciones que la componen. Para ello se ha realizado un proceso de segmentación, donde se han analizado las dependencias entre las operaciones de una misma etapa y se han establecido *streams* que permiten la ejecución paralela de dichas operaciones en una misma etapa.

A continuación, se ha realizado una experimentación con diferentes tamaños de segmento con el objeto de identificar cuál es la cantidad de datos que se deben de procesar en cada etapa para conseguir una implementación eficiente.

Todo esto se traduce en que la implementación segmentada que se ha desarrollado es competitiva. La naturaleza segmentada de esta implementación permite prácticamente un solapamiento total de operaciones entre la comunicación MPI *Alltoall* inter-nodo y el resto de operaciones de cálculo y comunicación. El resultado es una implementación hasta 3,4 veces más rápida que la implementación CPU basada en la librería FFTW. Por lo tanto, el uso de la implementación que se ha desarrollado en el presente trabajo de investigación, permite resolver la ecuación de Poisson en un clúster de GPU de manera eficiente mientras las CPU quedan liberadas y pueden ser utilizadas para realizar otro tipo de cálculos.

La implementación también muestra una muy buena escalabilidad hasta 16 GPU. Por el momento, no se ha podido ampliar la experimentación en un clúster con más GPU debido a que el entorno de experimentación que se ha utilizado no dispone del hardware necesario para ello. Aunque tal y como se ha mencionado, en el ámbito de experimentación que nos hemos podido mover la escalabilidad ha sido correcta.

Parte III

CONCLUSIONES

Capítulo 6

Conclusiones

Índice

Este último capítulo resume las principales aportaciones realizadas por esta tesis, junto con las publicaciones que han generado. Finalmente se recogen las líneas futuras que se derivan de este trabajo.

6.1. Conclusiones

Dado que cada uno de los capítulos finaliza con una sección dedicada a las conclusiones particulares del trabajo descrito en él, las conclusiones mostradas a continuación se limitan a ser una síntesis de las más importantes.

En el **capítulo 3** se ha analizado el rendimiento que ofrece la librería cuFFT de NVIDIA, tanto en consumo de memoria como en tiempo de ejecución, para dos de las principales opciones que ofrece: la opción ***batch*** y la opción ***stride***.

Los resultados obtenidos demuestran que el uso de la opción ***stride*** no

afecta a la memoria requerida por la librería cuFFT, ya que la cantidad de memoria a usar es, en ambos casos, equivalente al tamaño de los datos a procesar. Sin embargo, la opción ***batch*** sí que afecta directamente a los requerimientos de memoria, ya que aumenta linealmente el consumo de la memoria por el número de FFT a calcular en un mismo ***plan***.

Respecto al tiempo de ejecución, la opción ***batch*** afecta de manera significativa al cálculo de las FFT de tamaño pequeño y medio. La razón principal de este impacto para FFT de este tamaño radica en la infrautilización del hardware de la GPU.

Estos resultados ofrecen dos conclusiones prácticas:

- Se puede calcular el requerimiento de memoria necesario para el cálculo de las FFT que se han planificado
- Se puede estimar el impacto que tiene la reducción del número de FFT a calcular en ***batch*** en el tiempo de ejecución de las FFT.

En nuestra experimentación, y para el hardware que se ha utilizado, si por algún motivo se requiere limitar la cantidad de memoria necesaria para el cálculo de las transformadas, es posible limitar el consumo de memoria de la librería cuFFT alrededor de los 50 MB, ya que a partir de esta cantidad de memoria la opción ***batch*** no aporta beneficios en el cálculo de las transformadas.

Al analizar la penalización a la que se incurre al calcular FFT con datos no contiguos, los resultados muestran que esta penalización está condicionada por el tamaño de la FFT y por el número de FFT a calcular en ***batch***. La penalización más significativa que ha sido identificada en la experimentación ha alcanzado incluso una penalización del 300 % respecto a su equivalente FFT con datos contiguos.

En el **capítulo 4** se ha presentado una implementación para transponer de manera eficiente matrices de 3D en una GPU. La experimentación ha demostrado que para todas las posibles transposiciones *out-of-place* se consigue un rendimiento alto. También ha quedado patente la importancia del número de elementos que se transponen por cada hilo (EpT). En este caso, tras probar con diferentes valores y en 4 GPU diferentes con la arquitectura Fermi, la experimentación muestra que transponer 2 elementos por cada hilo es una buena opción.

También se han implementado una serie de transposiciones *in-place* básicas, que en la mayoría de los casos también han demostrado un rendimiento alto. En este caso, las dos transposiciones de rotación han obtenido un rendimiento 30 % inferior en comparación con el resto de las transposiciones del mismo tipo.

En el **capítulo 5** se ha descrito la implementación para resolver eficientemente la ecuación de Poisson en un clúster GPU. La implementación se basa en el algoritmo FFT, y la experimentación muestra que la comunicación MPI *Alltoall* es la dominante en este proceso.

La naturaleza segmentada de esta implementación permite prácticamente un solapamiento total de tareas entre la comunicación MPI *Alltoall* internodo y el resto de tareas de cálculo y comunicación. El resultado es una implementación hasta 3,4 veces más rápida que la implementación CPU basada en la librería FFTW y muestra una muy buena escalabilidad hasta 16 GPU.

6.1.1. Publicaciones

La difusión de los resultados obtenidos ha quedado avalada con la publicación de dos artículos en revistas (uno en revisión) y con la participación en dos congresos (uno internacional y otro nacional).

Publicaciones en revistas

Autores: J.L. Jodra, I. Gurrutxaga, J. Muguerza.
Título: Efficient 3D Transpositions in Graphics Processing Units
Publicación: International Journal of Parallel Programming. vol. 43, nº5, pp. 876-891, 2015.

Autores: J.L. Jodra, I. Gurrutxaga, J. Muguerza, A. Yera.
Título: Solving Poisson's Equation using FFT in a GPU Cluster
Publicación: Journal of Parallel and Distributed Computing. Enviado en 2015, pendiente de aceptación (segunda revisión).

Congresos

Autores: J.L. Jodra, I. Gurrutxaga, J. Muguerza.
Título: A study of memory consumption and execution performance of the cuFFT library
Tipo de participación: Presentación oral.
Congreso: 10th International conference on P2P, Parallel, Grid, Cloud and Internet Computing. 2015, Krakow, Poland.
Publicación: Proceedings of the 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing. IEEE CPS.

Autores: J.L. Jodra, I. Gurrutxaga, J. Muguerza.
Título: Resolviendo la ecuación de Poisson en un clúster de GPU
Tipo de participación: Presentación oral.
Congreso: XXVI Jornadas de Paralelismo (JP2015). 2015, Córdoba.
Publicación: Proceedings XXVI Jornadas de Paralelismo (JP2015).

6.2. Líneas futuras de trabajo

Las conclusiones obtenidas al final del trabajo de investigación de esta tesis doctoral permiten apuntar diferentes direcciones de investigación:

- **FFT con Transposiciones vs FFT con *stride*.** En el capítulo 3 se ha analizado el rendimiento que ofrece la librería cuFFT de NVIDIA. Una de las principales opciones que se ha estudiado ha sido aquella que permite calcular la FFT con matrices de entrada y salida con datos no contiguos. En el capítulo 4 se ha presentado una manera para realizar transposiciones en 3D de manera eficiente, así como un estudio preliminar en el que se deja entrever que puede ser más adecuado transponer los datos mediante un algoritmo eficiente para a continuación calcular la FFT con datos contiguos, que hacer uso de la opción ***stride*** de la librería cuFFT. Una de las líneas futuras más inmediatas consiste en realizar un estudio en profundidad que permita confirmar si los resultados obtenidos en el análisis preliminar se mantienen en la misma línea y si pueden aplicarse así como determinar los parámetros para los que se cumple (tamaño, *stride*, ...) para cualquier tamaño de matriz de entrada.
- **Transposiciones in-place.** En el capítulo 4 se han detallado en profundidad diferentes implementaciones para cada transposición 3D *out-of-place*. Las transposiciones 3D *in-place* plantean un problema más complejo. Sin embargo, se han implementado una serie de transposiciones 3D *in-place* básicas que han dado unos resultados con un rendimiento razonable. Otra de las líneas futuras más inmediatas consiste en implementar los 6 tipos de transposiciones posibles para cualquier tamaño de la matriz de entrada.
- **Condiciones de contorno no periódicas.** Debido a las características periódicas de la transformada de Fourier, la ecuación de Poisson resuelta mediante FFT sirve a priori sólo para condiciones de contorno periódicas. En cualquier caso, se puede variar el algoritmo para resolver la ecuación en otro tipo de contornos. Una de estas variaciones exige duplicar la longitud de cada lado de la malla de entrada y rellenar los nuevos casos con ceros, con lo que el coste de resolver una ecuación de Poisson en 3 dimensiones se multiplica aproximadamente por 8. De todos modos, existen optimizaciones que reducen considerablemente este aumento de cómputo [23]. Otra de las líneas futuras es, por tanto, modificar el código implementado para que pueda resolver la ecuación

de Poisson en un clúster de GPU con este tipo de condiciones de contorno no periódicas y optimizarlo para reducir el aumento de cómputo asociado al aumento del tamaño de la malla.

- **Permitir que los datos de entrada superen la memoria de la GPU.** Las implementaciones que se han realizado en el presente trabajo están limitadas por el tamaño de la memoria de la GPU en la que se ejecuta. Otra de las líneas futuras es la de realizar una implementación capaz de aprovechar la memoria que ofrecen las CPU – habitualmente de mayor tamaño que la memoria de las GPU – y que permita trabajar con matrices de datos que superen el tamaño de la GPU.
- **Optimizar comunicación inter-nodo.** En el capítulo 5 se ha visto que la comunicación inter-nodo es la tarea dominante en la implementación para resolver la ecuación de Poisson en un clúster de GPU. Por lo que futuras optimizaciones deberían trabajar en esta dirección. Una posible solución consiste en implementar la fase de comunicación mediante primitivas InfiniBand (por ejemplo, mediante *IB verbs*), en vez de utilizar las funciones estándar de MPI. Sin embargo, la realización de este cambio debe realizarse con cautela, ya que reduce la portabilidad del código limitándolo a redes de comunicación basadas en InfiniBand. Otra manera para reducir el coste de la comunicación puede ser la utilización de la tecnología GPUDirect RDMA de NVIDIA, la cual permite acceder directamente a la memoria de la GPU. Sin duda alguna, constituye otra línea futura a trabajar.

Bibliografía

[1] H. Meuer, E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer, "November 2015 | TOP500 Supercomputer Sites," Nov. 2015. [Online]. Available: http://www.top500.org/list/2015/11/

[2] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, "A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures," *Communications in Computational Physics*, 2013.

[3] W.-M. W. Hwu, *GPU Computing Gems Jade Edition*. Boston, MA: Morgan Kaufmann, 2011.

[4] W.-m. W. Hwu, Ed., *GPU Computing Gems Emerald Edition*, 1st ed. Amsterdam ; Burlington, MA: Morgan Kaufmann, Feb. 2011.

[5] P. García-Risueño, J. Alberdi-Rodriguez, M. J. T. Oliveira, X. Andrade, M. Pippig, J. Muguerza, A. Arruabarrena, and A. Rubio, "A survey of the parallel performance and accuracy of Poisson solvers for electronic structure calculations," *Journal of Computational Chemistry*, vol. 35, no. 6, pp. 427–444, Mar. 2014. [Online]. Available: http://doi.wiley.com/10.1002/jcc.23487

[6] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comp.*, vol. 19, no. 90, pp. 297–301, 1965. [Online]. Available: http://www.ams.org/mcom/1965-19-090/S0025-5718-1965-0178586-1/

[7] M. Frigo and S. Johnson, "FFTW: an adaptive software architecture for the FFT," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, 1998*, vol. 3, May 1998, pp. 1381–1384 vol.3.

[8] M. Pippig, "PFFT: An Extension of FFTW to Massively Parallel Architectures," *SIAM Journal on Scientific Computing*, vol. 35, no. 3, pp. C213–C236, May 2013. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/120885887

[9] O. Ayala and L.-P. Wang, "Parallel implementation and scalability analysis of 3d Fast Fourier Transform using 2d domain decomposition," *Parallel Computing*, vol. 39, no. 1, pp. 58–77, Jan. 2013. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0167819112000932

[10] Y. Chen, X. Cui, and H. Mei, "Large-scale FFT on GPU clusters," in *Proceedings of the 24th ACM International Conference on Supercomputing*. ACM, 2010, pp. 315–324. [Online]. Available: http://dl.acm.org/citation.cfm?id=1810128

[11] S. Chen and X. Li, "A hybrid GPU/CPU FFT library for large FFT problems," in *Performance Computing and Communications Conference (IPCCC), 2013 IEEE 32nd International*. IEEE, 2013, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6742796

[12] L. Gu, J. Siegel, and X. Li, "Using GPUs to compute large out-of-card FFTs," in *Proceedings of the international conference on Supercomputing*. ACM, 2011, pp. 255–264. [Online]. Available: http://dl.acm.org/citation.cfm?id=1995937

[13] N. Nandapalan, J. Jaros, A. Rendell, and B. Treeby, "Implementation of 3d FFTs Across Multiple GPUs in Shared Memory Environments," in *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, Dec. 2012, pp. 167–172.

[14] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka, "Bandwidth intensive 3-D FFT kernel for GPUs using CUDA," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, Nov. 2008, pp. 1–11.

[15] A. Nukada, K. Sato, and S. Matsuoka, "Scalable multi-gpu 3-d fft for tsubame 2.0 supercomputer," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 44. [Online]. Available: http://dl.acm.org/citation.cfm?id=2389056

[16] J. Wu and J. JaJa, "Optimized strategies for mapping three-dimensional ffts onto cuda gpus," in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–12. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6339608

[17] D. Bailey, "FFTs in external or hierarchical memory," in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, 1989. Supercomputing '89*, Nov. 1989, pp. 234–242.

[18] A. Gupta and V. Kumar, "The scalability of FFT on parallel computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 8, pp. 922–932, Aug. 1993.

[19] D. Takahashi and Y. Kanada, "High-Performance Radix-2, 3 and 5 Parallel 1-D Complex FFT Algorithms for Distributed-Memory Parallel Computers," *The Journal of Supercomputing*, vol. 15, no. 2, pp. 207–228, Feb. 2000. [Online]. Available: http://link.springer.com/article/10.1023/A%3A1008160021085

[20] R. B. Wilhelmson and J. H. Ericksen, "Direct solutions for Poisson's equation in three dimensions," *Journal of Computational Physics*, vol. 25, no. 4, pp. 319–331, Dec. 1977. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0021999177900018

[21] A. Shiferaw and R. Chand Mittal, "An Efficient Direct Method to Solve the Three Dimensional Poisson's Equation," *American Journal of Computational Mathematics*, vol. 01, no. 04, 2011.

[22] V. Fuka, "PoisFFT - A Free Parallel Fast Poisson Solver," *Appl. Math. Comput.*, vol. 267, no. C, pp. 356–364, Sep. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.amc.2015.03.011

[23] N. Dugan, L. Genovese, and S. Goedecker, "A customized 3d GPU Poisson solver for free boundary conditions," *Computer Physics Communications*, vol. 184, no. 8, pp. 1815–1820, Aug. 2013. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0010465513000817

[24] S. Goedecker and K. Maschke, "Efficient pseudopotentials and algorithms for electronic structure calculations based on density functional theory," Thesis, EPFL, Lausanne, 1992, doi:10.5075/epfl-thesis-988. [Online]. Available: http://infoscience.epfl.ch/record/31466

[25] V. K. Decyk and T. V. Singh, "Adaptable Particle-in-Cell algorithms for graphical processing units," *Computer Physics Communications*, vol. 182, no. 3, pp. 641–648, Mar. 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465510004558

[26] D. Rossinelli, M. Bergdorf, G.-H. Cottet, and P. Koumoutsakos, "GPU accelerated simulations of bluff body flows using vortex particle methods," *Journal of Computational Physics*, vol. 229, no. 9, pp. 3316–3333, May 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999110000197

[27] A. Kosior and H. Kudela, "Parallel computations on GPU in 3d using the vortex particle method," *Computers & Fluids*, vol. 80, pp. 423–428, Jul. 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0045793012000230

[28] G. Knittel, "A CG-based Poisson solver on a GPU-cluster," in *2010 International Conference on High Performance Computing (HiPC)*, Dec. 2010, pp. 1–10.

[29] J. Shi, Y. Cai, W. Hou, L. Ma, S. Tan, P.-H. Ho, and X. Wang, "GPU friendly Fast Poisson Solver for structured power grid network analysis," in *46th ACM/IEEE Design Automation Conference, 2009. DAC '09*, Jul. 2009, pp. 178–183.

[30] J. Krüger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," in *ACM Transactions on Graphics (TOG)*, vol. 22. ACM, 2003, pp. 908–916. [Online]. Available: http://dl.acm.org/citation.cfm?id=882363

[31] D. Goddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, and S. Turek, "Using GPUs to improve multigrid solver performance on a cluster," *International Journal of Computational Science and Engineering*, vol. 4, no. 1, pp. 36–55, Jan. 2008. [Online]. Available: http://www.inderscienceonline.com/doi/abs/10.1504/IJCSE.2008.021111

[32] H.-R. Jung, S.-T. Kim, J. Noh, and J.-M. Hong, "A heterogeneous CPU–GPU parallel approach to a multigrid Poisson solver for incompressible fluid simulation," *Comp. Anim. Virtual Worlds*, vol. 24, no. 3-4, pp. 185–193, May 2013. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/cav.1498/abstract

[33] J. C. Thibault and I. Senocak, "CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows," in *Proceedings of the 47th AIAA aerospace sciences meeting*, 2009, pp. 2009–758. [Online]. Available: http://arc.aiaa.org/doi/pdf/10.2514/6.2009-758

[34] S. Georgescu and H. Okuda, "Conjugate gradients on multiple GPUs," *Int. J. Numer. Meth. Fluids*, vol. 64, no. 10-12, pp. 1254–1273, Dec. 2010. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/fld.2462/abstract

[35] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser, "A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform," in *2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Feb. 2010, pp. 583–592.

[36] M. Griebel and P. Zaspel, "A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier-Stokes equations," *Computer Science-Research and Development*, vol. 25, no. 1-2, pp. 65–73, 2010. [Online]. Available: http://link.springer.com/article/10.1007/s00450-010-0111-7

[37] D. A. Jacobsen, J. C. Thibault, and I. Senocak, "An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters," in *48th AIAA aerospace sciences meeting and exhibit*, vol. 16, 2010. [Online]. Available: http://arc.aiaa.org/doi/pdf/10.2514/6.2010-522

[38] J. Wu and J. JaJa, "High Performance FFT Based Poisson Solver on a CPU-GPU Heterogeneous Platform," in *2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, May 2013, pp. 115–125.

[39] Jing Wu, J. JaJa, and E. Balaras, "An Optimized FFT-Based Direct Poisson Solver on CUDA GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 550–559, Mar. 2014. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6470608

[40] J. Wu and J. JaJa, "Optimized FFT computations on heterogeneous platforms with application to the Poisson equation," *Journal of Parallel and Distributed Computing*, vol. 74, no. 8, pp. 2745–2756, Aug.

2014. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0743731514000653

[41] "Historia de NVIDIA: de la tarjeta gráfica al procesador móvil |NVIDIA." [Online]. Available: http://www.nvidia.es/object/corporate-timeline-es.html

[42] J. Fung, F. Tang, and S. Mann, "Mediated reality using computer graphics hardware for computer vision," in *Sixth International Symposium on Wearable Computers, 2002. (ISWC 2002). Proceedings*, 2002, pp. 83–89.

[43] C. Aimone, A. Marjan, and S. Mann, "EyeTap video-based featureless projective motion estimation assisted by gyroscopic tracking," in *Sixth International Symposium on Wearable Computers, 2002. (ISWC 2002). Proceedings*, 2002, pp. 90–97.

[44] J. Fung and S. Mann, "Using multiple graphics cards as a general purpose parallel computer: applications to computer vision," in *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004*, vol. 1, Aug. 2004, pp. 805–808 Vol.1.

[45] ——, "Computer vision signal processing on graphics processing units," in *IEEE International Conference on Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04)*, vol. 5, May 2004, pp. V–93–6 vol.5.

[46] "Historia de AMD." [Online]. Available: http://www.amd.com/es-xl/who-we-are/corporate-information/history

[47] "Historia de ATI Technologies," Nov. 2015, page Version ID: 87442656. [Online]. Available: https://es.wikipedia.org/w/index.php?title=ATI_Technologies&oldid=87442656

[48] "OpenCL Fast Fourier Transform." [Online]. Available: http://www.bealto.com/gpu-fft_intro.html

[49] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[50] "Hardware | GeForce." [Online]. Available: http://www.geforce.com/hardware

[51] "Quadro Professional Workstation Solutions | NVIDIA." [Online]. Available: http://www.nvidia.com/object/quadro.html

[52] "The World's Fastest Mobile Processors | NVIDIA Tegra | NVIDIA." [Online]. Available: http://www.nvidia.com/object/tegra.html

[53] "High Performance Computing (HPC) and Supercomputing | NVIDIA Tesla | NVIDIA." [Online]. Available: http://www.nvidia.com/object/tesla-supercomputing-solutions.html

[54] "NVIDIA Updates GPU Roadmap; Unveils Pascal Architecture For 2016." [Online]. Available: http://www.anandtech.com/show/7900/nvidia-updates-gpu-roadmap-unveils-pascal-architecture-for-2016

[55] "NVIDIA Tesla Technical Specifications - solution for high performance computing | NVIDIA." [Online]. Available: http://www.nvidia.com/object/tesla_tech_specs.html

[56] "NVIDIA - Arquitectura CUDA de última generación, alias Fermi | NVIDIA." [Online]. Available: http://www.nvidia.es/object/fermi_architecture_es.html

[57] "NVIDIA Kepler Compute Architecture | High Performance Computing|NVIDIA." [Online]. Available: http://www.nvidia.com/object/nvidia-kepler.html

[58] "Maxwell: arquitectura de GPU de nueva generación|NVIDIA." [Online]. Available: http://www.nvidia.es/object/maxwell-gpu-architecture-es.html

[59] "Inside Pascal: NVIDIA's Newest Computing Platform | Parallel Forall." [Online]. Available: https://devblogs.nvidia.com/parallelforall/inside-pascal/

[60] "CUDA 8 Features Revealed | Parallel Forall." [Online]. Available: https://devblogs.nvidia.com/parallelforall/cuda-8-features-revealed/

[61] "CUDA C Programming Guide." [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[62] C. Sterken, "Jean Baptiste Joseph Fourier," *Interplay of Periodic, Cyclic and Stochastic Variability in Selected Areas of the H-R Diagram*, vol. 292, p. 21, 2003.

[63] M. R. Spiegel, J. Liu, L. Abellanas, and L. A. Rapún, *Fórmulas y tablas de matemática aplicada*. McGraw-Hill, Jan. 2005.

[64] A. L. Fetter and J. D. Walecka, *Theoretical Mechanics of Particles and Continua*. Dover Publications, Dec. 2003.

[65] M. Spiegel and S. Lipschutz, *Fórmulas Y Tablas De Matemática Aplicada - 4ª Edición*, 4th ed. Madrid: McGraw Hill, Nov. 2014.

[66] E. W. Weisstein, Ed., *CRC Concise Encyclopedia of Mathematics CD-ROM, Second Edition*, 2nd ed. Boca Raton, FL: CRC Press, Oct. 2002.

[67] G. Kaiser, *A Friendly Guide to Wavelets*. Boston: Birkhäuser Boston, 2011. [Online]. Available: http://link.springer.com/10.1007/978-0-8176-8111-1

[68] M. Rahman, *Applications of Fourier Transforms to Generalized Functions*. WIT Press, 2011.

[69] S. Krantz, *The Integral:A Crux for Analysis*. Morgan & Claypool, 2011. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6812532

[70] W. Rudin, *Principles of Mathematical Analysis*, 3rd ed. McGraw-Hill Education, Jan. 1976.

[71] S. W. Smith, *Scientist and Engineer's Guide to Digital Signal Processing*. San Diego, Calif: Bertrams, 1997.

[72] J. O. S. III, *Mathematics of the Discrete Fourier Transform (DFT): with Audio Applications —- Second Edition*, 2nd ed. W3K Publishing, Apr. 2007.

[73] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, 3rd ed. Upper Saddle River: Prentice Hall, Aug. 2009.

[74] H. Sorensen, D. Jones, M. Heideman, and C. Burrus, "Real-valued fast Fourier transform algorithms," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 35, no. 6, pp. 849–863, Jun. 1987.

[75] M. Heideman, D. Johnson, and C. Burrus, "Gauss and the history of the fast fourier transform," *IEEE ASSP Magazine*, vol. 1, no. 4, pp. 14–21, Oct. 1984.

[76] G. Strang and K. Aarikka, *Introduction to Applied Mathematics*. Wellesley, Mass: Wellesley-Cambridge Press, Jan. 1986.

[77] "cuFFT," Jul. 2014. [Online]. Available: https://developer.nvidia.com/cuFFT

[78] M. Hestenes and E. Stiefel, "Methods of Conjugate Gradients for Solving Linear Systems," *Journal of Research of the National Bureau of Standards*, vol. 49, no. 6, pp. 409–436, Dec. 1952.

[79] L. Greengard and V. Rokhlin, "A Fast Algorithm for Particle Simulations," *J. Comput. Phys.*, vol. 73, no. 2, pp. 325–348, Dec. 1987. [Online]. Available: http://dx.doi.org/10.1016/0021-9991(87)90140-9

[80] A. Brandt, "Multi-level adaptive solutions to boundary-value problems," *Math. Comp.*, vol. 31, no. 138, pp. 333–390, 1977. [Online]. Available: http://www.ams.org/mcom/1977-31-138/S0025-5718-1977-0431719-X/

[81] C. A. Rozzi, D. Varsano, A. Marini, E. K. U. Gross, and A. Rubio, "Exact Coulomb cutoff technique for supercell calculations," *Physical Review B*, vol. 73, no. 20, May 2006. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevB.73.205119

[82] N. Brenner, "Algorithm 467: Matrix Transposition in Place," *Commun. ACM*, vol. 16, no. 11, pp. 692–694, Nov. 1973. [Online]. Available: http://doi.acm.org/10.1145/355611.355612

[83] E. G. Cate and D. W. Twigg, "Algorithm 513: Analysis of In-Situ Transposition [F1]," *ACM Trans. Math. Softw.*, vol. 3, no. 1, pp. 104–110, Mar. 1977. [Online]. Available: http://doi.acm.org/10.1145/355719.355729

[84] S. Chatterjee and S. Sen, "Cache-efficient matrix transposition," in *Sixth International Symposium on High-Performance Computer Architecture, 2000. HPCA-6. Proceedings*, 2000, pp. 195–205.

[85] F. Gustavson, L. Karlsson, and B. K\a agström, "Parallel and Cache-Efficient In-Place Matrix Storage Format Conversion," *ACM Trans. Math. Softw.*, vol. 38, no. 3, pp. 17:1–17:32, Apr. 2012. [Online]. Available: http://doi.acm.org/10.1145/2168773.2168775

[86] M. Bian, F. Bi, and F. Liu, "Matrix transpose methods for SAR imaging system," in *2010 IEEE 10th International Conference on Signal Processing (ICSP)*, Oct. 2010, pp. 2176–2179.

[87] I.-J. Sung, "Data layout transformation through in-place transposition," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2013. [Online]. Available: https://www.ideals.illinois.edu/handle/2142/44300

[88] I.-J. Sung, J. Gómez-Luna, J. M. González-Linares, N. Guil, and W.-M. W. Hwu, "In-place transposition of rectangular matrices on accelerators," in *PPoPP '14 Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM Press, 2014, pp. 207–218. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2555243.2555266

[89] G. Ruetsch and P. Micikevicius, "Optimizing matrix transpose in CUDA," *Nvidia CUDA SDK Application Note*, vol. 18, 2009. [Online]. Available: http://www.cs.colostate.edu/~cs675/MatrixTranspose.pdf

[90] ——, "An Efficient Matrix Transpose in CUDA C/C++ | Parallel Forall," 2013. [Online]. Available: http://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc/

[91] P. D. Haynes and M. Côté, "Parallel fast fourier transforms for electronic structure calculations," *Comp. Phys. Commun*, 2000.

[92] L. J. Clarke, I. Štich, and M. C. Payne, "Large-scale ab initio total energy calculations on parallel computers," *Computer Physics Communications*, vol. 72, no. 1, pp. 14–28, Oct. 1992. [Online]. Available: http://www.sciencedirect.com/science/article/pii/001046559290003H

[93] S. A. Broughton and K. M. Bryan, *Discrete Fourier Analysis and Wavelets: Applications to Signal and Image Processing*. Hoboken, N.J: Wiley-Blackwell, Nov. 2008.

[94] D. Takahashi, A. Uno, and M. Yokokawa, "An Implementation of Parallel 1-D FFT on the K Computer," in *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, Jun. 2012, pp. 344–350.

[95] D. Takahashi, "Implementation of Parallel 1-D FFT on GPU Clusters," in *2013 IEEE 16th International Conference on Computational Science and Engineering (CSE)*, Dec. 2013, pp. 174–180.

[96] "Nukada FFT library." [Online]. Available: http://matsu-www.is.titech.ac.jp/~nukada/nufft/

[97] A. Nukada and S. Matsuoka, "Auto-tuning 3-D FFT Library for CUDA GPUs," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 30:1–30:10. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654090

[98] "GPUFFTW." [Online]. Available: http://gamma.cs.unc.edu/GPUFFTW/

[99] "ACL - AMD Compute Libraries." [Online]. Available: http://developer.amd.com/tools-and-sdks/opencl-zone/acl-amd-compute-libraries/

[100] M. F. Berman, "A Method for Transposing a Matrix," *J. ACM*, vol. 5, no. 4, pp. 383–384, Oct. 1958. [Online]. Available: http://doi.acm.org/10.1145/320941.320952

[101] P. F. Windley, "Transposing Matrices in a Digital Computer," *The Computer Journal*, vol. 2, no. 1, pp. 47–48, Jan. 1959. [Online]. Available: http://comjnl.oxfordjournals.org/content/2/1/47

[102] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE Comput. Soc, 1999, pp. 285–297. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=814600

[103] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[104] A. El-Moursy, A. El-Mahdy, and H. El-Shishiny, "An Efficient In-place 3d Transpose for Multicore Processors with Software Managed Memory Hierarchy," in *Proceedings of the 1st International Forum on Next-generation Multicore/Manycore Technologies*, ser. IFMT '08. New York, NY, USA: ACM, 2008, pp. 10:1–10:6. [Online]. Available: http://doi.acm.org/10.1145/1463768.1463781

[105] B. Catanzaro, A. Keller, and M. Garland, "A Decomposition for In-place Matrix Transposition," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 193–206. [Online]. Available: http://doi.acm.org/10.1145/2555243.2555253

[106] G. Ruetsch and M. Fatica, *CUDA Fortran for Scientists and Engineers: Best Practices for Efficient CUDA Fortran Programming*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[107] G. L. Yuan, A. Bakhoda, and T. M. Aamodt, "Complexity Effective Memory Access Scheduling for Many-core Accelerator Architectures," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 34–44. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669119

[108] J. L. Jodra, I. Gurrutxaga, and J. Muguerza, "Efficient 3d Transpositions in Graphics Processing Units," *Int J Parallel Prog*, pp. 1–16, Apr. 2015. [Online]. Available: http://link.springer.com/article/10.1007/s10766-015-0366-5

[109] "Bull Welcome - bullx supercomputer suite." [Online]. Available: http://www.bull.com/bullx-supercomputer-suite

[110] "InfiniBand® Trade Association: Home." [Online]. Available: http://www.infinibandta.org/

[111] "Mellanox Technologies: InfiniBand Performance Benchmarks." [Online]. Available: http://www.mellanox.com/page/performance_infiniband

[112] "Open MPI: Version 1.6.5." [Online]. Available: https://www.open-mpi.org/software/ompi/v1.6/

[113] "Drivers - Download NVIDIA Drivers." [Online]. Available: http://www.nvidia.es/Download/index.aspx?lang=es-es

[114] J. Bruck, C.-T. Ho, E. Upfal, S. Kipnis, and D. Weathersby, "Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 11, pp. 1143–1156, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1109/71.642949

[115] M. Frigo and S. G. Johnson, "FFTW for version 3.3.3," Nov. 2012. [Online]. Available: http://www.fftw.org/fftw3.pdf

Acrónimos

A

ALU
Arithmetic Logic Unit

API
Application Programming Interface

B

BLAS
Basic Linear Algebra Subprograms

C

CC
Compute Capability

CPU
Central Processing Unit

CU
Compute Unit

cuBLAS
CUDA Basic Linear Algebra Subprograms

CUDA
Compute Unified Device Architecture

cuFFT
CUDA Fast Fourier Transform

D

DFT
Discrete Fourier Transform

DMA
Direct Memory Access

DRAM
Dynamic Random Access Memory

DSP
Digital Signal Processor

DTFT
Discrete-Time Fourier Transform

F

FFT
Fast Fourier Transform

FFTW
Fastest Fourier Transform in the West

FMM
Fast Multipole Method

FPGA
Field Programmable Gate Array

FS
Fourier Serie

FT
Fourier Transform

G

GC

Gradiante Conjugado

GPC

Graphics Processing Cluster

GPGPU

General-Purpose computing on GPU

GPU

Graphics Processing Units

H

HPC

High Performance Computing

HW

Hardware

I

IB

InfiniBand

IBM

International Business Machines Corp.

IDFT

Inverse Discrete Fourier Transform

IFT

Inverse Fourier Transform

ISF

Interpolating Scaling Functions

M

MG

MultiGrid

MPI

Message Passing Interface

N

NIC

Network Interface Card

O

OpenCL

Open Computing Language

OpenGL

Open Graphics Library

P

PC

Personal Computer

PCI

Peripheral Component Interconnect

PCIe

Peripheral Component Interconnect Express

PDE

Partial Differential Equation

PFFT

Parallel Fast Fourier Transform

R

RAM

Random Access Memory

RDMA

Remote Direct Memory Access

S

SGEMM

Single precision floating General Matrix Multiply

SIMD

Single Instruction Multiple Data

SIMT

Single Instruction Multiple Threads

SM

Stream Multiprocessor

SMM

Maxwell SM

SMX

Next-Generation SM

SP

Stream Processor

SPA

Scalable Processor Array

SPMD

Single Program Multiple Data

SSD

Solid State Disk

SW

Software

T

TPC

Texture Processor Cluster

U

UVA

Unified Virtual Addressing

V

VPU

Vector Processing Unit

Glosario

A

ALU

Arithmetic Logic Unit
Es un circuito digital que calcula operaciones aritméticas (como suma, resta, multiplicación, etc.) y operaciones lógicas (si, y, o, no), entre valores (generalmente uno o dos) de los argumentos

API

Application Programming Interface
Conjunto de subrutinas, funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción

B

BLAS

Basic Linear Algebra Subprograms
Librería para calcular funciones para operaciones vectoriales y matriciales básicas

C

CC

Compute Capability
Representado por un número de versión, identifica cuáles son las características soportadas por el hardware de una GPU que utiliza CUDA

CPU

Central Processing Unit
La parte de una computadora en la que se encuentran los elementos que sirven para procesar datos

CU

Compute Unit
Conjunto de unidades SIMD en GPU de AMD

cuBLAS

CUDA Basic Linear Algebra Subprograms
Librería de NVIDIA CUDA para calcular funciones para operaciones vectoriales y matriciales básicas

CUDA

Compute Unified Device Architecture
Compilador y conjunto de herramientas de desarrollo creadas por NVIDIA que permiten a los programadores usar una variación del lenguaje de programación C para codificar algoritmos en GPU de NVIDIA

cuFFT

CUDA Fast Fourier Transform
Librería de NVIDIA CUDA para calcular la FFT

D

DFT

Discrete Fourier Transform
Transformada discreta utilizada en el análisis de Fourier

DMA

Direct Memory Access
Permite a cierto tipo de componentes de una computadora acceder a la memoria del sistema para leer o escribir independientemente de la CPU principal

DRAM

Dynamic Random Access Memory
Tipo de tecnología de memoria RAM basada en condensadores, los cuales pierden su carga progresivamente, necesitando de un circuito

dinámico de refresco que, cada cierto período, revisa dicha carga y la repone en un ciclo de refresco

DSP

Digital Signal Processor
Es un sistema basado en un procesador o microprocesador que posee un conjunto de instrucciones, un hardware y un software optimizados para aplicaciones que requieran operaciones numéricas a muy alta velocidad

DTFT

Discrete-Time Fourier Transform
Transformada en tiempo discreto utilizada en el análisis de Fourier

E

ecuación diferencial

Ecuación en la que intervienen derivadas de una o más funciones desconocidas

F

FFT

Fast Fourier Transform
Algoritmo eficiente que permite calcular la transformada de Fourier discreta (DFT) y su inversa

FFTW

Fastest Fourier Transform in the West
Librería software para calcular la DFT

FMM

Fast Multipole Method
Técnica numérica desarrollada para acelerar el cálculo en los problemas de los n-cuerpos

FPGA

Field Programmable Gate Array
Es un dispositivo semiconductor que contiene bloques de lógica cuya interconexión y funcionalidad puede ser configurada *in situ* mediante un lenguaje de descripción especializado.

FS

Fourier Serie
Serie infinita que converge puntualmente a una función periódica y continua a trozos (o por partes)

FT

Fourier Transform
Transformación matemática empleada para transformar señales entre el dominio del tiempo (o espacial) y el dominio frecuencial

G

GC

Gradiante Conjugado
Algoritmo para resolver numéricamente los sistemas de ecuaciones lineales cuyas matrices son simétricas y definidas positivas

GPC

Graphics Processing Cluster
Puede considerarse una GPU en si mismo. Lo poseen todas las tarjetas de NVIDIA a partir de la arquitectura Fermi. Está compuesto por el motor raster y hasta cuatro SM

GPGPU

General-Purpose computing on GPU
Concepto relativamente reciente dentro de informática que trata de estudiar y aprovechar las capacidades de cómputo de una GPU

GPU

Graphics Processing Units
Co-procesador dedicado al procesamiento de gráficos u operaciones de coma flotante, para aligerar la carga de trabajo del procesador central en aplicaciones como los videojuegos o aplicaciones 3D interactivas

H

hardware

Hardware
Se refiere a todas las partes físicas de un sistema informático; sus componentes son: eléctricos, electrónicos, electromecánicos y mecánicos

HPC

High Performance Computing
Herramienta muy importante en el desarrollo de simulaciones computacionales a problemas complejos. Utiliza tecnologías computacionales como los clústeres, supercomputadores o mediante el uso de la computación paralela.

I

IB

InfiniBand
Bus de comunicaciones serie de alta velocidad, baja latencia y de baja sobrecarga de CPU, diseñado tanto para conexiones internas como externas

IBM

International Business Machines Corp.
Empresa multinacional estadounidense de tecnología y consultoría con sede en Armonk, Nueva York. IBM fabrica y comercializa hardware y software para computadoras, y ofrece servicios de infraestructura, alojamiento de Internet, y consultoría en una amplia gama de áreas relacionadas con la informática, desde computadoras centrales hasta nanotecnología

IDFT

Inverse Discrete Fourier Transform
Inversa de la DFT

IFT

Inverse Fourier Transform
Inversa de la FT

ISF

Interpolating Scaling Functions
Método para obtener valores intermedios o valores fuera del intervalo para el que se dispone de datos

M

MG

MultiGrid
Grupo de algoritmos para solucionar ecuaciones diferenciales usando una jerarquía de discretizaciones

MPI

Message Passing Interface
Estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores

N

NIC

Network Interface Card
Periférico que actúa de interfaz de conexión entre aparatos o dispositivos

O

OpenCL

Open Computing Language
Consta de una interfaz de programación de aplicaciones y de un lenguaje de programación que permite crear aplicaciones con paralelismo a nivel de datos y de tareas que pueden ejecutarse tanto en CPU como en GPU

OpenGL

Open Graphics Library
Especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D

P

PC

Personal Computer
Es un tipo de microcomputadora diseñada en principio para ser utilizada por un solo humano a la vez. Habitualmente, la sigla PC se refiere a las computadoras IBM PC compatibles.

PCI

Peripheral Component Interconnect
Bus estándar de computadoras para conectar dispositivos periféricos directamente a la placa base

PCIe

Peripheral Component Interconnect Express
Nuevo desarrollo del bus PCI que usa los conceptos de programación y los estándares de comunicación existentes, pero se basa en un sistema de comunicación serie mucho más rápido

PDE

Partial Differential Equation
Es aquella cuyas incógnitas son funciones de diversas variables, con la peculiaridad de que en dicha ecuación figuran no solo las propias funciones sino también sus derivadas

PFFT

Parallel Fast Fourier Transform
Solución paralela para la FFT

R

RAM

Random Access Memory
Se utiliza como memoria de trabajo de computadoras para el sistema operativo, los programas y la mayor parte del software

RASTER

La **rasterización** es el proceso por el cual una imagen descrita en un formato gráfico vectorial se convierte en un conjunto de píxeles o puntos para ser desplegados en un medio de salida digital, como una pantalla de computadora, una impresora electrónica o una imagen de mapa de bits (bitmap)

RDMA

Remote Direct Memory Access
Consiste en el acceso directo desde la memoria principal de un ordenador a la de otro sin cooperación del sistema operativo

S

SGEMM

Single precision floating General Matrix Multiply
Algoritmo para multiplicar matrices con datos en coma flotante de

precisión simple. Es una operación que NVIDIA utiliza para medir el rendimiento de sus GPU

SIMD

Single Instruction Multiple Data
Técnica empleada para conseguir paralelismo a nivel de datos

SIMT

Single Instruction Multiple Threads
Es un modelo de ejecución paralela, usado en GPGPU, donde se simula la ejecución multihilo en procesadores SIMD

SM

Stream Multiprocessor
Es la parte de una GPU que ejecuta los núcleos de CUDA

SMM

Maxwell SM
Es el nuevo SM introducido en la microarquitectura Maxwell

SMX

Next-Generation SM
Es el nuevo SM introducido en la microarquitectura Kepler

software

Software
Equipo lógico o soporte lógico de un sistema informático, que comprende el conjunto de los componentes lógicos necesarios que hacen posible la realización de tareas específicas

SP

Stream Processor
Unidades de proceso que trabajan en paralelo y que ejecutan las instrucciones del núcleo que se ha asignado a un SM. También conocido como núcleo CUDA

SPA

Scalable Processor Array
Conjunto de SP organizados en SM

SPMD

Single Program Multiple Data
Las tareas se dividen y se ejecutan de forma simultánea en múltiples procesadores con diferentes datos de entrada con el fin de obtener resultados más rápidamente

SSD

Solid State Disk
Un tipo de dispositivo de almacenamiento de datos que utiliza memoria no volátil, como la memoria flash, para almacenar datos

T

TPC

Texture Processor Cluster
Conjunto de SM, unidades de textura y de control lógico

U

UVA

Unified Virtual Addressing
Proporciona un único espacio de memoria combinado para la memoria principal del sistema y las memorias de las GPU

V

VPU

Vector Processing Unit
Unidad de procesamiento que ejecuta instrucciones con operaciones vectoriales tanto con enteros como en coma flotante de precisión simple y doble

Índice alfabético

J

K

L

M

N

O

P

Q

R

S

T

U

V

W